Psychology of Programming Interest Group Annual Conference 2014

University of Sussex,
Old Ship Hotel, Brighton
25-27th June 2014

Proceedings

Edited by

Benedict du Boulay & Judith Good

MESSAGE FROM THE CHAIRS

Welcome to the Psychology of Programming Interest Group (PPIG) Annual Conference 2014. This is the 25th Anniversary of the first PPIG meeting. It is taking place at the Old Ship Hotel in Brighton on the 25th – 27th June 2014.

The Psychology of Programming Interest Group was established in 1987 in order to bring together people from diverse communities to explore common interests in the psychological aspects of programming and/or computational aspects of psychology. The group attracts cognitive scientists, psychologists, computer scientists, software engineers, software developers, HCI researchers etc, in both academia and industry.

This year the topics include Programming and Creation, Novice Learning, Cognitive Factors, Representations and Interfaces, and Social & Affective Issues. Authors will have come from around the world including Australia, Brazil, Canada, Finland, Germany, The Netherlands, US and the UK.

The conference continues a tradition of hosting a Doctoral Consortium specifically to enable research students in the relevant disciplines to come together, give presentations and exchange ideas. We thank Jim Buckley of the University of Limerick for managing the consortium.

We have invited speakers from academia and industry: Robert Biddle from Carleton University, Canada; Marian Petre from the Open University, UK; Alan Blackwell from Cambridge University, UK; and David Gilmore from GE Global Research.

We are grateful to all the members of the Programme Committee, Steering Group and the Doctoral Consortium Chair for their hard work preparing this high quality technical programme, and to our student helpers for making the event run smoothly.

Benedict du Boulay & Judith Good

COMMITTEES

Local Organisers and Programme Chairs

Benedict du Boulay and Judith Good
Department of Informatics, University of Sussex, UK

Doctoral Consortium Chair

Jim Buckley, University of Limerick, Ireland

Steering Group

Benedict du Boulay, Judith Good, Thomas Green and Maria Kutar

Programme Committee

Rachel Bellamy, IBM Research, USA
Roman Bednarik, University of Eastern Finland, Finland
Moti Ben-Ari, Weizmann Institute of Science, Israel
Alan Blackwell, Cambridge University, UK
Richard Bornat, University of Middlesex, UK
Jim Buckley, University of Limerick, Ireland
Paul Cairns, University of York, UK
Steven Clarke, Microsoft Scotland, UK
Sylvia Da Rosa, Universidad de la Republica, Uruguay
Chris Douce, Open University, UK
Alistair Edwards, University of York, UK
Michael English, University of Limerick, Ireland
Chris Exton, University of Limerick, Ireland
Thomas Green. University of York, UK
Yanguo Jing, London Metropolitan University, UK
Babak Khazaei, Sheffield Hallam University, UK
Charles Knutson, Brigham Young University, USA
Maria Kutar, University of Salford, UK
Louis Major, University of Cambridge, UK
Lindsay Marshall, Newcastle University, UK
Alex McLean, University of Leeds, UK
Marian Petre, Open University, UK
John Rooksby, University of Glasgow, UK
Jorma Sajaniemi, University of Eastern Finland, Finland
Helen Sharp, Open University, UK
Markku Tukiainen, University of Eastern Finland, Finland

INVITED SPEAKERS

Robert Biddle, Carleton University, Canada.
Marian Petre, Open University, UK.
Alan Blackwell, Cambridge University, UK.
David Gilmore, GE Global Research, USA.

INDEX BY SESSION

Cognitive Factors

Novice Learning

Social and Affective Issues

Representations and Interfaces

Programming and Creation

INDEX BY FIRST AUTHOR

Cognitive Flexibility and Programming Performance

Marianne Leinikka

Finnish Institute of Occupational Health
Helsinki, Finland

marianne.leinikka@ttl.fi

Arto Vihavainen

Department of Computer Science
University of Helsinki, Finland

arto.vihavainen@helsinki.fi

Jani Lukander

Finnish Institute of Occupational Health
Helsinki, Finland

jani.lukander@ttl.fi

Satu Pakarinen

Finnish Institute of Occupational Health
Helsinki, Finland

Faculty of Behavioral Science
University of Helsinki, Finland

satu.pakarinen@ttl.fi

Keywords: POP-V.A. Cognitive Theories, POP-IV.F. Exploratory

Abstract

Cognitive flexibility is an integral part in adaptive behavior such as learning, and can be seen as one explaining factor in performance in various tasks. In this study, we explore the possibilities of a dynamic psychological test that can be administered online to assess cognitive flexibility and apply the test across a student population that is learning to program. While our results imply that cognitive flexibility has little correlation with the score of a traditional pen-and-paper programming exam as well as the students' average grade from their first semester of studies, cognitive flexibility does play a role in the efficiency with which students solve programming errors. Moreover, our results imply that while cognitive flexibility correlates with the students' efficiency in solving programming errors, the correlation is more evident with novice programmers than with programmers with existing experience.

1. Introduction

Cognitive flexibility, the ability to switch between modes of thought and to simultaneously think about multiple concepts, is an essential part of adaptive behavior, such as flexible everyday functioning and learning (Boger-Mehall 1996). While the underlying processes such as attention and short-term memory evolve over age from infants to old age (Anguera et al. 2013, Crone et al. 2006) and vary considerably between individuals of same age and culture (Daneman and Carpenter 1980), the cognitive performance also varies as a consequence of stress (Olver et al. 2014, Aggarwal et al. 2014), fatigue and sleep deprivation (Fogt et al. 2010, Sallinen et al. 2013), as well as environmental disturbances such as interruptions and background noise (Banbury and Berry 1998, Reynolds et al. 2013).

As learning has become a lifelong task, measuring cognition has become increasingly popular for promoting the productivity and well-being of learning, working, and aging citizens. Cognitive flexibility has been traditionally measured with neuropsychological tests such as the Trail Making test (Reitan 1955, Reitan 1958) and The Stroop Test (Stroop 1935), which are, however, targeted for detecting rather gross developmental or acquired impairments in clinical conditions, and require extensive expertise and resources that are limited in the healthcare and educational systems. On the other hand, more sensitive and more specific cognitive tests for evaluating mild cognitive

impairments or cognitive processes in normal population exist, but those are developed for research purposes and require the acquisition of expensive exclusive software platforms.

Another way to measure cognitive flexibility is by means of task-switching paradigms. Several variations of the task-switching paradigms exist (Dibbets and Jolles 2006, Gupta et al. 2009, Kopp and Lange 2013), but in general, participants are put to work in a controlled environment, where frequent but random shifts between two tasks are required. The paradigm is based on two main principles: first, people are faster and more accurate when performing a task repeatedly; this phenomenon is called *repetition benefit*, and second, when changing the task or switching between tasks one's performance becomes slower and less accurate/more erroneous; described with the *switch cost* and *mixing cost* (for a review on task-switching, see Monsell 2003).

When considering computer programming, a considerable amount of effort has been invested in identifying factors and creating tests that can be used to seek out students who e.g. struggle with learning to program. While some discussions on the cognitive aspects exist (Parnin 2010), typical approaches include programming aptitude tests (Evans and Simkin 1989, Tukiainen 2002), tests that measure skills such as mathematical and spatial reasoning (White and Sivitanides 2003, Fincher et al. 2006), and the use of different self-perception metrics to detect students likely to succeed or fail (Bergin and Reilly 2006, Ramalingam et al. 2004). With the improved computing facilities, more opportunities have arisen due to the ability to gather data from students' programming process, making it possible to capture and analyze programming sessions (Jadud 2006, Rodrigo et al. 2009, Watson et al. 2013).

In this work, we contribute to the understanding of the link between programming performance and cognitive flexibility, as well as provide researchers with a tool for measuring cognitive flexibility. Our study implies that while results from a cognitive flexibility test do not correlate with traditional course outcome metrics such as the result from a pen-and-paper exam, cognitive flexibility plays a role in the efficiency with which students solve programming errors. Moreover, cognitive flexibility correlates more strongly with the performance of novice programmers than with the performance of more experienced programmers, suggesting that programming experience influences the results.

The remainder of the paper is structured as follows. Next section discusses the research methodology, target groups, and the system used in more detail, after which we discuss the test results and provide a detailed analysis. In the fourth section, we discuss the limitations of our study, and finally, in the fifth section, we conclude with an overview of the work and outline future work.

2. Research Method and Materials

Our study consists of two main questions, where we first identify whether the task-switching test that is administered online performs comparably with existing task-switching studies. Once the first question has been investigated, we seek for connections between task-switching performance and programming performance.

RQ 1: Are the results from the task-switching game comparable with existing task-switching studies?

RQ 2: What correlation is there between task-switching results and programming performance? More explicitly, how do the task-switching results correlate with...

RQ 2.1: … error-solving velocity?

RQ 2.2: … programming course exam scores?

RQ 2.3: … overall performance during the first semester of studies?

The study is based on two datasets from courses offered by the University of Helsinki. The first dataset is from a programming course targeted for University students that was held during fall 2013, while the second dataset is from an open programming course from spring 2014 that was offered for free for everyone. While the University students who participated in the study completed the game in a class with a teacher and received a small chocolate bar as a token of appreciation for participation,

the participants from the open programming course have had no additional motivating factors for completing the game outside the course material asking them to participate in ongoing research.

For the RQ 2.1., we use snapshot data gathered from both programming courses, while for RQ 2.2 and RQ 2.3, snapshot data from only the course offered for the University students is used as there is no formal exam procedure in the open programming course. Linear regression analysis is performed on the results from the task-switching study and the snapshot data used in each research question.

2.1 Test Procedure

The task that is used for this study is a modified three-phased version of a Number-Letter task (Vandierendonck et al. 2010, Pashler 2000). The test utilizes a defined subset of Arabic numbers (2-9) and Latin letters (a, e, g, i, k, m, r, u) that are commonly used in the Anglo-American, German and Fenno-Ugric language groups. Letter-number pairs (e.g. a7) are presented to the participant either above or below a stationary horizontal line, and to avoid the possibility of the participants fixating on a certain location a horizontal jitter is used. The responses are given from the keyboard by pressing either the X or M key. The test has three tasks, and at the end of each task, the percentage as well as the mean value of reaction time for correct answers is shown to the participant.

In the first task, the detection task, a baseline *reaction time* is measured by requesting the subject to press key X as soon as they see a stimulus on screen. The detection task consists of 10 trials. The letter-number pairs are visible until the response, but for a maximum of 1950 ms. Button presses 40-1950 ms post-stimulus onset are accepted as responses. The responses up to 39 ms post-stimulus onset are discarded from the data to discount possible mis-hits.

The second task measures *categorization*. First, letter-number pairs are presented above the horizontal line and the task is to categorize the number of the pair as either odd or even by pressing X or M, respectively. Thereafter, stimulus pairs are presented below the horizontal line and the task is to categorize the letter of the pair as either consonant or vowel by pressing X or M, respectively. The letter-number pairs are quasi-randomized so that an equal amount of each letter and number will appear but in randomized pairs of all possible combinations. The categorization task consists of 80 trials, first 40 above and then 40 below the line.

Finally, in the third task, the *task-switching* task, the participant performs both the categorization tasks alternatingly. The letter-number pairs are presented alternatingly above or below the horizontal line. The task changes according to the location cue: when the letter-number pair is presented above the horizontal line, the task is to classify shown numbers as odd or even, and when the pair is presented below the horizontal line, the task is to classify the letter of the pair as consonant or vowel, by pressing either X or M, respectively.

The task-switching task consists of 168 stimulus pairs with 63 task switches. 1 to 6 stimulus pairs are presented consecutively below or above the line before a switch. There are 23 trials consisting of a single pair before switch, 10 trials constituting 2, 3, and 4 repetitions before switch, each, and 5 trials of 5, and 6 repetitions, each. The number of repetitions before switch was pseudorandomized.

In the Categorization and the Task-switching tasks, the letter-number pairs are presented until the participant's response, but for a maximum of 2500 ms. The rate of stimulus presentation is tied to the subjects responses in the following way: The successive stimulus is presented 150 ms after a correct response, and 1500 ms after an incorrect or missed response. Key presses occurring between 40 ms and 2500 ms from the stimulus onset are accepted as responses.

The participants are instructed that both speed and accuracy in the task are equally important. The participants are obliged to briefly practice each of the tasks (excluding the Detection task) at least once but a maximum of three times before the actual task. Each practice round contains six letter-number pairs. As our task-switching task has been accommodated for classroom use, it is notably shorter when compared to the ones used in previous studies, and lasts 7-10 minutes on average.

2.2 System Description

For the participants, the test is provided as an online test that can be linked e.g. from course material or via an email. Source code, hosting instructions, and an online version are available at http://github.com/measureself/cognitive-flexibility

When considering the system architecture, the front-end of the system has been developed as a single-page web application that can be used both in mobile phones and desktop computers (the mobile version requires an additional application in the mobile phone that acts as a wrapper in order to accommodate touch-screen user interfaces). The application has been built using modern web technologies (HTML5, CSS3 and JavaScript), and it uses jQuery to avoid cross-platform issues. The system has been designed to be easily extendable, and provides easy access points for changing the language, alphabet and input configuration used, as well as for changing other configuration parameters such as the response windows.

The backend implementation of the system is abstracted behind a REST-API, which means that the storage implementation can be changed if deemed necessary without modifications to the front-end. All data is transferred using JSON, and the existing backend has been implemented using Java EE and Spring Framework. In addition to Spring Framework, EclipseLink is used to abstract the data layer, making it easier to use different databases. Currently, by default, the system uses a MySQL-server as the database.

When asking users to take the test, one can either provide the users with an URL that contains the desired username, which automatically will then be used for identification, or, users can be asked to sign into the system via a front page. The authentication mechanism provided by the backend is modular, and additional mechanisms such as support for a third party authentication service can be added in a straightforward fashion. In the current version, the system only stores the desired username and related results, and does not ask for any additional information. Hence, mapping demographic information should be done separately, or added to the system if needed.

In order to measure reaction times accurately, the system utilizes high resolution time support whenever available in the browser. The *Performance.now()*-interface provided by the high resolution time support gives the browser the access to the current time in sub-millisecond precision, which has previously been impossible. Naturally, we acknowledge that the system that the participant has, and the load on participants' systems also affects the timing. This is partially solved by gathering information on the user's system (user agent, window size), but, in the current version, no feasible means for measuring the load on the user's machine exist.

3. Evaluation and Discussion

For evaluation, a total of four studies are conducted. In the first study, we verify that the results obtained from the system are compatible with existing task-switching studies, and in the three subsequent studies we investigate the correlations between the task-switching performance and (1) error solving velocity, (2) programming course exam scores, and (3) credits and average grade from the first semester.

3.1 Are the results from our system comparable with existing task-switching studies?

To answer RQ 1, a total of 298 participants completed the entire task-switching game. Of those, 15 participants were excluded from the data due to performing at or below the level of 75% (Monsell 2003) and five participants due to missing demographic information. Of the remaining 278 participants (mean 25.7 years, $\sigma = 8.6$) 39 were female (mean 25.1 years, $\sigma = 5.4$). Tables 1 and 2 present the mean values and the standard deviations for the Detection task and the Categorization task and the corresponding values for the Task-Switching task. The hit rates, false responses, and reaction times in the task-switching task as a function of stimulus pair repetitions after switch are shown in Figure 1.

	Detection task	Categorization task
Hits (%)	96.3, σ = 17.4	94.7, σ = 5.2
False responses (%)	-	5.1, σ = 5.1
Misses (%)	3.7, σ = 17.4	1.1, σ = 0.5
RT for correct answers (ms)	434.4, σ = 141.4	640.2, σ = 74.4
RT for incorrect answers (ms)	-	628.6, σ = 137.0

Table 1 – The mean values and the standard deviations of key parameters for the Detection and the Categorization task.

	Task-switching task Stimulus pair position after switch					
	1	2	3	4	5	6
Hits (%)	87.3, σ = 9.4	92.1, σ = 9.1	94.6, σ = 6.9	95.6, σ = 8.4	93.1, σ = 11.1	95.1, σ = 17.1
False responses (%)	9.4, σ = 8.1	5.5, σ = 7.5	4.1, σ = 5.8	3.4, σ = 7.5	6.3, σ = 10.6	4.1, σ = 15.1
Misses (%)	3.2, σ = 4.5	2.5, σ = 5.3	1.3, σ = 3.2	1.0, σ = 3.6	0.6, σ = 3.3	0.7, σ = 6.0
RT for correct answers (ms)	1193.3, σ = 179.7	863.3, σ = 158.4	776.4, σ = 127.9	778.2, σ = 156.6	793.1, σ = 168.4	796.4, σ = 228.4
RT for incorrect answers (ms)	1160.7, σ = 396.3	1111.9, σ = 458.9	964.6, σ = 399.6	890.0, σ = 376.1	862.3, σ = 347.0	731.5, σ = 325.6

Table 2 – The mean values and the standard deviations of key parameters for the Task-switching task as a function of stimulus pair position after switch.

Within the task-switching task, the reaction time for correct responses immediately after the switch (position 1) differed from the mean RT for repetitions (hits for the stimulus pairs in positions 3-6): $t277 = 50.2$, $P < 0.001$. Similarly, the reaction time for correct responses at position 2 after the switch differed from the mean RT for repetitions (positions 3-6): $t277 = 13.6$, $P < 0.001$. This switch cost was on average 424 ms (σ = 141 ms), i.e., 56.4% for position 1 and 94 ms (σ = 115 ms), i.e., 12.6% for position 2 after the switch.

When comparing the Task-switching task and the Categorization task, the mean reaction time for hits after the switch (mean of positions 3-6) differed from the reaction time for hits in the Categorization task: $t277 = 21.4$, $P < 0.001$. This mixing cost was on average 129 ms (σ = 100 ms), i.e., 20.4% for the mean of positions 3-6 after the switch.

As expected, switching between tasks results in poorer accuracy (decreased hit rate) and slower response (increased reaction time; Figure 1). This effect is strongest immediately after switch and decreases with task repetition: the switch cost for the stimulus position 1 after the switch was 56.4% and for position 2, 12.6%. The decrease in reaction time already between the stimulus pair immediately after switch (position 1) and the pair following it (position 2) suggests that the studied participants are able to recover fast from the switch and inhibit the irrelevant task rule.

Even though repetition of the same task results in participants being faster and more accurate the more the task is repeated in succession the performance in the Task-switching task never reaches the level of speed and accuracy in the Categorization task. This mix cost for the mean positions 3-6 was 20.4%, indicating that despite repetition the categorization was more difficult within the Task-switching task

as compared to the simple Categorization task. The benefit of repetition is also seen in the proportion of false responses, i.e., the error rate. The error rate was higher immediately after the switch compared to the stimulus pair in positions 3-6 (9.4%, σ = 8.1 and 4.3%, σ = 5.0, respectively; Figure 1). The results are in accordance with the previous studies (Monsell 2003).

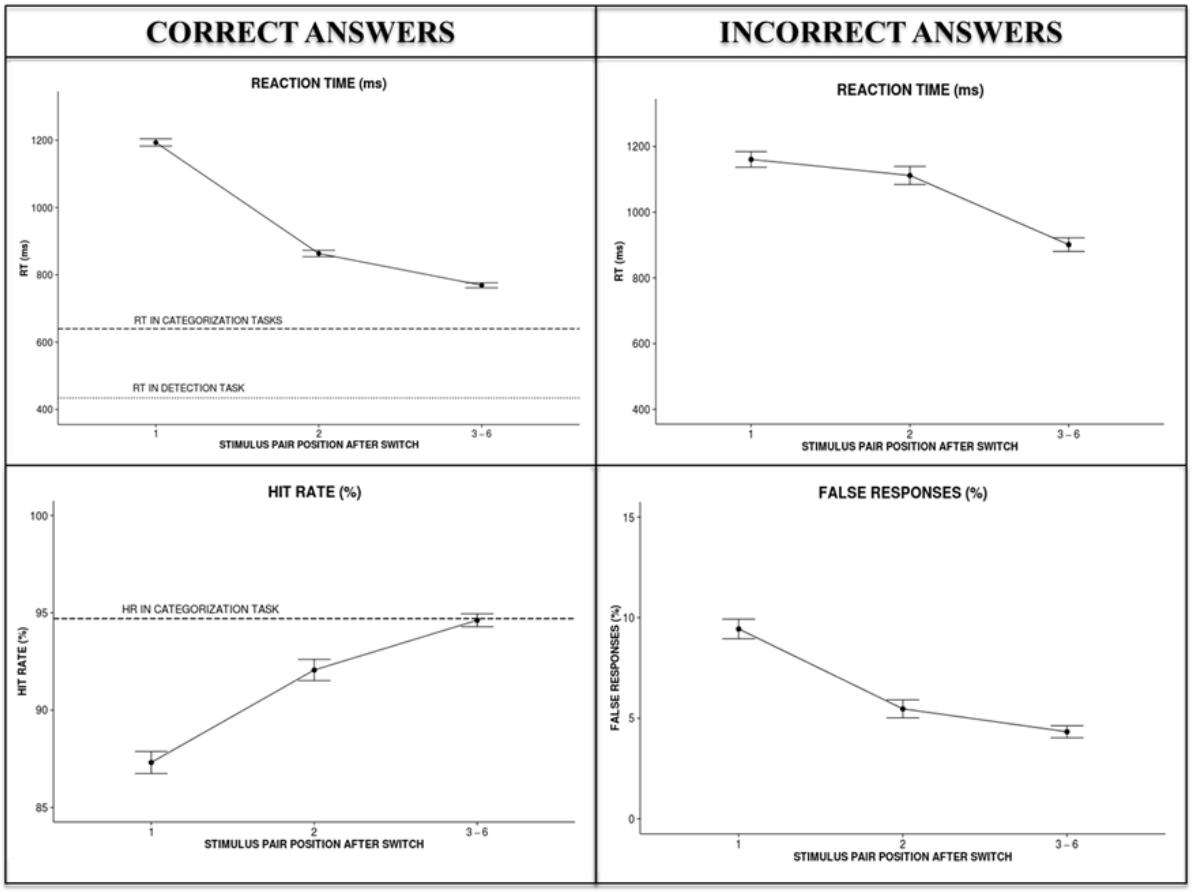


Figure 1 – Reaction times, hits and false responses of 278 participants as a function of stimulus pair repetitions after switch in the Task-switching task. The upper panels in Figure 1 represent the mean reaction times for correct (left panel) and incorrect (right panel) responses as a function of stimulus pair position (1-6) after switch. The lower panels represent the proportion of correct (left panel) and incorrect (right panel) answers as a function of stimulus pair position (misses not shown). Error bars denote the standard error of mean (S.E.M.). For comparison, the mean reaction time (RT) in the Detection task (dotted line) and in the Categorization task (dashed line) are presented in the upper left panel. Also the proportion of correct answers in the Categorization task (dotted line) is shown in the lower left panel.

3.2. Are the task-switching results correlated with programming performance?

To answer the RQ 2.1; "Are the task-switching results correlated with performance in solving programming errors?" we investigate key-level snapshot data that has been extracted as students program in the programming courses. The snapshots are from participants that have worked on a problem where the goal is to create a data repository for bird observations. The problem is open ended, which means that the structure of the student's solution is not helped or enforced in any way, and the criterion for success is that the student's application fulfills the functionality that is provided as the following textual input-output description (inputs from the user are denoted with italics) with additional notes on the different commands needed.

```
? Add
Name: Raven
Latin Name: Corvus Corvus
? Add
Name: Seagull
Latin Name: Dorkus Dorkus
? Observation
What was observed:? Seagull
? Observation
What was observed:? Turing
Is not a bird!
? Observation
What was observed:? Seagull
? Statistics
Seagull (Dorkus Dorkus): 2 observations
Raven (Corvus Corvus): 0 observations
? Show
What? Seagull
Seagull (Dorkus Dorkus): 2 observations
? Quit
```

We consider compilation errors in the snapshots (e.g. incorrect type, missing class, missing semicolon) as programming errors, and utilize the median time that it takes for the participant to solve compilation errors as the measure of performance. Participants have been asked about their programming background, and participants have been split into two categories based on their answers.

In the dataset with data from students that have worked on the bird repository problem as well as answered the task-switching questionnaire, there are 45 participants with no programming experience, while 65 participants have previous programming experience.

	r (Pearson) no programming experience, N=45	*r (Pearson)* existing programming experience, =65
RT in Detection task	.18	.18
RT in Categorization task	.39**	.25*
Switch cost for position 1	-.32*	-.12
Switch cost for position 2	-.21	-.09
Total RT for Task-Switching task	.54**	.30*
RT for correct answers in Task-Switching task	.57**	.32**
RT for repetition in Task-Switching task	.57**	.36**
Mixing cost	.47**	.16
RT-Task-Switching-minus-RT-Detection	.35*	.12

(* $p < 0.05$, ** $p < 0.01$)

Table 3 –Pearson product moment correlation coefficients for various aspects of task-switching and time taken by participants to solve programming errors, classified by programming experience.

In Table 3, we can observe that the correlations between the median problem solving times are stronger in the population that have had no previous programming experience before the start of the course; the task switching correctness shows moderate correlation with the median speed for solving a programming error, while other indicators show low to no correlation. For the population that had previous programming experience, the correlations, when existing, are low, as their results are more heavily influenced by their programming experience. As the reaction time in the detection task does not correlate with the participants' problem solving speed, we can assume that the correlations are not explained by reaction time.

Next, we will investigate RQ 2.2 “Are the task-switching results correlated with performance in programming course exams?” and RQ 2.3 “Are the task-switching results correlated with overall performance in the first semester?”. For the course exam performance, we observe the points that students have received in a written paper exam (0-30), while for the overall performance in the first semester, we look at the number of study credits and average grade that the students have received during their first semester.

In the dataset used for RQ 2.2 and RQ 2.3, we have received the study records from 45 students who both participated in the task-switching study during their first semester, and agreed to their information being used for the study.

	Performance in programming course exam	Number of study credits received during the first semester	Average grade during the first semester
	r (Pearson)	*r (Pearson)*	*r (Pearson)*
RT in Detection task	-.11	.15	-.15
RT in Categorization task	-.15	.05	-.13
Switch cost for position 1	.08	-.08	.12
Switch cost for position 2	.06	-.05	.01
Total RT for Task-Switching task	-.12	-.09	-.32*
RT for correct answers in Task-Switching task	-.14	-.05	-.34*
RT for repetition in Task-Switching task	-.19	-.02	-.38*
Mixing cost	-.13	-.08	-.34*
RT-Task-switching-minus-RT-Detection	-.01	-.16	-.15

(* $p < 0.05$)

Table 4 – Pearson product moment correlation coefficients for various aspects of task-switching and performance in a written programming course exam, number of study credits received during the first semester, and the average grade during the first semester.

In Table 4 we can observe that the task-switching performance has no correlation with the performance in the course exam and the amount of credits that the students receive during the first semester. However, low negative correlation can be observed between the average grade from the first semester and the third task-switching task. One plausible explanation for the negative correlation, while hypothetical at this point, is that students with higher cognitive flexibility have learned in their previous studies (e.g. in high-school) that they do not have to work too hard to succeed at some level. If they have entered their university studies with the same expectation, it is possible that the lack of effort can be visible in the first semester grades.

4. Limitations of Study

Our assumption for the study was that the majority of study participants come from a homogeneous population (i.e. the participants are university students and people interested in learning programming). When considering the previous programming experience, variance in existing computing skills was not taken into account outside the reported programming experience. This reduces the internal validity as it is possible that other attributes such as the tendency to play computer games may explain some of the results. In our current study, we assume that both populations have people who also play computer games. As it is possible that this is not the case, we wish to stress that

one should seek for additional background variables; in our case the obvious "hindsight" was that we should have asked the participants to elaborate on their programming experience and whether they spend time playing e.g. computer games. Similarly, although cognitive flexibility varies over age, we did not consider the participants' ages in the study, as we did not focus on students with learning impairments.

Another limitation is that our data comes from two teaching contexts (local, online) that both follow the same pedagogy and material (for details on the pedagogy, see Vihavainen et al. 2011). While every teacher has their own approach to teaching programming, an objects-early -approach is used at the University of Helsinki. In addition, the students are guided in local labs, and students tend to form study groups before the exams to study "exam-related" -topics. That is, the correlations may differ when the study is replicated in another teaching context. However, correlations from all tests were reported, which makes replication easier.

We assume that some variance will be visible depending on the teaching approach, language used, tools used, and naturally the student population. However, we sought to remedy parts of this by using linear regression for fitting the models. While this may have led to losing some insight that could have been gained with e.g. polynomial regression, this deliberately avoids overfitting and thus our results can be used as a baseline for future investigation.

5. Conclusions and Future Work

We have described initial steps in measuring cognitive flexibility using an online task-switching test, and sought out correlations between different aspects in learning to program. Cognitive flexibility was chosen over other measures such as IQ as it provides a lens to abilities which may be related to computer programming; the ability of switching between modes of thought and the ability of thinking about multiple concepts simultaneously. We first verified that the results from the task-switching game are comparable with existing studies, after which we sought correlations between task-switching performance and programming performance. While the correlation between the performance in solving programming errors and the task-switching performance was significant, the outcome of a paper exam and first-semester studies had practically no correlation with the task-switching performance.

We acknowledge that the conditions and teaching context, such as the programming language, pedagogy, and different support mechanisms used differ considerably across different teaching contexts. While we have taken initial steps to create understanding on how cognitive flexibility affects programming performance, further work is needed to determine the applicability of the task-switching test across different teaching contexts. Similarly, comparing its efficiency with other tests is needed.

As the system created for this study is published as an open-source project that is free-of-charge and also relatively device independent, gathering larger datasets from students in different contexts is straightforward. The system improves the existing opportunities on evaluating the effects of different demographic variables such as sex, age, cultural background and education not just on programming performance, but on performance in other learning contexts as well. On the basis of the test one can also draw a learning curve on successful repetitions of the test. Effects of fatigue, stress, or the time of day can be assessed by comparing different sessions from the same individuals or if the data set is sufficiently large, even as a cross-sectional study of different individuals, provided that information on their stress or time-of-day when taking the test is available for instance via a web-based survey.

Our work leaves several questions open. Possible directions that require effort from multiple parties include investigating whether the observations hold in (1) different teaching contexts (e.g. online, lab-centric environment, traditional classroom environment), (2) different populations, (3) how the existing predictors such as programming aptitude tests and IQ tests relate to task-switching performance, and (4) how successive repeated test sessions are visible in both the test results and possibly other factors.

6. Acknowledgements

This research has been funded by the Tekes project #337910202.

7. References

Aggarwal, N. T., Wilson, R. S., Beck, T. L., Rajan, K. B., de Leon, C. F. M., Evans, D. A., & Everson-Rose, S. A. (2014). Perceived Stress and Change in Cognitive Function Among Adults 65 Years and Older. Psychosomatic medicine, 76(1), 80-85.

Anguera, J. A., Boccanfuso, J., Rintoul, J. L., Al-Hashimi, O., Faraji, F., Janowich, J., ... & Gazzaley, A. (2013). Video game training enhances cognitive control in older adults. Nature, 501(7465), 97-101.

Banbury, S., & Berry, D. C. (1998). Disruption of office-related tasks by speech and office noise. British Journal of Psychology, 89(3), 499-517.

Bergin, S., & Reilly, R. (2005, June). The influence of motivation and comfort-level on learning to program. In Proceedings of the PPIG (Vol. 17, pp. 293-304).

Boger-Mehall, S. R. (1996). Cognitive flexibility theory: Implications for teaching and teacher education. In Society for Information Technology & Teacher Education International Conference (Vol. 1996, No. 1, pp. 991-993).

Crone, E. A., Wendelken, C., Donohue, S., van Leijenhorst, L., & Bunge, S. A. (2006). Neurocognitive development of the ability to manipulate information in working memory. Proceedings of the National Academy of Sciences, 103(24), 9315-9320.

Daneman, M., & Carpenter, P. A. (1980). Individual differences in working memory and reading. Journal of verbal learning and verbal behavior, 19(4), 450-466.

Dibbets, P., & Jolles, J. (2006). The switch task for children: Measuring mental flexibility in young children. Cognitive Development, 21(1), 60-71.

Fogt, D. L., Kalns, J. E., & Michael, D. J. (2010). A comparison of cognitive performance decreases during acute, progressive fatigue arising from different concurrent stressors. Military medicine, 175(12), 939-944.

Gupta, R., Kar, B. R., & Srinivasan, N. (2009). Development of task switching and post-error-slowing in children. Behavioral and Brain Functions, 5(38), 1-13.

Jadud, M. C. (2006, September). Methods and tools for exploring novice compilation behaviour. In Proceedings of the second international workshop on Computing education research (pp. 73-84). ACM.

Kopp, B., & Lange, F. (2013). Electrophysiological indicators of surprise and entropy in dynamic task-switching environments. Frontiers in human neuroscience, 7.

Monsell, S. (2003). Task switching. Trends in cognitive sciences, 7(3), 134-140.

Olver, J. S., Pinney, M., Maruff, P., & Norman, T. R. (2014). Impairments of Spatial Working Memory and Attention Following Acute Psychosocial Stress.Stress and Health.

Parnin, C. (2010). A cognitive neuroscience perspective on memory for programming tasks. In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG).

Pashler, H. (2000). 12 Task Switching and Multitask Performance. Control of cognitive processes, 277.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004, June). Self-efficacy and mental models in learning to program. In ACM SIGCSE Bulletin (Vol. 36, No. 3, pp. 171-175). ACM.

Reitan, R. M. (1955). The Relation of the Trail Making Test to Organic Brain Damage1. Group, 3(1), 1.

Reitan, R. M. (1958). Validity of the Trail Making Test as an indicator of organic brain damage. Perceptual and motor skills, 8(3), 271-276.

Reynolds, J., McClelland, A., & Furnham, A. (2013). An investigation of cognitive test performance across conditions of silence, background noise and music as a function of neuroticism. Anxiety, Stress & Coping, (ahead-of-print), 1-12.

Rodrigo, M. M. T., Baker, R. S., Jadud, M. C., Amarra, A. C. M., Dy, T., Espejo-Lahoz, M. B. V., ... & Tabanao, E. S. (2009). Affective and behavioral predictors of novice programmer achievement. ACM SIGCSE Bulletin, 41(3), 156-160.

Sallinen, M., Onninen, J., Tirkkonen, K., Haavisto, M. L., Härmä, M., Kubo, T., ... & Porkka-Heiskanen, T. (2013). Effects of cumulative sleep restriction on self-perceptions while multitasking. Journal of sleep research, 22(3), 273-281.

Stroop, J. R. (1935). Studies of interference in serial verbal reactions. Journal of experimental psychology, 18(6), 643.

Tukiainen, M., & Mönkkönen, E. (2002). Programming aptitude testing as a prediction of learning to program. In Proc. 14th Workshop of the Psychology of Programming Interest Group (pp. 45-57).

Vandierendonck, A., Liefooghe, B., & Verbruggen, F. (2010). Task switching: interplay of reconfiguration and interference control. Psychological bulletin,136(4), 601.

Vihavainen, A., Paksula, M., & Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. In Proceedings of the 42nd ACM technical symposium on Computer science education (pp. 93-98). ACM.

Watson, C., Li, F. W., & Godwin, J. L. (2013). Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on (pp. 319-323). IEEE.

White, G., & Sivitanides, M. (2003). An empirical investigation of the relationship between success in mathematics and visual programming courses. Journal of Information Systems Education, 14(4), 409-416.

Reasoning about Complexity – Software Models as External Representations

Simon C. Lynch

School of Computing
Teesside University
United Kingdom
s.c.lynch@tees.ac.uk

Joseph Ferguson

School of Education
Deakin University
Melbourne, Australia
jpfergus@deakin.edu.au

Keywords: POP-I.B. internal and external representations, POP-III.B. NetLogo, POP-V.A. mental models.

Abstract

Programming is traditionally considered to be an activity which aims only to produce a software artefact as its primary goal. With this view programming languages are simply the notations which define these artefacts. This paper examines the relationships between internal representations (mental models) and external representations (notations and other forms) arguing that program code behaves as an external representation in a similar way to mathematical or logical notations but with the added property that code can be executed and its notational consequences observed.

Furthermore some environments allow program operation to be manipulated at run-time; we propose that these systems also operate as external representations and that programming language statements and their run-time environments can thereby be utilised as reasoning systems to promote the exploration and discovery of new understandings. In this context we consider NetLogo as a framework for reasoning about complex and emergent systems, evaluating its suitability from a representational perspective.

Introduction

This work was originally motivated by the desire to find a notation suitable for representing and reasoning about complex and emergent systems. We require this representational framework to be capable of describing executable computer models of these complex systems. The target audience for using this framework is broad, ranging from senior secondary school pupils to postgraduates. While we assume basic numeracy and some ability to reason logically, we do not rely on prior exposure to specific programming concepts from our users.

The application area, which forms the focus of our investigation, is in describing natural and/or ecological systems, e.g. interaction between bacteria and their environments; predator-prey relationships; flocking; genetic drift. We use the term "complexity" in its common usage to describe systems with many parts or interactions. Reasoning about these complex systems is challenging because, although we may be able to predict the next system state given some current state, it becomes increasingly difficult as we attempt to predict states further into the future. Our meaning of complexity may sometimes partially correlate with algorithmic complexity and “Big O” notation but we are not fundamentally concerned with the performance of algorithms here.

In this paper we consider emergent systems to be a subset of complex systems. Emergent systems are those that exhibit some "radical novelty" (Goldstein 1999, p. 50) or whose macroscopic behaviours are not predictably defined by the behaviours of their parts. In preliminary discussions with target users we found a tendency to assume that emergence (and to a lesser extent complexity) can only be exhibited by large-scale and/or non-deterministic systems. However, we will later illustrate that this may not be the case (see the "Vants" example below).

We do not enter a philosophical debate about determinism, choosing instead to consider determinism from an observational perspective – so if a normal observer could not be expected to have the level of information required to accurately predict all future states/events then a system is observationally non-deterministic. In terms of the notations and reasoning presented in this paper, we consider systems to be non-deterministic if the descriptions of them (formal or verbal) draw on probabilistic behaviour.

Representations and Reasoning

For the purpose of this investigation, we view our representational framework from two different perspectives; as a notation suitable for specifying computer models and also as an "external representation" to support reasoning. Discussion about internal and external representations has described internal representations as the "knowledge and structure in individuals' minds" and external representations as those in the external environment (Zhang & Patel 2006, p. 334). In our case this external representation is the notation used to describe systems, i.e. the program code (though we form a less restricted view as we develop the ideas in this paper). Traditional approaches to cognition consider external representations as "peripheral aids" typically operating as notations used only to aid memory (Wang, Johnson, Sun & Zhang 2005; Zhang & Wang 2009) and in computing, code is regarded as a product – the result of some problem solving activity – but this presents too narrow a view of the potential value of external representations.

Cox (1999) and Lehrer (2006) argue that external representations, models in particular, enable students to map the natural world to the representational world. It is this externalisation, where the demands of reasoning are distributed across the external representations (forming a distributed cognition system) which enables students to work with their ideas and to engage in reasoning (Zhang & Norman 1994; Zhang 1997a, 1997b; Zhang 1998; Xu, Tytler, Clarke & Rodriguez 2012) – the external and internal representations interact to enable reasoning and exploration. Prain and Tytler (2012) suggest that external representations can be considered as affordances; they provide individuals with opportunities to achieve certain reasoning or problem solving outcomes. Consider, for example, the mathematical system for writing numbers as an external representation and the meaning of numbers and symbols as an internal representation. Some tasks only become possible (long division is an example for most people) when both representations are used. Moreover, many mathematical proofs and formulas have only been discovered with the aid of such external representations. Thus representations are not only products of inquiry that reflect current understanding, but the use of representations is a process that promotes the development of new understanding (Carolan, Prain, Waldrip 2008; Tytler & Prain 2010). Representations afford reasoning.

Following other research (Lemke 2003, 2004; Waldrip, Prain & Carolan 2010; Prain & Tytler 2012), we use Peirce's (1998a; 1998b) novel conceptualisation of representations and reasoning to explore software models as external representations that afford reasoning. Peirce proposes a specific relationship between representations and reasoning, encapsulated in his notion of *logic as semiotic*. Peirce (1998b) argues that "logic is the art of reasoning" (p. 11) and that reasoning is the process "to find out, from the consideration of what we already know, something else which we do not know" (1992a, p. 111). Peirce (1992c, 260) describes semiotic as "the science of the general law of signs" with a sign consisting of the triad of the *object* (the thing being represented), the *representamen* (the thing doing the representing), and the *interpretant* (the effect of the relationship between the object and the representamen on those involved in the communicative act) (1998a). The maxim "logic as semiotic" thus defines reasoning as: "the process by which representations operate to make meaning" and reasoning becomes a representational process; one cannot reason without using representations.

Peirce (1992a) delineates a number of different ways by which inquiry can be conducted, identifying the scientific method (with its direct reference to an external reality) as the approach which supports the most effective types of reasoning. Peirce argues that reasoning can be in the form of deduction, induction or abduction (1992b) and considers deduction to be the least productive as it is only analytical in nature, while induction and abduction can synthesize new inference and are thus more productive. Abduction (the process through which hypotheses about reality are generated and then tested through scientific means) is most highly valued, for while induction tends to be classificatory in nature, abduction is explanatory resulting in the production of truly new understanding for the individual. For this reason, representations that afford abduction are the most valuable in developing understandings of complexity.

For our work we are interested in finding an existing representation that will enhance users' mental models (internal representations) and reasoning processes as much as possible. So it is important to find a representation that aids exploration and understanding of the phenomena we wish to study but

which also provides a basis for developing computer models. That is: we want some written, external representation to complement users' internal representations and form a partially distributed framework for cognition, producing an external cognitive artefact which facilitates exploration and discovery (i.e. abductive reasoning) while also offering the ability to be executed/interpreted by some kind of computational system. Consequently our representation will behave in a similar way to mathematical/logical representations but will also have the added property that it can be executed, its results observed and that this observation will further enhance its utility as a reasoning system.

In order to evaluate suitable representations we examined the nature of user-constructed representations, since we accept the findings of Prain and Tytler (2012) which indicate "strong conceptual gains and a high level of ... engagement" (p. 2752) for students using self-constructed representations. It is through constructing their own representations, and in so doing making abstractions about scientific observations, as well as engaging with the canonical representations of science that students undertake meaningful learning. In our case we cannot simply adopt any user constructed representation, no matter how semantically valid, because we require a formal representation which can be executed to investigate system properties and to produce observable results. Nonetheless we support a representation-construction pedagogy for teaching and learning science. This is a form of guided inquiry in which tutors support representational development by presenting challenges to students, while also constraining the production of representations to that which is both semantically unambiguous and in fine enough detail to describe the relevant features of a system (Tytler, Haslam, Prain & Hubber 2009; Hubber, Tytler & Haslam 2010). The tutor then channels users towards constructing representations that appropriately model specific phenomena and which begin to approach the efficacy of canonical representations. It is also important for tutors to guide students in the discussion of the adequacy of both user-constructed representations and existing representations and to promote discussion about which representations are suitable for which purposes (diSessa & Sherin 2000; diSessa 2004; Ainsworth, Prain & Tytler 2011). The choice of representation is important; different representations make different features explicit and facilitate different reasoning. Consequently, students are also encouraged to switch between representations and in doing so re-represent their understandings to experience the value of different representations and to further develop their knowledge (Ainsworth 1999; Prain & Waldrip 2006; Prain, Tytler & Peterson 2009).

We adopted a similar approach to the development of tutor-led, user-constructed representations while also requiring our tutors to steer students towards existing representations if this was possible within the context of their problem solving. While we recognise that there is some potential conflict of interest here (the tutor aims to facilitate free construction of representations while also hoping to steer users towards some exiting system) the process is highly informative to the final choice of representation. We also notice that users have a feeling of *ownership* in the choice of representation by participating in this process. And additionally, as argued by Prain and Tytler (2012), when students engage in the process of building their representations they carry out "authentic scientific knowledge-building" (p.2753) and discover new phenomena of the systems they intend to model. In this way their process of engagement has genuine benefit in addition to any product they may develop.

NetLogo as a Candidate Representation

We evaluated various notations and programming languages as candidate representations, these included Java and Netlogo as well as mathematical and logic based notations. NetLogo was included in the study because of the volume of research reporting its successful use in conceptualising complex systems, natural systems in particular (Jacobson & Wilensky 2006; Wilensky & Reisman 2006; Wilenksy & Novak 2010; Levy & Wilensky 2011; Dickes & Sengupta 2013). Our findings concur with this research but we approach the study from a new perspective, that of external representations.

In this paper we concentrate more on the evaluation of the suitability of NetLogo as a representation than a thorough assessment of the relative merits of different representations. We nevertheless make comparisons where appropriate. We assume readers are familiar with the features of Java and mathematical notations, but provide a brief outline of NetLogo since this is less widely used.

NetLogo is an agent-based modelling and programming environment based on M.I.T.'s StarLogo system. Designed for a wide range of audiences, it aims to present a low learning threshold for new users but a high ceiling for advanced experimentation. Specifically it aims to be accessible for novice programmers and users who are not necessarily from a scientific background. NetLogo programs are called "models."

NetLogo provides a programming language where concepts are expressed in terms of agents and their environments (2-dimensional worlds of tiles/patches) and presents an animated graphics panel, the *world*, which shows the actions of agents and their states at run-time. It provides tools for producing control panels (buttons, scrollbars, etc.) and graphs (see Fig 1). Typically, some controls allow the operation of models to be manipulated at run-time thereby modifying the behaviour of running systems and providing an additional level of experimentation. Standard controls allow models to be paused or to have their running speed slowed or increased.

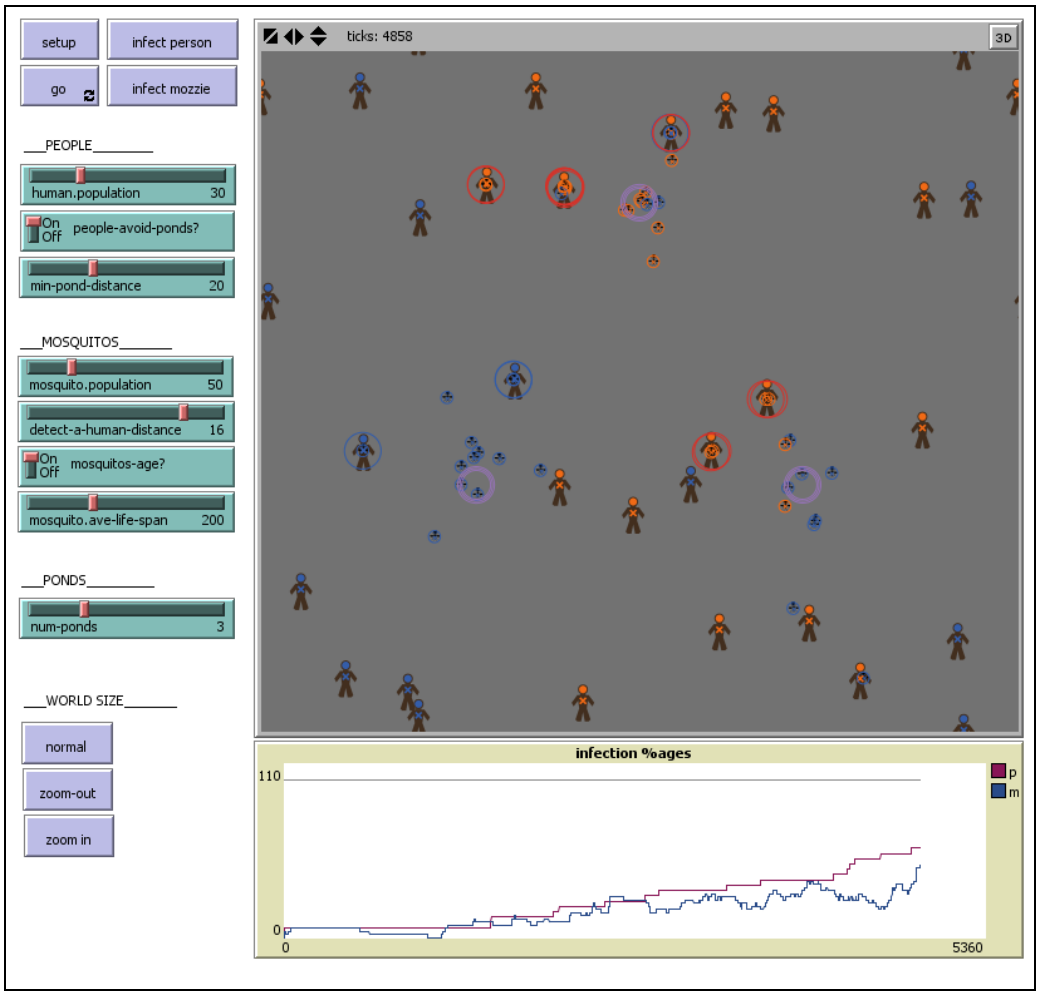


Fig. 1. A NetLogo model showing malaria transmission between humans and mosquitoes.

The figure below (see Fig 2) shows example code for a predator / prey model containing "foxes" and "rabbits". The code shown defines the set-up and activity of rabbits. Code for the foxes is similar. In this model both species start with a low preference for activity (a random percentage between 0% and 10%). The model runs multiple decision-action cycles for foxes and rabbits. When a fox "lands on" a rabbit, the rabbit dies and a new rabbit is cloned. The clone inherits its preference for activity from its parent but randomly adjusts this by +/- 5%. A similar approach is taken with foxes – those who "eat" the fewest rabbits are replaced by the clones of other foxes and these clones also randomly adjust their activity levels when they are created. The model shows how the populations become increasingly active over time as rabbit and fox populations “co-evolve” larger values for their "active%" variables.

```
breed [rabbits rabbit]      ;; rabbits are one type of agent...
rabbits-own [active%]       ;; ...they have a probability of being active

to setup-rabbits            ;; this procedure creates a population of rabbits
  create-rabbits 70                   ;; create 70 rabbits
  ask rabbits
  [ set shape "rabbit"                ;; set their appearance
    set color white
    setxy random-xcor random-ycor     ;; set random x,y coordinates...
    set active% (random 10)           ;; and a 0%-10% activity probability
  ]
end

to move-rabbits                       ;; this procedure describes the behaviour
  ask rabbits                         ;; of rabbits for each cycle.
  [ if (trigger-probability active%)  ;; if this rabbit is active
    [ face nearest-of foxes           ;; find the nearest fox
      right 180                       ;; turn around 180 deg
      wiggle                          ;; randomly turn a bit left or right
      forward 0.5                     ;; move forward 1/2 step
    ]
  ]
end

to clone-a-rabbit
  ask one-of rabbits      ;; ask a rabbit to clone itself...
  [ hatch 1               ;; ...then mutate the activity of the clone
    [ set active% (randomly-adjust active%) ]
  ]
end
```

Fig 2. Sample code for a co-evolving predator / prey model.

Evaluating NetLogo as an External Representation – Users' Experiences

We evaluate the use of NetLogo as an external representation in two ways. First we examine examples of NetLogo models, how they are specified, how they illustrate complexity and the response of user groups. Secondly (see later section "addressing key criteria") we consider how NetLogo meets the properties of external representations summarised by Zhang & Patel (2006).

Example 1 – Vants

The first example we use here serves to illustrate how complexity and emergence can exist in, and arise from, systems that are deterministic and simple to specify. This example also highlights the use of the NetLogo world as an external representation, which both informs and supports reasoning. The example is a simplified version of Langton's virtual ants (Langton 1986) and based on a similar NetLogo model from Wilensky (2005).

The virtual ant world consists of a 2-dimensional grid of tiles (patches), these (logically) have a black side and a white side and are initialised to be white-side up (they have their colour set to white). In our simplified version there is only one ant. This ant repeatedly flips the tile it is standing on, moves forward one tile, then turns right or left depending on the colour of the new tile it is on. In NetLogo the behaviour of this ant is specified below.

```
ask ants
[ flip-tile
  forward 1
  ifelse (pcolor = white)
  [ right 90 ]
  [ left 90  ]
]
```

The ant starts in a world of white tiles but after 5 moves starts to encounter tiles that it has flipped to black. As the model progresses, the ant wanders chaotically, flipping and re-flipping tiles. After 10,000 cycles the ant has created a figure of tiles with no visibly discernible pattern (see Fig. 3). The virtual ant model is deterministic and its complexity is unnoticed by many users who assume complex systems require multiple agents (or causative entities) and that these agents must interact. In fact the ant world behaves like a 2-dimensional Turing Machine with the colours of the tiles defining a set of binary-encoded instructions. In a generalised model, or models containing multiple ants, various patterns of self-organised, emergent behaviour may occur. Even with our simple model we can observe organised structures emerging after 10,500 cycles (see Fig. 4) and witness other larger scale behaviour patterns (our ant can sometimes be seen "running" up prebuilt columns or "undoing" – reversing patterns of tiles created earlier).

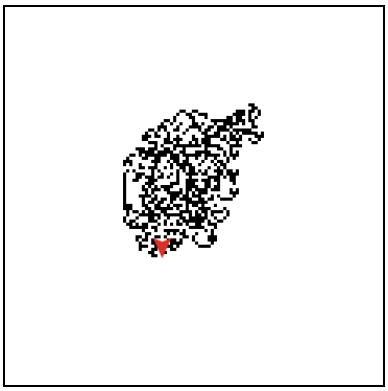


Fig. 3. Vant at 10,000 cycles.

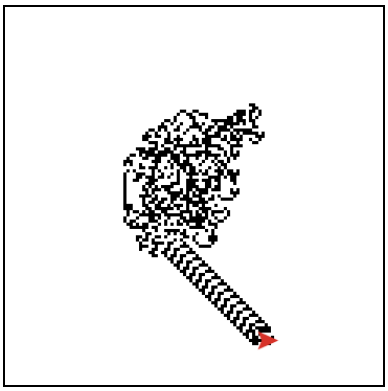


Fig. 4. Vant at 11,500 cycles.

From the perspective of our examination of NetLogo as an external representation there are a number of points of interest:

- the first evidence of emergence, self-organisation and, by implication, complexity arises after 10,000 cycles and is only evident by visual observation, with the NetLogo world providing a conceptualisation of a top-down view of a physical/geographical environment. The evidence of emergence is explicit and self-evident, no analysis is required and the structures are clearly perceived;
- before examining the system using NetLogo as a reasoning tool we found no students (including those who are experienced programmers or mathematicians) to correctly predict the emergent properties of this system from its rules;
- the NetLogo code describing activity of the ant(s) is understood by the youngest and least experienced of our user group who are able (i) to act out the ant behaviour (and other behaviours coded in a similar style) and (ii) to understand (and then explain) the cause of the emergent behaviour when it is investigated using the NetLogo agent inspector;
- before exposure to the NetLogo Vants model, a group of programming students were given the system's rules and asked to code the ant behaviour using a programming language of their choice and report on their findings. Of the 12 respondents, none identified any complex or emergent behaviour, primarily because they either (i) terminated the testing too soon, (ii) failed to visualise behaviour at all or (iii) failed to visualise behaviour from the perspective of the world.

Example 2 – Genetic Drift

Genetic drift describes the process where, over time, random effects cause changes in the frequency of gene variations within a population. In small populations genetic drift can cause some gene types to die out completely, though in the absence of other evolutionary pressures gene frequencies will tend to remain approximately constant in larger populations. Genetic drift is considered to be one of the

processes which can influence speciation. It can be modelled mathematically (see later comments) or by using various types of algorithm (Hartl & Clark 2007; Wilensky 1997; Tian 2008). In our implementation (developed independently of Wilensky (1997) but resulting in a similar specification) we model genetic variations using colour on a population of static individuals. At each cycle, one individual has its genetic type replaced by the type of one of its neighbours – this is achieved simply by setting its colour to that of one of its neighbours.

The NetLogo code is specified as follows (note that in this case the agents are named "bugs"):

```
to go
  ask one-of bugs
  [ set color [color] of one-of neighbors ]
end
```

Modelling genetic drift in this way is of interest for various reasons: (i) it operates like a type of cellular automata; (ii) the model can be extended to investigate types of speciation; (iii) of relevance here, the model involves a high level of simple agent-agent interaction which causes it to exhibit emergent properties. See Fig. 5 and Fig. 6 for initial and later states of low population models and Fig. 7 for a later state of a large population model where the visual effect is similar to that of cloud formation.

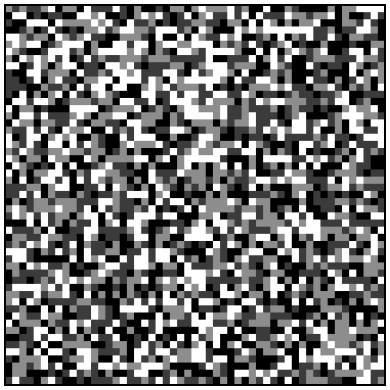


Fig. 5. Drift – initial

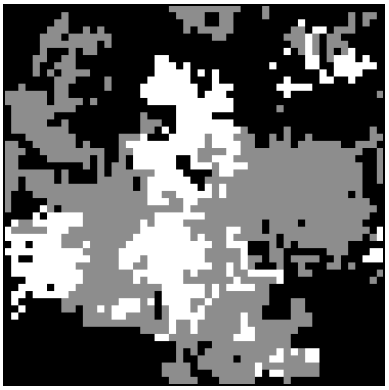


Fig. 6. Drift – later

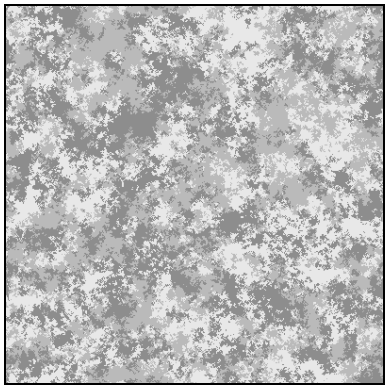


Fig. 7. Drift – large scale

Note that an important attribute for external representations is that they facilitate experimentation (i.e. abduction). As an example of this consider adapting the model to extend the neighbourhood for selection. Two steps are required: (i) adding a slider/scrollbar to the interface controlling a numeric variable *neighbourhood*; and (ii) making a small change to the code used earlier, as shown below.

```
to go
  ask one-of bugs
  [ set color [color] of one-of bugs in-radius neighbourhood ]
end
```

The genetic drift model appears deceptively simple in NetLogo suggesting that genetic drift may be fundamentally simple, however this is not supported by our investigation. One of our advanced user groups (final year computing undergraduates) was shown two alternative mathematical equations (Fig. 8) describing the probabilistic behaviour of a simple genetic drift process (in which selection occurs globally rather than with near neighbours). They were unable to recognise the equations as relating to genetic drift or reason with them to predict systems behaviour.

(i) $$Y_n = \prod_{i=1}^{n} \frac{g(X_i)}{f(X_i)}$$ (ii) $$\frac{(2N)!}{k!(2N-k)!} p^k q^{2N-k}$$

Fig. 8. Two probabilistic equations describing aspects of genetic drift.

Our advanced group were split into 8 development teams (of 3-4 students in each) and tasked to produce their own models of genetic drift using Java, C# or C++. Some teams failed to produce working models or incorrectly specified their models; in one model, for example, a “bug” would reset itself to the most commonly occurring type/colour rather than to a random choice of its neighbouring types, causing a significant change in the emergent behaviour. Of those teams who had correctly specified their models some still failed to observe aspects of behaviour including: (i) localised convergence (clustering) of types; and (ii) extinction of types with smaller populations. Only 3 of the 8 teams correctly observed behaviour, this contrasts with the level of success of teams using NetLogo (who all made relevant observations).

Evaluating NetLogo as an External Representation – Addressing Key Criteria

An external representation acts as no more than a memory aid, having limited utility, if it simply mirrors an internal representation. In order to form a distributed cognitive environment, external representations need to augment and expand the capacity of internal representations (Ainsworth 1999, 2006; Cox 1999). It is important that there is a conceptual correspondence between the internal and external representations otherwise the burden of translating between them outweighs any benefit offered.

Our observations show that NetLogo functions well as an external representation with users in our problem domain; its different facets (code, world and UI controls) allow it to interact with users' internal representations in a variety of ways (see Fig. 9).

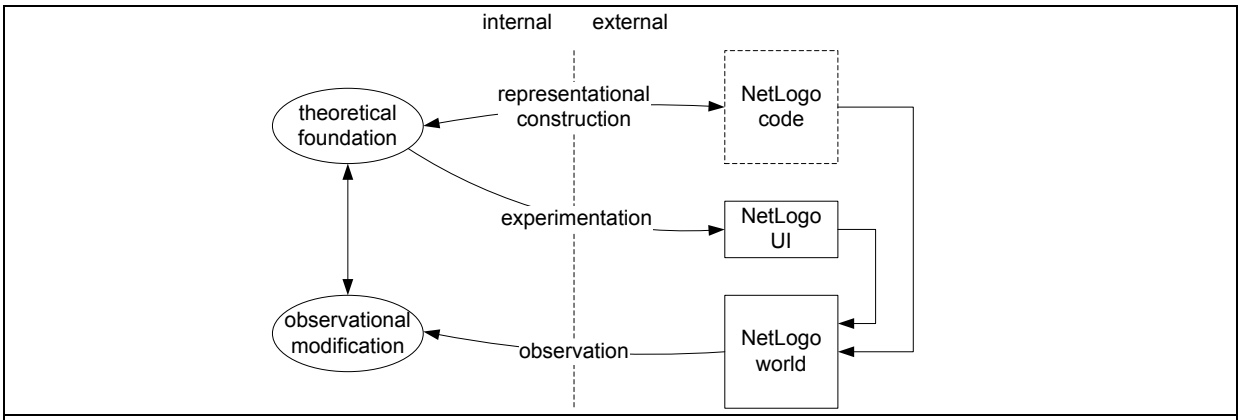


Fig. 9. Interaction between internal representation and NetLogo.

Zhang & Patel (2006) summarise properties of external representations, specifically that they provide the following (list edited from Zhang & Patel):

1 memory aids (to reduce memory load);

2 directly perceived information;

3 knowledge/skills unavailable internally;

4 support for easy recognition & direct inference;

5 effortless support for cognitive behaviour;

6 generation of more efficient action sequences;

7 facilities to stop time, supporting perceptual rehearsal with visible & sustainable information;

8 reduced need for abstractions;

9 aids to decision making (accuracy & effort).

We observe that the process of producing NetLogo code is itself constructive for reasoning and the development of understanding. Though more investigation is required in this area of study, initial

findings suggest that this effect is more prevalent with NetLogo code than with other programming languages. We propose possible reasons for this:

- tutors emphasise the importance of the *process* of code production rather than the model as an artefact. The primary criterion for success is identified as understanding a system; the production of a robust, fully functioning model is a secondary goal. Clearly this approach can also be taken with other languages, but the mind-set of our students tends to consider the use of other languages as an exercise in program specification not learning;
- NetLogo code describes activity in terms of the actions of individual agents, this produces a subtle shift in the conceptualisation of code elements from "objects having things done to them" to "agents interacting in their world". This influences the way students discuss their code, causing them to personalise (or anthropomorphise) their agents' behaviour and to more easily rationalise the macroscopic behaviours of their systems as resulting from the collective actions of a population of individuals;
- for the types of population-based multi-agent systems we have used in our study, the level of abstraction provided by NetLogo code matches the style of students' verbal description of agent activity. Its primitives and constructs lend themselves to these types of system, limiting the need for students to create additional abstractions. This is in contrast to development in languages like Java, C#, etc. where programmers need to specify the lower-level mechanics of interactions between agents and the worlds they inhabit;
- NetLogo code production is closely coupled with model experimentation, so phases of code production are shorter and more informed by model observation (the details of our data collection and analysis are beyond the scope of this paper).

The animated NetLogo world (the graphics panel) showing the movement, birth and death of agents, with facilities for pausing behaviour and slow-motion, addresses most of the representational facets identified by Zhang and Patel (2006) and listed earlier. In contrast to program code and other notation-based representations, the changing system state is directly perceivable in NetLogo and may be cross-referenced with graphs included in models. Agents of different types/breeds are easily recognised as they are typically represented using icons of different shapes and colours. Additionally agent-agent interaction is made observable and explicit with the use of other graphical tools.

NetLogo control panels allow running models to be changed dynamically, altering models as they run. This enhances experimentation by allowing users to immediately visualise the consequences of modifying the attributes and behaviours of agents – the agents' new behaviour will be immediately observed. In this way NetLogo clearly affords opportunities for abduction. These controls work in conjunction with other (default) controls that allow agents to be paused, their worlds to be increased/decreased in size, etc. and an agent "inspector" which provides facilities to examine the internal states of agents and "follow" their individual movements in the graphics world.

By following the discussions of student groups, using an ethnographic approach, we observe trends in the behaviour of students developing NetLogo models:

1 students engage in phases of observation and experimentation, which directly inform code modification and structure plans for further experimentation. We notice that experimentation tends to focus more on system behaviour than testing code (in contrast to the use of other languages) and a greater tendency to push experiments to the point at which behaviour "breaks" revealing the thresholds of system dynamics;

2 a continued tendency to anthropomorphise agents when discussing activity seen in the graphics world;

3 an ability to discuss modelling abstractions with increased clarity and understanding. In assessing this we used various models, which used either one-to-one or one-to-many relationships between NetLogo agents and their natural analogs.

The part of our study discussed above involved study teams of student programmers. These students were at intermediate or advanced levels of competence with Java, C# or C++ and had some additional competence with other languages. They were given a one-hour overview of NetLogo, which included the use of models and building control panels as well as NetLogo code. They were then asked to experiment with some pre-constructed models and were later asked to build software to investigate other systems. Most of our findings here have concentrated on this, but it is also interesting to note that subsequently:

- 16 groups investigated swarm dynamics (including bacterial activity, altruistic and parasitic behaviour, disease pandemics, etc.), all 16 groups chose to use NetLogo, self-learning the platform to the necessary level;
- 14 of 16 groups chose NetLogo to model evolution;
- 6 of 8 groups used NetLogo as an exploratory tool for designing agent deliberation for a collaborative problem solving system (the others described the system at a more abstract level using BDI style plans);
- the quality of student models as well as their investigations compared favourably with work they undertook in other languages (based on results for assessed work).

Conclusions

NetLogo has been successfully used by various researchers from education and science to model complex systems and teach programming. For our work, the models themselves are less important than the investigation into complex systems that modelling may afford. With this approach we are less interested in programmed computer models as artefacts and more interested in the processes that are used to formulate them. Consequently, while other work has examined NetLogo as a modelling tool or as a system to teach programming to novices, we have evaluated NetLogo from an alternative perspective: that of an external representation that affords reasoning.

Specifically, we have examined how well NetLogo serves to support the perception and analysis of complex systems, particularly those exhibiting some emergent property, and how it functions to provide opportunities for reasoning about complex systems. We have achieved this by following groups of students when they experimented with pre-constructed NetLogo models and also when they built their own models. We have also considered how NetLogo addresses the features of external representations identified by Zhang and Patel (2006) among others.

Our findings strongly support the use of NetLogo as a reasoning system to improve understanding of complexity with both novice and advanced users. We have observed that the different facets of NetLogo (code, control panel and graphics environment) interact with users' internal representations, serving to inform their experimentation and complement their understanding. In this way Netlogo operates as an external representation which facilitates cycles of hypothesis and test, thereby promoting abductive enquiry and exploration. We find that this compares favourably to using other languages and notations that typically offer different types of language semantics and more restricted representational forms. In addition, we find that NetLogo fulfils the criteria for external representations as they are defined in related work, supporting users' reasoning with tools to animate system behaviours, which makes the interaction of system entities explicit and the emergent properties of systems visually observable.

References

Ainsworth, S. (1999). The functions of multiple representations. *Computers & Education, 33* (2-3), 131-152.

Ainsworth, S. (2006). DeFT: A conceptual framework for considering learning with multiple representations. *Learning and Instruction, 16* (3), 183-198.

Ainsworth, S., Prain, V., & Tytler, V. (2011). Drawing to learn in science. *Science, 333* (6046), 1096-1097.

Carolan, J., Prain, V., & Waldrip, B. (2008). Using representations for teaching and learning science. *Teaching Science: The Journal of the Australian Science Teachers Association, 54* (1), 18-23.

Cox, R. (1999). Representation construction, externalised cognition and individual differences. *Learning & Instruction, 9* (4), 343-363.

Dickes, A., & Sengupta, P. (2013). Learning natural selection in 4th grade with multi-agent-based computational models. *Research in Science Education, 43* (3), 921-953.

diSessa, A. A., & Sherin, B. L. (2000). Meta-representation: An introduction. *Journal of Mathematical Behavior, 19* (4), 385-398.

diSessa, A. A. (2004). Metarepresentation: Native competence and targets for instruction. *Cognition and Instruction, 22* (3), 293-331.

Goldstein, J. (1999), Emergence as a construct: History and issues. *Emergence: Complexity and Organization, 1* (1): 49–72.

Hartl, D., & Clark, A. (2007). Principles of Population Genetics (4th ed.). Sinauer Associates. p. 112.

Hubber, P., Tytler, R., & Haslam, F. (2010). Teaching & learning about force with a representational focus: Pedagogy and teacher change. *Research in Science Education, 40* (1), 5-28.

Jacobson, M. J., & Wilensky, U. (2006). Complex systems in education: Scientific and educational importance & implications for learning sciences. *Journal of the Learning Sciences, 15* (1), 11-34.

Langton, C. G. (1986). Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena, 22* (1-3): 120–149.

Lehrer, R., & Schauble, L. (2006). Cultivating model-based reasoning in science education. In R. K. Sawyer (Ed.), *The Cambridge handbook of the learning sciences* (pp. 371-387).

Lemke, J. (2003). Mathematics in the middle: Measure, picture, gesture, sign, and word. In M. Anderson, A. Saenz-Ludlow, & V. V. Cifarelli (Eds.), *Educational perspectives on mathematics as semiosis: From thinking to interpreting to knowing* (pp. 215-234). Ottawa: Legas Publishing

Lemke. (2004). The literacies of science. In E. W. Saul (Ed.), *Crossing borders in literacy and science instruction: Perspectives on theory and practice* (pp. 33-47). Newark: International Reading Association/National Science Teachers Association

Levy, S. T., & Wilensky, U. (2011). Mining students' inquiry actions for understanding of complex systems. *Computers & Education, 56* (3), 556-573.

Peirce, C. S. (1998a). What is a sign? In N. Houser, A. De Tienne, J. R. Eller, C. L. Clarke, C. Lewis, & D. B. Davis (Eds.), *The essential Peirce – Selected philosophical writings – Volume 2 (1893 – 1913)* (Vol. 2, pp. 4-10). Bloomington: Indiana University Press.

Peirce, C. S. (1998b). Of reasoning in general. In N. Houser, A. De Tienne, J. R. Eller, C. L. Clarke, C. Lewis, & D. B. Davis (Eds.), *The essential Peirce – Selected philosophical writings – Volume 2 (1893 – 1913)* (Vol. 2, pp. 11-26). Bloomington: Indiana University Press.

Peirce, C. S. (1992a). The fixation of belief. In N. Houser, & C. Kloesel (Eds.), *The essential Peirce – Selected philosophical writings – Volume 1 (1867 – 1893)* (Vol. 1, pp. 109-123). Bloomington: Indiana University Press.

Peirce, C. S. (1992b). Deduction, induction and hypothesis. In N. Houser, & C. Kloesel (Eds.), *The essential Peirce – Selected philosophical writings – Volume 1 (1867 – 1893)* (Vol. 1, pp. 186-199). Bloomington: Indiana University Press.

Peirce, C. S. (1992c). An outline classification of the sciences. In N. Houser, A. De Tienne, J. R. Eller, C. L. Clarke, C. Lewis, & D. B. Davis (Eds.), *The essential Peirce – Selected philosophical writings – Volume 2 (1893 – 1913)* (Vol. 2, pp. 258-262). Bloomington: Indiana University Press.

Prain, V., Tytler, R., & Peterson, S. (2009). Multiple representations in learning about evaporation. *International Journal of Science Education, 31* (6), 787-808.

Prain, V., & Tytler, R. (2012). Learning through constructing representations in science: A framework of representational construction affordances. *International Journal of Science Education, 34* (17), 2751-2773.

Prain, V., & Waldrip, B. (2006). An exploratory study of teachers' and students' use of multi-modal representations in primary science. *International Journal of Science Education, 28* (15), 1843-1866.

Tian, J. P. (2008). Evolution algebras and their applications. Lecture Notes in Mathematics 1921. Berlin: Springer-Verlag. p. 11.

Tytler, R., Haslam, F., Prain, V., & Hubber, P. (2009). An explicit representational focus for teaching and learning about animals in the environment. *Teaching Science: The Journal of the Australian Science Teachers Association, 55* (4), 21-27.

Tytler, R., & Prain, V. (2010). A framework for re- thinking learning in science from recent cognitive science perspectives. *International Journal of Science Education, 32* (15), 2055-2078.

Waldrip, B., Prain, V., & Carolan, J. (2010). Using multi-modal representations to improve learning in junior secondary science. *Research in Science Education, 40* (1), 65-80.

Wang, H., Johnson, T. R., Sun, Y., & Zhang, J. (2005). Object location memory: The interplay of multiple representations. *Memory & Cognition, 33* (7), 1147-1159.

Wilensky, U. (1997). Models of genetic drift. http://ccl.northwestern.edu/netlogo/models/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wilensky, U. (2005). NetLogo Vants model. http://ccl.northwestern.edu/netlogo/models/Vants. Center for Connected Learning, Northwestern University, Evanston, IL.

Wilensky, U., & Reisman, K. (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories—An embodied modeling approach. *Cognition and Instruction, 24* (2), 171-209.

Wilensky, U., & Novak, M. (2010). Teaching and learning evolution as an emergent process: The BEAGLE project. In R. S. Taylor, & M. Ferrari (Eds.), *Epistemology and science education: Understanding the evolution vs. intelligent design controversy* (pp. 213-243). New York: Routledge.

Xu, L., Tytler, R., Clarke, D., & Rodriguez, C. (2012). The value of multi-theoretic analyses: Representational & distributed cognition perspectives on classroom sequence about matter. Manuscript submitted for publication.

Zhang, J., & Norman, D. A. (1994). Representations in distributed cognitive tasks. *Cognitive Science, 18* (1), 87-122.

Zhang, J. (1997a). The nature of external representations in problem solving. *Cognitive Science, 21* (2), 179-217.

Zhang, J. (1997b). Distributed representation as a principle for the analysis of cockpit information displays. *International Journal of Aviation Psychology, 7* (2), 105-121.

Zhang, J. (1998). A distributed representation approach to group problem solving. *Journal of the American Society for Information Science, 49* (9), 801-809.

Zhang, J., & Patel, V. L. (2006). Distributed cognition, representation, and affordance. *Pragmatics & Cognition, 14* (2), 333-341.

Zhang, J., & Wang, H. (2009). An exploration of the relations between external representations and working memory. *PLoS One, 4* (8), 1-10.

Exploring Core Cognitive Skills of Computational Thinking

Ana Paula Ambrosio

Computer Science Institute
Federal University of Goias
apaula@inf.ufg.br

Leandro da Silva Almeida

Institute of Education
Universidade do Minho
leandro@ie.uminho.pt

Joaquim Macedo

Department of Informatics
Universidade do Minho
macedo@di.uminho.pt

Amanda Franco

Institute of Education
Universidade do Minho
Amanda.hr.franco@gmail.com

Keywords: POP-II.A. novices, POP-V.A. cognitive theories, POP-VI.F. exploratory

Abstract

Although still innovative and not largely disseminated, Computational Thinking is being considered as a critical skill for students in the 21st century. It involves many skills, but programming abilities seem to be a core aspect since they foster the development of a new way of thinking that is key to the solution of problems that require a combination of human mental power and computing power capacity. This paper presents an exploratory study developed to select psychological assessment tests that can be used to identify and measure Computational Thinking cognitive processes, associated to the programming component, so that strategies can be developed to promote it. After the literature review, we identified four central cognitive processes implied in programming, therefore important to Computational Thinking, and accordingly selected a set of four tests that were administered to a sample of 12 introductory programming students. Our results suggest that spatial reasoning and general intelligence are crucial dimensions for introductory programming, being also correlated to the students' academic success in this area. However, arithmetic reasoning and attention to detail tests did not correlate. Based on these results, directions for future research have been defined in order to effectively identify and develop the core cognitive processes of programming, ergo, to help develop Computational Thinking.

1. Introduction

Computing pervades everyday life (Bundy, 2007) and drives innovation necessary to sustain economic competitiveness in a globalized environment (White, 2010). Moreover, a growing number of researchers believe information processing occurs naturally in nature and that computation is needed to understand and, eventually, control such processing (Denning, 2009; Furber, 2012). In this sense, "Computational Thinking" is considered a critical skill for students in the 21st century, and one of the four primary skills, along with reading, writing and arithmetic (CSTB, 2010; Qualls & Sherrell, 2010).

Computational Thinking (CT) can be defined as the ability to interpret the world as algorithmically controlled conversions of inputs to outputs (Denning, 2009). Cuny, Snyder and Wing have defined this construct as the "(…) thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent" (n.d. as cited in Wing, 2011, p. 1). In other words, CT involves learning to think about, represent and solve problems that require a combination of human cognitive power and computing capacity (CSTB, 2010).

The main reason for the increasing interest in CT is the understanding that scientists will need to be computationally literate, and in the future, it will not be possible to do science without such literacy (Emmott, 2006). Computation is key to the solution of problems in all areas of expertise, and rather than playing merely a supporting role, computer science not only has introduced new, important ways to view and understand the world in which we live, but is transforming all other disciplines. This way, students that do not know how to reason computationally are highly undermined in their abilities as problem solvers (Emmott, 2006). Several skills and cognitive functions have been associated to CT, many of them related to problem solving. Moreover, this construct includes the ability to think logically, algorithmically and recursively, as well as creativity, ability to explain and ability to work in teams (CS4FN, n.d.). As to specific CT techniques employed by students, these include: problem decomposition, pattern recognition, pattern generalization to define abstractions or models, algorithm design, and data analysis and visualization, i.e., being able to collect, analyze and represent data in meaningful ways (Google, n.d.).

What is now being defined as CT has received, over time, other designations, such as “Algorithmic Reasoning”, “Algorithmic Thinking” or even “Process Thinking”, all closely related to Computer Programming. Thus, the development of CT has much in common with learning to program. During a workshop on “The Scope and Nature of Computational Thinking” (CSTB, 2010), several researchers highlighted the importance of programming within CT. However, teaching algorithms and computer programming presents challenges that persist to this day. Learning to program is a particularly difficult task, involving diverse knowledge and skills (Robins, Rountree & Rountree, 2003; Jenkins, 2002). Introductory programming, more specifically, is a highly complex activity, involving sub-tasks related to different knowledge domains and a variety of cognitive processes (Pea & Kurland, 1984). Here, a set of skills are valued: reading comprehension; critical thinking and systemic thinking; cognitive meta-components identification; planning and problem solving; creativity and intellectual curiosity; mathematical skills and conditional reasoning; procedural thinking and temporal reasoning; analytical and quantitative reasoning; as well as analogical, syllogistic and combinatorial reasoning.

Our exploratory study is part of a wider project aiming the conceptualization, assessment and promotion of CT. One of its primary purposes it to identify the skills associated to programming, ergo, central in CT – in fact, one of the assumptions of our study is that this construct and programming are closely related, so many of the cognitive processes used in one are also associated to the other. With this in mind, we selected a set of cognitive tests that could be used as assessment tools in signalling the cognitive processes associated to CT, and which we will call “computational thinking skills”. More specifically, we focused on psychological tests assessing different cognitive functions such as reasoning, numeric, spatial and attention skills. These four cognitive tests were applied to a sample of introductory programming students, in order to identify those skills that seem to be more relevant in programming. The rationale behind the choice of programming students to constitute the sample of our study is that novice programmers have to develop computational thinking skills to succeed, and this usually happens only when they begin their college education.

2. Computer programming and cognitive processes

One of the main cognitive variables considered in relation to programming refers to schemas or mental models (Cañas, Bajo, & Gonzalvo, 1994; Pennington, 1987; Soloway & Ehrlich, 1984; Wiedenbeck, LaBelle, & Kain, 2004). The importance attached to mental models seems to be associated to conceptual clarity. In this sense, it is believed that the existence of appropriate mental models is crucial in designing programs that are a direct representation of the first. In "Programming as theory building", Naur (1985) discusses the need for the programmer to have a "theory" about how the program clearly solves the problem in hands, as opposed to programming as a production process. This deep understanding of the problem and of the solution allows the programmer to not only implement the solution, but to be able to support what he is doing with explanations, justifications and answers to queries about it. The quality of the program is thus directly determined by the quality of the theory built by the programmer.

Mental models have been studied in relation to different aspects of programming and used in diverse ways to evaluate student's computing ability. Several authors analyzed the mental model of students regarding computer operation (Mayer, 1989; Perkins, Schwartz, & Simmons, 1988). For instance, Mayer (1989) claims that students without a fit mental model of how the computer processes variables and data in memory present greater difficulty in understanding programming language commands. Dehnadi (2006) suggests that students who succeed in programming courses are those that use a certain mental model, independently of its specificities, for the allocation and manipulation of variables in a consistent manner.

Authors have also been concerned with the differences between experts and novices regarding their mental models. According to Cañas et al. (1994), people have different mental representations of computer programs: these can be based on syntactic or semantic aspects, and the mental representations that are centered on semantic aspects are closer to experts' mental models. Ackermann and Stelovsky (1987) state that while novices perceive programs as sequences of commands, experts tend to group commands in schemes that represent given functionalities. To analyze the representation or mental model of a subject, tasks such as recall, categorization by similarity or association can be used, and seem to be sensitive to changes that occur in mental representation as students learn to program. Curiously enough, and contrarily to prior expectations, the students' mental model is not affected by previous programming experience; however, developing mental models leads to greater self-efficacy and better results (Wiedenbeck et al., 2004).

Despite several studies, or perhaps as a result of them, perception remains that programming is problem solving. Programmers learn to program by building programs, hence, by problem solving. Moreover, its main objective is to implement programs that solve computational problems. In fact, the resolution of problems is an important part of their work; however, in addition to restrictions that are normally found in problem solving, programming requires that these solutions are computationally executable. This imposes additional restrictions on the problems that must be implemented by general, effective and finite algorithms.

Once the solution to a problem has been found, the path can be reused to solve similar problems. Through multiple episodes of analog problem solving, individuals can induce abstract schematic representations of problems and their solutions, which can be retrieved and applied as solution plans when structurally similar problems are presented (Gick & Holyoak, 1980, 1983). So a schema can be defined as a cognitive structure that allows a problem solver to recognize a particular class of problems as belonging to a given state that normally requires particular moves or strategies. In other words, they create mental models.

Perhaps because of this relationship between programming and problem solving, evaluations of programming skills have been historically based on tests related to intelligence and deductive logic. In the early ages of computer programming, tests began to be developed in order to identify those who had "aptitude" for programming, a measure of the applicant's natural ability to understand and learn new things and perform new tasks. It was understood that the ability to program was innate and that not all could be programmers. For economic reasons, it was then necessary to identify those individuals who would succeed in learning programming. In the 60's, these aptitude tests were used by 68% of computer companies surveyed in the U.S.A. and 73% in Canada (Dickman & Lockwood, 1966 as cited in Robins, 2010). Many of these tests continue to be used by companies to select developers, but less intensely and as part of a broader process of selection.

The first of these tests to succeed was the *Programmer Aptitude Test* (PAT), developed by *IBM* in the 50's. The test consisted of three parts. First, the subject had to identify the next number in a series. Second, they had to identify analogies represented in figures, and in the third part had to do arithmetic problems. IBM currently uses the IPATO, an online test developed by IBM to test logical reasoning and ability to process information quickly.

A battery that obtained great success, still used today, is the *Aptitude Assessment Battery Programming* (AABP), developed in 1968 by Jack M. Wolfe . This battery is based on the traits and skills identified by Wolfe and simulates daily work tasks, assessing logic reasoning, ability to

interpret complex specifications, documentation and annotation skills, problem-solving skills, accuracy, attention to detail, speed, concentration and ability to follow instructions accurately.

The *Computer Programmer Aptitude Battery* (CPAB) is a battery of tests that determines individual aptitude for computer programmer or system analyst. Published by Vangent, Inc., is composed of five subtests that assess the subject's ability to understand the vocabulary used in mathematics, management and systems engineering literature; reasoning identified by the ability to translate ideas and operations in textual notation into mathematical notation; the ability to use abstract reasoning to find patterns in given letter series; the numerical ability translated by the ability to quickly estimate reasonable answers to calculations; and the ability to analyze problems and find solutions to a logical sequence using diagrams. There is a short and a long version. The short contains only those parts related to Reasoning and Diagramming that make up the long version, and which have been shown to be those that best predict the performance of programmers. Both tests are comparable and can be used for retest.

Another widely used battery is the *Berger battery*, a set of proficiency and aptitude tests marketed by Psychometrics, Inc., Henderson, NV (www.psy-test.com). Apart from tests on several programming languages, there is the B-APT for people with no programming experience which aims to identify candidates for training. It consists of 30 questions that must be answered in 1h15min and uses a hypothetical language that candidates must use to write small programs.

These tests are mainly used for recruiting programmers, and are more suited to experienced adult programmers, where we are interested in identifying core cognitive skills important to programming, especially to help those students that have difficulty in learning to program. Furthermore, the correlation between the results obtained in testing and the subjects' programming skills has been consistently low (Mayer & Stalnaker, 1968; Pea & Kurland, 1984). Experiments have shown little or no relationship between the final ranking of students in programming and their evaluated "aptitude" (Davy & Jenkins, 1999; Mazlack, 1980). In his article, Wolfe (1971) discusses the limitations of programming aptitude tests, including the use of multiple-choice questions, the undergraduate students' "tricks" in the testing and the inclusion of mathematical issues that reduce the test's effectiveness for commercial programming.

According to Weinberg (1998), being an area where there are basically no known factors that influence the behavior of programmers, much of the psychology of programming research attempts to identify what is important to be measured. So the big question is not "how to measure", but "what to measure". In this sense, and according to the same author, the use of aptitude tests is much more an analysis of tools than of subjects, so we first must define "what to measure". In this sense, and based on the review of literature, we have identified a set of cognitive abilities that seem to be correlated to programming success and that we considered in our study.

3. Materials and Methods

3.1 Participants

The study was undertaken with 12 freshmen students from the University of Minho in Portugal: five students from the BSc in Informatics Engineering (LEI) course, a 1st Cycle 3 year course that results from a transition to the Bologna Process of the old BSc in Informatics and Systems Engineering, and seven students from the BSc in Computer Science (LCC) course, a 1st Cycle 3 year course that results from a transition to the Bologna Process of the old BSc in Mathematics and Computer Science. Half of the participants were female and half male, with ages between 18 and 29 years old (M = 20.6; SD = 3.55). The participants were selected from two contrasting groups: students with good academic results and students with poor academic results. Both groups were selected by their teachers, based on their observations, since tests were applied in the beginning of the course and that grades were not yet available.

3.2 Instruments

To identify basic cognitive skills involved in CT, we used validated cognitive tests. Four tests were chosen based on perceived associations of the basic skills assessed by the tests and CT (or programming) found in the literature: a general intelligence test, a spatial reasoning test, a mathematical reasoning test, and an attention to details test. Very briefly, the intelligence test was chosen to verify the general idea that programming ability is innate and closely related to intelligence level, mainly from an abstraction and inductive reasoning point of view; the spatial reasoning test was proposed to verify the subjects' ability to "see" the problem and work with it mentally, reasoning with figures and patterns; the mathematical reasoning test aimed to evaluate the subjects' problem solving abilities and the role of mathematical knowledge in programming; finally, the attention to details test was used to evaluate the subjects' capacity to identify pertinent details and pattern recognition abilities.

Test D.48 (Pichot, 1949) is a nonverbal test that measures the g factor of subjects based on their abstraction capacity, ability to understand relations, reasoning by analogy and ability to solve new problems (induction). It is composed of 44 items that must be completed in 25 minutes. Each item contains a series of dominos to which a new one must be added taking into account the series' configuration. To discover the missing domino, the subject must apply reasoning, not only attending to the number of dots in the domino (discovering, for example, ascending values), but also to the configuration (discovering, for example, a particular symmetry) and analogy (discovering what B is to A as to discover what D is to C). This test has shown high correlation to academic achievement.

The *Spatial Reasoning test* assesses two components associated to spatial factor – the ability to recognize and view figures, and the ability to rotate or follow the movements of figures. It is a subtest of the *Differential Reasoning Tests* (*Bateria de Provas de Raciocínio, BPR-5*), composed of five subtests evaluating reasoning (common cognitive processes) with different content items (abstract, verbal, numerical, spatial and mechanical). The *Spatial Reasoning* subtest is composed of 20 items containing series of 3D cubes to be completed in 8 minutes. The series are obtained through the rotation of the cubes by movements that can be constant (for example, from left to right) or alternate (for example, left and up). By identifying the movements that took place, the subject must choose, amongst the alternatives, the cube that represents the next cube in the series, obtained by applying the identified movements (Almeida & Primi, 1996 as cited in Almeida, Antunes, Martins, & Primi, 1997).

The *General Aptitude Test Battery* (GATB) (U.S. Department of Labor, 1983), consists of 12 separately timed subtests which make up nine aptitude scores (general learning ability, verbal aptitude, numerical aptitude, spatial aptitude, form perception, clerical perception, motor coordination, finger dexterity and manual dexterity). Two subtests from the Portuguese version of GATB were used: the arithmetic reasoning subtest and the tool matching subtest. The *Arithmetic Reasoning subtest* assesses the ability to verbally solve expressed arithmetic problems and the ability to deal intelligently with numbers. It is a measure of general intelligence and numerical aptitude. The problems require basic mathematical skills like addition, subtraction, multiplication or division. The test includes operations with whole numbers, rational numbers, ratio and proportion, interest and percentage, and measurement. It contains 25 word problems with five alterative answers, to be answered within a seven minute time limit. Finally, the *Tool Matching subtest* assesses the ability to understand pertinent details in objects in pictorial or graphic material, as well as the ability to make visual comparisons or discrimination of detail. It consists of a series of exercises containing a stimulus drawing and four black-and-white drawings of simple shop tools. The four drawings have different parts of the tools painted in black and white. The subject must identify which of the four options exactly matches the stimulus drawing. There are 49 items, measuring form perception, with a time limit of five minutes.

3.3 Procedures

Participation was voluntary and in the students' free time. For this reason, the tests were administered in three different sessions to comply with the students' schedule. Preceding the application of the tests, we presented the goals of our study and assured the confidentiality of the individuals' results.

Students received one test at a time, and the instructions were read out loud previously. Since each test had a specific time limit, when it was up, the test was collected and another one given. The application of the four tests took approximately one hour. The order in which the tests were given to the students varied between sessions.

A raw score was obtained for each test, adding the number of correct items completed by the students. Thus, even though other formulas may be used to calculate global factors, as the aptitude scores in the GATB, these were not used in our analysis.

The quantitative analysis was undertaken. The results were correlated to the student's final exam result and final grade in their second semester programming course, and with a performance evaluation done by the teacher, that attributed a classification from 1 (min) to 5 (max) according to his perception of the student's aptitude and facility in learning programming, disregarding the student's effort and motivation. This evaluation was important to identify those students that showed good capacity for programming, but had poor academic results due to other factors, such as not doing homework assignments and missing the final test, as well as those students that were able to pass the class mainly due to their motivation and hard work, despite a blatant lack of skill.

4. Results and Discussion

In Table 1, we present the descriptive data (number of students that took the test, maximum possible score in the test, minimum and maximum scores obtained by the students, mean and standard deviation), regarding the cognitive tests undertaken by our sample as well as the students' academic achievement. Values have been standardized to maximum score 10 in order to facilitate reading and comparison of results. This exploratory study has shown interesting results: tests varied significantly, not only when comparing the different test scores for each student, but also when comparing the results our sample achieved in each test. This means that no student was consistently good in all tests, and in a given test there was a significant variation between the minimum and maximum number of correct answers. Furthermore, the students' academic results also varied: eight students were approved in the discipline and three failed, and for one the grades were not available; moreover, one of the students was approved after retaking the exam, obtaining a significantly better result (almost doubled).

Performance indicators	*N*	*Maximum possible score*	*Min.*	*Max.*	*M*	*SD*
D48	12	44	6.0	9.0	7.27	1.11
Spatial Reasoning	12	20	2.5	9.0	6.13	1.93
Arithmetic Reasoning	12	20	1.5	7.5	4.67	1.87
Matching Tools	12	49	7.0	9.0	7.55	.79
Final grade	11	20	4.5	8.5	6.18	1.38
Exam	11	20	2.0	7.8	4.40	1.93
Teacher evaluation	11	5	5.0	10.0	8.18	1.94

Table 1. Standardized students' results according to each cognitive test

In Table 2, the correlation coefficients between psychological tests and two academic achievement measures are presented. Analysing the correlation between the students' evaluation factors, we verified a very strong correlation between the grade obtained in the exam and the final grade (r = .898, $p < .001$). The teacher's evaluation correlated with the exam grade (r = .614, $p < .05$) but not with the final grade (r = .489, *n.s.*). This could be expected since the exam grade depends less on motivation and dedication than the final grade, which takes into account other factors such as home assignments and class work. Spearman's rank correlation coefficient was also tested, but did not yield better coefficient and significance values.

		D48	Spatial Reasoning	Arithmetic Reasoning	Matching Tools
FinalGrade	Pearson Correlation	.491	.579	.111	.310
	Sig. (2-tailed)	.125	.062	.746	.353
Exam	Pearson Correlation	.378	.427	-.069	.273
	Sig. (2-tailed)	.252	.191	.840	.416

Table 2. Correlation between performance indicators and results from cognitive tests (n=11)

As our main objective was to cross cognitive abilities (as measured by the four presented tests) and the students' performance in introductory programming courses, the correlation between the cognitive tests and the academic results was undertaken. Due to the small number of subjects, we will consider the correlation coefficient to verify the effect size since it gives us a perception of the strength of relationship between two variables (Field, 2009) without taking into account the significance – since the significance depends on the correlation coefficient and the degrees of freedom, and the latter is under the influence of the size of the sample (in our case small), which has an impact on the significance.

The correlations between psychological tests and academic classification in the domain of computer science point to two distinct patterns of correlation with regard to their effect size. On one hand, the D48 test (g factor) and the spatial reasoning test (rotation of figures in space and mental retention of these positions to continue the task) showed a large effect when correlated to the students' academic results, as opposed to the attention to detail test and the arithmetic reasoning test, which showed small and medium effects respectively. Thus, the students' computational skills, at the academic learning level, seem to require more from their logic-deductive reasoning skills (infer and apply or generalize relations) and a holistic or simultaneous organization of the information (spatial organization). Simple attention or calculus tasks do not seem to be relevant in differentiating performance in computer science students; this possibly means that, being very basic skills, with no inherent significant difficulty, all students attain a very similar level of performance, and therefore, they are not relevant to the differentiating goal of our study. It should be noted that the lack of correlation concerning the *Arithmetic Reasoning test* supports previous research suggesting that mathematical knowledge is not correlated to programming ability. However, the test was composed of word problems, which could imply the need for problem solving abilities, and may account for the medium size effect encountered. In this sense, further investigation may be needed to verify the correlation of problem solving abilities and computational thinking.

The data obtained in our study agrees with results obtained in other tests. The PISA 2003 test (OECD, 2003), in addition to items related to the domains of reading, mathematics and scientific literacy, included items related to problem-solving that evaluated processes very similar to those necessary in programming. According to its results, less than 20% of the students were able to solve difficult problems (level 3). In such problems, students should be able to analyze a situation and make decisions, handle multiple conditions simultaneously, think about relationships underlying a problem, solve it in a systematic way, evaluate their work and communicate results. About half of the students were able to answer level 2 questions, which imply good reasoning, being able to confront unfamiliar problems and being capable of coping with a certain degree of complexity. Data analysis showed a high correlation between problem solving and other areas, especially mathematics. However, the performance in solving mathematical problems that involve simple calculations, without further inferences, has a low correlation with performance in problem solving. These findings align with the evaluation of programming teachers that mathematical knowledge does not determine success in programming. They believe that the correlation between mathematics and programming is linked to the development of mental processes common to both areas, such as inference.

5. Conclusions

In our study, we aimed to identify the cognitive processes that are central in programming, therefore implied in CT, in order to understand which skills students must develop in order to succeed in learning introductory programming. By identifying which are these skills, and how they relate to academic success, it is viable to apply fitter pedagogic methodologies that will foster this essential type of thinking. This matter is most important, given the transversal pertinence of CT for every individual, and that the problems faced in teaching computer programming courses pass from the computer science realm to a global setting, justifying further study of the cognitive processes associated with Programming and CT, as to stimulate and promote it in all levels of education.

Our exploratory study emphasizes the complexity of CT as a mental process, for which is necessary to discriminate the underlying cognitive functions that comprise this construct. To do so, further investigation is needed, using a larger sample. In future research, our proposal regarding the cognitive processes that are core to CT needs refinement. First, to confirm the correlation between academic success in programming, and intelligence and spatial reasoning, other tests may be included, such as *Raven Progressive Matrices* and other spatial tests involving 3D figures from the *Differential Reasoning Tests*. It may also be important to include working memory tests, in light of recent research that identifies these executive processes as crucial for intellectual functioning. Second, to confirm the lack of correlation between programming and attention to details, a different perception test should replace the *Tool Matching* test, which may not have shown correlation due to the test format and lack of face validity. Similarly, the arithmetic reasoning test should be replaced by two distinct tests: one that measures problem solving abilities and another that measures math abilities, as to confirm the correlation between programming and problem solving that has been verified in related works, as well as the lack of correlation with pure mathematical abilities. Nevertheless, selected tests must present a higher level of difficulty as the results obtained seem to indicate low degree of student differentiation in the tests that were applied, making it impossible to estimate if these cognitive processes are really important to explain the academic result differences presented by the introductory programming students. Finally, future studies must combine qualitative and quantitative methodologies. For this, a larger number of subjects must be considered, always combining two levels of academic achievement in introductory programming, or, adopting a logic of expert versus novice, select students with different skill levels in this domain. Besides applying the tests, these two groups can be interviewed to interpret their performance in the tests. Content analysis of the answers may yield leads that approximate the learning that occurs and the different levels of academic performance of these students, with the inherent processes of the test items in which they had greater or lesser difficulty in solving. This might shed some light in understanding to what extent programming aptitude is “innate” and depending on intelligence, or developed, or even in analyzing the students’ schemas and mental models, and trying to comprehend how they develop.

6. References

Ackermann, D., & Stelovsky, J. (1987). The role of mental models in programming: From experiments to requirements for an interactive system. Lecture Notes in Computer Science, 282, 53-69.

Almeida, L. S., Antunes, A. M., Martins, T. B. O., & Primi, R. (1997). Bateria de Provas de Raciocínio (BPR-5): Apresentação e procedimentos na sua construção. Actas do I Congresso Luso-Espanhol de Psicologia da Educação (pp.295-298). Coimbra: Associação dos Psicólogos Portugueses.

Almeida, L. S., Candeias, A., Primi, R., Ramos, C., Gonçalves, A. P., Coelho, H., Dias, J., Miranda, L., & Oliveira, E. P. (2003). Bateria de Provas de Raciocínio (BPR5-6): Estudo nacional de validação e aferição. Revista Psicologia e Educação, 2 (1), 5-15.

Bundy, A. (2007). Computational thinking is pervasive. Journal of Scientific and Practical Computing, 1, 67-69.

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. International Journal of Human-Computer Studies, 40 (5), 795-811.

CS4FN, Computer Science for Fun, (n.d.). Retrieved February 16, 2012, from http://www.cs4fn.org/computationalthinking/

CSTB, Computer Science Telecommunications Board, (2010). Report of a workshop on the scope and nature of computational thinking. Washington, D.C.: The National Academies Press. Retrieved February 5, 2012, from http://www.nap.edu/openbook.php?record_id=12840&page=R1

Davy, J., & Jenkins, T. (1999). Research-led innovation in teaching and learning programming. Proceedings of ITiCSE '99 (pp.5-8). Cracow, Poland.

Dehnadi, S. (2006). Testing programming aptitude. In P. Romero, J. Good, E. A. Chaparro, & S. Bryant (Eds.), Proceedings of the 18th Workshop of the Psychology of Programming Interest Group (pp.22-37). University of Sussex.

Denning, P. (2009). The profession of IT- Beyond computational thinking. Communications of the ACM, 52 (6), 28-30.

Emmott, S. (2006). (Ed.). Towards 2020 Science. Microsoft Research. UK: Cambridge.

Field, A. (2009) Discovering Statistics using SPSS, 3rd ed. Sage Publications, London UK.

Furber, S. (2012) Shut down or restart? The way forward for computing in UK schools. London, UK: The Royal Society. Retrieved February 5, 2012, from http://royalsociety.org/education/policy/computing-in-schools/report/

Gick, M. L., & Holyoak, K. J. (1980). Analogical problem solving. Cognitive Psychology, 12, 306-355.

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. Cognitive Psychology, 15, 1-38.

Google. (n.d.). Exploring computational thinking. Retrieved February 5, 2012, from http://www.google.com/edu/computational-thinking/what-is-ct.html

Jenkins, T. (2002). On the difficulty of learning to program. In Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences (53-58). Loughborough: University United Kingdom.

Mayer, R. E. (1989). The psychology of how novices learn computer programming. In E. Soloway, & J. C. Spohrer (Eds.), Studying the novice programmer. Hillsdale, NJ: Lawrence Erlbaum Associates.

Mayer, D. & Stalnaker, A. (1968) Selection and evaluation of computer personnel - the research history of SIG/CPR, ACM '68 Proceedings of the 1968 23rd ACM national conference, New York, NY

Mazlack, L.J. (1980) Identifying Potential to Acquire Programming Skill. Comm. ACM, Vol. 23, pp 14-17.

Naur, P. (1985). Programming as theory building. Journal of Systems Architecture: The Euromicro Journal, 15 (5), 253-267.

OECD, Organization for Economic and Co-operation and Development. (2003). PISA 2003: First results from Pisa 2003 - Executive Summary. Retrieved February 5, 2012, from http://www.oecd.org/dataoecd/1/63/34002454.pdf.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive prerequisites of learning computer programming (Technical Report No.18). New York: Bank Street College of Education, Center for Children and Technology.

Pennington, N. (1987). Comprehension strategies in programming. In E. Soloway, & S. Iyengar (Eds.), Empirical studies of programmers: Second workshop (pp.100-113). Norwood, NJ: Ablex.

Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional strategies for the problems of novice programmers. In R. E. Mayer (Ed.), Teaching and learning computer programming (pp.153-178). Hillsdale, NJ: Lawrence Erlbaum Associates.

Pichot, P. (1949). Les test mentaux en Psychiatrie: Tome premier; instruments et methods. Paris: Presses Universitaries de France.

Qualls, J. A., & Sherrell, L. B. (2010). Why computational thinking should be integrated into the curriculum. Journal of Computer Science in Colleges, 25, 66-71.

Robins, A. (2010). Learning edge momentum: A new account of outcomes in CS. Computer Science Education, 20 (1), 37-71.

Robins A., Rountree J., Rountree N. (2003) Learning and Teaching Programming: A Review and Discussion. Computer Science Education, 13(2): 137-172

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programmer knowledge. IEEE Transactions of Software Engineering, SE-10, (5), 595-609.

U.S. Department of Labor. (1983). The dimensionality of the General Aptitude Test Battery (GATB) and the dominance of general factors over specific factors in the prediction of job performance for the U.S. Employment Service. (USES Test Research Report No. 44). Washington, D.C.: U.S. Government Printing Office.

Weinberg, G. M. (1998). The psychology of computer programming (Silver Anniversary Edition) (2nd ed.). New York: Dorset House Publishing.

White, J. (2010). The state of computer science education (Special Report). Retrieved February 5, 2012, from http://www.cioupdate.com/reports/article.php/3915826/Special-Report---The-State-of-Computer-Science-Education.htm

Wiedenbeck, S., LaBelle, D., & Kain, V. (2004). Factors affecting course outcomes in introductory programming. In E. Dunican, & T. R. Green (Eds.), Proceedings of the 16th Workshop of the Psychology of Programming Interest Group (pp.97-110). Carlow, Ireland: Institute of Technology Carlow.

Wing, J. (2011) Research notebook: Computational thinking - What and why? Retrieved February 5, 2012, from http://link.cs.cmu.edu/article.php?a=600

Wolfe, J. M. (1971). Perspectives on testing for programming aptitude. ACM '71 Proceedings of the 1971 26th annual conference (pp.268-277). Chicago, Illinois.

Wolfe, J. M. (1968). Aptitude Assessment Battery Programming. Walden Personnel Testing & Consulting Inc.

The Object Relational impedance mismatch from a cognitive point of view

Laura Benvenuti
Hogeschool van Amsterdam
Wibautstraat 2-4
1091 GM Amsterdam – The Netherlands
lbe@acm.org

Gerrit C. van der Veer
Sino European Usability Center,
Dalian Maritime University,
Dalian, 116026 China
gerrit@acm.org

Keywords: POP-I.A. group characteristics, POP-II.C. working practices, POP-V.A. mental models

Abstract

Computer Science has evolved towards a discipline with different branches. Scholars study and define artefacts from different viewpoints: computer languages, environments, paradigms. Practitioners work in multidisciplinary teams. They design, produce and link software that was designed according to different paradigms. We are interested in the communication between these practitioners. Do they refer to the same concepts when they use the same words? We designed an experiment to assess this.

1. Introduction

Computer Science has developed different branches, each with its own body of knowledge and its own problem area. Object Oriented software and relational databases have their origins in different approaches to the digitalization of information. There are evident differences between the Object Oriented (OO) and the Relational paradigm. The concepts underpinning the relational model are prescriptive and formal, while those underpinning object schemas are more descriptive (Ireland,2011).

What happens when the paradigms meet, when engineers link together software that was designed from different points of view? This area is characterized by a persistent problem, the Object Relational impedance mismatch. Many attempts have been made to solve the problem by developing new software (Object Relational Mappings), where tables correspond to classes and rows correspond to objects. This is problematic, though. In an OO program, an object has an identity independent of its state, but specific to the program execution. In the relational model, the row is identified by its attribute values (the state of the row) and it is accessible through set operations only: not directly. Direct access to objects during the execution of OO programs is possible and is based on navigation.

The problem with these solutions is their validation. Computer scientists, in particular database engineers, work on abstract entities. Technical solutions of the Object Relational impedance mismatch can only be evaluated if working practitioners perceive a common field of application, if they work with the same abstract entities.

In this paper, we focus on mental representations of abstract entities. In section 2 we explore the notion of a mental model. In section 3 we give an overview of research on cognitive aspects of programming and database interaction. Reflection follows in section 4. In section 5 we describe an experiment to assess differences in mental models between programmers and database professionals. The results until now are shown in section 6; preliminary conclusions are drawn in section 7.

2. Backgrounds

2.1. Mental models

According Craik, that the mind constructs “small-scale models” of reality and uses them to anticipate events (Johnson-Laird, 1989). Johnson-Laird formulated a theory of mental models, meant to explain human thinking and reasoning. Norman (1983) is more specifically concerned with mental models in Human-Computer Interaction. Users, states Norman, construct mental models of computer systems incrementally, while interacting with systems. The resulting models are constrained by the users’ prior knowledge, needs and context. These models are often incomplete. They are limited by the human

information processing system, by experience and by needs that can be contrasting: the need to focus and the need to retain important details. They are parsimonious in order to reduce mental complexity. User mental models are unscientific and unstable. They evolve over time: people learn, people forget. People use metaphors to simplify their models. Nevertheless, mental models are functional to support tasks such as planning, execution, assessment of results and understanding of unexpected events.

Despite the attention to mental models and their conception, there is little agreement on the exact definition of the term "mental model". Does the term refer to temporary structures in Working Memory (WM) or knowledge structures in Long Term Memory (LTM)? Cañas and Antolí (1998) introduce this definition: a mental model is "*the dynamic representation that is formed in WM combining the information stored in LTM and the extracted information from the environment*". From now on, we will follow Cañas' and Antolí's definition.

Human WM is limited. People take these limitations into account and adopt strategies to keep mental models manageable in WM. The question here is: can we assume mental models of practitioners with different backgrounds to be compatible with each other?

2.2. Assessing mental models: the teach-back protocol

Mental models can not be observed directly. Van der Veer (1990) extended an hermeneutic method designed by Pask, intended to elicit information about mental models (the Teach-Back method), and adopted it to detect differences in mental representations of users interacting with computers.

A situation is simulated where the respondent has to interact with a computer. The participant is asked to explain the computer's functioning to an imaginary counterpart, a colleague or a student, who has similar experience with the situation. The questions are designed to activate both declarative and procedural knowledge structures. They are presented on white sheets of paper, and the participants are instructed to express themselves in whatever way they consider most adequate: text, drawings, keywords, diagrams etc. In this manner, the participants are encouraged to externalize the mental model they made of the situation. Next, the protocols are scored "blind", along pre-defined scoring categories in order to map the respondent's mental representations. Rating implies (1) reading the protocol in its entirety and trying to understand fully what it says; (2) trying to formulate how the participant understands the space of the teach-back question and (3) classify the responses into relevant categories for the purpose of the study. Rating the answers is a complex task: the rater has to interpret the participant's intention and classify it by means of scoring rules. This task requires considerable training. In order to safeguard reliability, scoring is done independently by two or more persons.

3. Literature review

In the next sections, we will review the literature on understanding of programming constructs.

3.1. Cognitive aspects of (OO) programming

Robins, Rountree and Rountree (2003) provide us with an extensive literature review on research concerning learning and teaching programming between 1970-2003. In a survey study, Détienne (1997) assesses claims about the cognitive benefits of the OO paradigm.

Robins et al. mention different kinds of mental models involved in learning programming. Many studies have noted a central role played by a model (or abstraction) of the computer. Du Boulay et al. (1989) call this the "notional machine", an idealized, conceptual computer that is defined with respect to the language. Novices should develop an appropriate notional machine to master a programming language: the notional machine underlying Pascal is very different from the machine underlying PROLOG. A study by Mayer (as cited by Robins et al, 2003) confirms that students supplied with a notional machine model perform better than students who are not given the model.

A model of program comprehension is provided by Pennington (1987). Comprehension occurs in the context of a problem domain. The program's text is re-organized in mental representations with help of available knowledge structures. Pennington studies expert FORTRAN and COBOL programmers, and finds significant differences in their performances of comprehension tasks.

The OO approach has claimed to make modeling the problem domain easier for programmers. Détienne investigates this claim. She finds that novice programmers have difficulties in class creation. They need to start with a procedural representation of the situation. This seems to indicate that knowledge is organized in terms of procedures, not in terms of objects and relations. But for expert OO designers, the claims find support. Expert OO designers seem to shift between object view and procedure view.

3.2. Cognitive aspects of user-database interaction

Cognitive aspects of query languages were studied at the same time as cognitive aspects of programming (Reisner, 1981). One of the issues was the procedurality of the query language. Some query languages specify more step-by-step methods to obtain results than others. SQL, today's standard, is a set-oriented language, and was been labeled "non-procedural" in the debate.

There is no need to know how data are stored in a Relational Database in order to write a query: knowledge of the (abstract) data model of the database is sufficient. Chan, Wei and Siau (1993) focus on cognitive processes of abstraction in the user-database interface. They distinguish three levels of abstraction: a conceptual level (a description of the user's world), a logical level (describing the database world in mathematical terms) and a physical level (describing states in computer memory). De Haan and Koppelaars (2007) explicitly address database professionals. Professionals need to control the RDBMS, which is why database engineers should master the logical level, and the database world's description in terms of set theory. Database professionals work with "objects" in the database world, a mathematical construct. No studies on the cognitive aspects of this kind of user-database interaction are known to us.

4. A need for empirical study

The question here is how programmers, designers and professionals characterize the abstract software structures they work with. We observe that this characterization can change over time. Novice OO programmers mainly adopt a procedural strategy, expert OO designers are able to switch between object and procedure view. Knowledge of RDBMS software is not needed to formulate queries, but expert database professionals often do understand what is " inside the machine". Database experts, too, switch between abstract (relational) view and procedure view.

WM has limited capacity. Even if experts of both disciplines switch between object view and procedural view, this does not mean that they can use the two views simultaneously, or that they are able to switch while communicating with colleagues.

We found different references to mental models in the literature concerning cognitive aspects of (OO) programming and user-database interaction. These are: (1) the notional machine, simulating an idealized computer, (2) the object view, where the individuation of (problem domain) objects guides the design activity, emphasizing static aspects of the solution, (3) the procedure-centred view, emphasizing the dynamics of a program and (4) a set-theoretical model, describing the problem domain in abstract terms and thus defining the database world.

We hypothesize differences between groups in the instantiated mental models used to handle a concept that is fundamental to both disciplines: the "object".

5. A first experiment: how do professionals understand their systems?

We designed a teach-back protocol to elicit information about the participants' mental models and recruited students of comparable age and level of education, enrolled in different computing curricula. The participants' answers are scored along categories, derived from the four types of mental models mentioned in section 4. Three raters are involved in this experiment. Two are currently lecturing Computer Science in Dutch higher education, one is senior designer of documentation for scientific software. All the raters have a background in computer science and hold a Master's degree.

The protocol was tested in a pilot study with 6 participants, all lecturers of Computer Science in higher education. Based on the feedback, textual changes were made to the questions.

5.1 Questions

The protocol is introduced by a brief case description, an online Bookstore (see Figure 1).

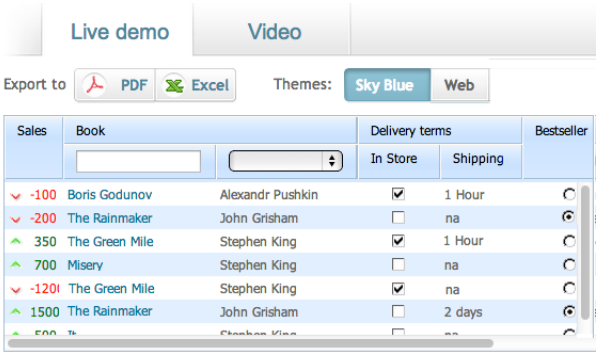


Figure 1: The context of the teach-back questions: an online Bookstore

The case description states that only information about books is relevant for our purposes, and that, therefore, the rest of the data is skipped. This results in a dataset with repeating rows having the same state. Participants are asked to count and describe the "Book-objects" they distinguish. No indications are provided for the choice of a theoretical context.

This situation is, in fact, ambiguous: the question can be answered by describing: (1) books in the bookstore's assortment (objects in the problem domain), (2) Book-objects in permanent storage (in the programming domain) or (3) instances of Book-objects (in the programming domain)

Participants are instructed to explain in writing to an imaginary fellow student: (1) How many different Book-object they see and what these objects are, (2) What happens if the system is asked to produce additional information about one of the books. Up to now, we have been concentrating on the analysis of the first part of question (1): "How may different Book-objects do you see?"

5.2 Scoring categories

To establish the number of different Book-objects, the rater scores one of the following answers:

- 4 objects (or 5, if the participant counts the last row, which is only partially visible),
- 6 objects (or 7, if the participant counts the last row),
- O (a number that is not traceable to Book-objects, or no number mentioned),
- B (both answers "4 objects" and "6 objects" are mentioned and explained).

5.3 Participants

The teach-back questions were answered by four groups of male Bachelor students, enrolled in professional curricula:

- 35 students attending 1st year classes Business IT & Management. Most of them enrolled in a professional curriculum in a computing discipline in 2013, 2 in 2010.
- 19 attending 3rd year classes Business IT & Management. Most enrolled in 2011, 1 in 2010, 1 in 2008.
- 18 attending 3rd year classes Computer Science, field of study Information Engineering. Most enrolled in 2011, 6 in 2010, 1 in 2009, 1 in 2002.
- 29 attending 3rd year classes Computer Science, field of study Software Engineering. Most enrolled in 2011. 4 in 2010.

All groups had attended at least one course "Relational Databases" and one course "Introduction to Programming in Java". The Business IT & Management curriculum emphasizes modelling. The

Computer Science curriculum concerns software development and has two fields of study. Software Engineering puts the emphasis is on programming, Information Engineering on the construction of business solutions. (Studiegids Hogeschool Utrecht, 2012).

5.4 Hypotheses

With this experiment, the following hypotheses are tested:

H1: "there is no significant difference between the conceptualization of the notion of "object" reported by 1st and 3rd year students Business IT & Management".

H2: "there is no significant difference between the conceptualization of the notion of "object" reported by members of the following groups: 3rd year students Business IT & Management, 3rd year students Information Engineering and 3rd year students Software Engineering".

6. Results

The following categories were scored:

- 4: 47 participants. Many of them explain their choice ("There are 6 Book-objects, but two of them occur twice"), indicating that they are counting sets of objects.
- 6: 26 participants. Explanations vary from "n rows, n Book-objects" to "6 Book-objects. Some occur twice, but they are different objects" and "I am thinking OO-Java".
- O: 28 participants.

Category "B" was not scored in our samples. The answers are summarized in Table 1 and Table 2.

	4	**6**	**O**	**n**
1st year Business IT & Management	19	6	10	35
3rd year Business IT & Management	7	1	11	19

Table 1: number of Book-objects counted by students Business IT & Management

We found differences close to significance between the samples in Table 1 (chi sqr = 4.87; $p<0.1$) and reject H1. Most students of the 1st year Business IT & Management count 4 objects. They seem to interpret "objects" as problem domain entities or as database entries. 3rd year students seem to be less certain in their interpretation: their answer is scored more frequently "O".

	4	**6**	**O**	**n**
3rd year Business IT & Management	7	1	11	19
3rd year Information Engineering	5	10	3	18
3rd year Software Engineering	16	9	4	29

Table 2: number of Book-objects counted by 3rd year students, sampled by curriculum

We found significant differences between the samples in Table 2 (chi sqr = 19.19; $p<0.01$) and reject H2. We conclude that future Information Engineers seem to interpret "objects" as instances: the answer "6 objects" is predominant. Future Software Engineers show the opposite preference and count 4 objects. Business IT & Managements students seem to elude the question.

7. Preliminary conclusions.

The analysis of the experiment has just started. It is too early to draw full conclusions about the conceptualization of the notion of "object". We can nevertheless underpin some observations.

Our investigation shows at least two ways to characterize the abstract notion of "object" that are currently used by professionals. These characterizations are not compatible and lead to different judgements about "objects". In the same situation, some professionals will identify 4 objects but others will perceive 6. The preference for 4 or 6 objects is not distributed ad random between professionals. We found indications for differences between groups of 3rd year students, enrolled in different computing curricula. Different preferences in the conceptualization of "object" can be a source of communication problems between groups of computing professionals. The lack of agreement about the definition of one of the basic notions of the discipline is alarming, just as the apparent difficulties to recognize this issue and to discuss it.

We suggest that professional organizations and institutes for higher education establish a refined terminology on this subject, with different terms for instances, stored objects and problem domain objects. This will not solve the Object Relational impedance mismatch, but will help professionals to discuss it without risking to add severe misunderstandings to the problems they are trying to solve.

8. Acknowledgements

We thank the Hogeschool Utrecht and Johan Versendaal for their support; we thank the students and their lecturers for their cooperation.

9. References

du Boulay, O'Shea, Monk, 1989: *The black box inside the glass box; presenting computing concepts to novices,* International Journal of man-machine studies 14: 237-249

Cañas, J. J., & Antolí, A. (1998). *The role of working memory in measuring mental models*. In Proceedings of the Ninth European Conference on Cognitive Ergonomics–Cognition and Cooperation. European Association of Cognitive Ergonomics (EACE): Rocquencourt, France.

Chan, H. C., Wei, K. K., & Siau, K. L. (1993). User-database interface: the effect of abstraction levels on query performance*. *MIS Quarterly*, *17*(4), 441-464.

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers*, *9*(1), 47-72.

De Haan, L., & Koppelaars, T. (2007). *Applied mathematics for database professionals*. Apress.

Ireland, C. (2011, January). Exploring the Essence of an Object-Relational Impedance Mismatch-A novel technique based on Equivalence in the context of a Framework. In *DBKDA 2011, The Third Int. Conference on Advances in Databases, Knowledge, and Data Applications* (pp. 65-70).

Johnson-Laird, Ph., *Mental Models*, chap. 12 from: The foundations of Cognitive Science, M.I. Posner (Ed.), Cambridge, MA MIT Press, 1989

Norman, D. (1983). *Some observations on mental models*, chap. 1 from: Mental models, Gentner, D., & Stevens, A. L. *(Eds.). (1983). Psychology Press.*

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, *19*(3), 295-341.

Reisner, P. (1981). Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys (CSUR)*, *13*(1), 13-31.

Robins, A., J. Rountree, N. Rountree (2003): *Learning and teaching programming: a review and discussion*, in: Computer Science Education 2003, vol. 13, pp. 137-172

Studiegids Bacheloropleidingen Institute for ICT 2012-2013 (2012), Hogeschool Utrecht, Utrecht, september 2012

van der Veer, G. C. (1990). *Human-computer interaction: learning, individual differences, and design recommendations*. PhD thesis, Vrije Universiteit, Amsterdam.

Learning Syntax as Notational Expertise when using DrawBridge

Alistair Stead

Computer Laboratory
Cambridge University
Alistair.Stead@cl.cam.ac.uk

Alan F. Blackwell

Computer Laboratory
Cambridge University
Alan.Blackwell@cl.cam.ac.uk

Keywords: POP-II.A. Novices, POP-III.D. Editors, POP-VI.E. Computer Science Education Research

Abstract

We report an experiment that explores the classroom application of DrawBridge, a system designed to support the acquisition of skill in using text-based programming language syntax. Graphical syntax is increasingly popular in educational programming languages such as Scratch, but anecdotal reports from classroom teachers suggest that students are now finding it difficult to make the transition to conventional text syntax. This research investigates text syntax use as a specific notational competence, and evaluates the use of DrawBridge to scaffold text syntax acquisition.

1. Introduction

Learning to code, like learning mathematics, involves skill in manipulating symbols. In mathematics, symbols are rather standardised - the conventions of symbolic algebra and so on. As a result, many learners see mathematics as being equivalent to the use of these particular symbols, and find it hard to imagine a different mathematics in which the same concepts might be associated with alternative symbol systems. However, learning to code involves not only familiarity with a particular symbol system (a programming language), but also the realisation that the learner may soon have to learn a completely different one while retaining some abstract meta-understanding of the underlying concepts. The "computational thinking" agenda, like the "new maths" of the 1960s, places an emphasis on teaching these underlying concepts rather than any particular language syntax (Wing, 2006).

In the psychology of programming field, it is understood that choices made with regard to language syntax and environment can have significant implications for usability of the language (Green, Petre, & Bellamy, 1991) (Green & Petre, 1996)). When designing languages for use in schools, it is certainly important that they be usable (McKay & Kölling, 2013) (Pane & Myers, 1996). Recent generations of instructional language and environment, such as Alice, Scratch and Greenfoot, have been extremely successful in improving usability by learners. However, the novel syntax used in some of these languages does not expose learners to some of the fundamental syntactic features of conventional languages: parsing, identifiers, delimiters and so on. Furthermore, these very features are well known (to teachers of programming) to pose significant obstacles to learners, just as the correct use of mathematical notation can be a significant obstacle to those learning algebra.

In programming language design, a distinction is often made between keywords, identifiers, and the other elements of concrete syntax that are used to delimit and relate that vocabulary. Typically, concrete syntax such as operators and delimiters are implemented using punctuation symbols and whitespace. The popularly described "visual languages" often include keywords and identifiers, but replace much of the detailed concrete syntax with colouring, region boundaries, and other graphic devices. We describe this as "graphical syntax", by comparison to the textual concrete syntax that is constructed from punctuation marks and whitespace.

1.1. Notational Expertise

The question addressed in this research is whether there might be better ways of acquiring conventional coding skills - the skills associated with conventional code syntax. In contrast to "computational thinking" that focuses on abstract concepts independent of any symbol system (Wing, 2008), we might regard this objective as teaching *notational expertise*. We are not necessarily advocating notational expertise as an opposing view, or as a new and competing curriculum priority. We simply observe that it brings a different perspective to the teaching of coding, and one that might usefully be explored through experiments. Furthermore, the concept of notational expertise offers a novel research question for the psychology of programming, which has well-established understanding with regard to the usability of notation, but not necessarily with regard to the learnability of notation. In mainstream HCI, it has long been understood that learnability and usability are alternative and complementary requirements.

We are conducting this research through the design and evaluation of a specific educational system, called DrawBridge. DrawBridge aims to provide a motivational and scaffolded (Wood, Bruner, & Ross, 1976) (Guzdial, 1994) approach to learning programming language syntax. There are two scaffolding strategies. One of these allows learners to construct a program initially by manipulating a block-syntax representation (as used in Scratch), and subsequently by viewing and modifying the same program in conventional text syntax. The other scaffolding strategy allows users to create example code with programming by demonstration - making animations by dragging objects on a canvas. The motivational elements of DrawBridge support broader creative experiences: the animated objects are captured from drawings made on paper by the students, and the resulting animation can be viewed directly as a web page running in a browser.

1.2. Outline of Paper

The remainder of this paper is arranged as follows: the next section briefly describes the functionality of DrawBridge. We then describe two specific research questions related to the development of notational expertise, and two studies - an interview-based pilot study and a larger classroom experiment - investigating these questions. Finally, we discuss the finding of these studies, and an agenda for further investigation of notational expertise in the context of the DrawBridge system.

2. Overview of DrawBridge

DrawBridge adopts a familiar strategy that is shared by mathematics education and user interface design, of introducing abstract concepts via concrete manipulation. This general strategy is influenced by the Piagetian model of childhood development, which observes that children develop abstract mathematical reasoning skills only after they become practiced at concrete physical action. Alan Kay famously adapted Piaget's model of childhood stages in his slogan for the Smalltalk language and WIMP GUI: "doing with images makes symbols" (Kay, 1990). This strategy is applied in DrawBridge by providing students with a tool that allows them to provide their own concrete drawings (made using pen and paper), which are then captured and segmented for use as digitally animated characters. DrawBridge literally presents users with a sequence of stages, displayed on the screen in pairs, in which concrete drawing becomes represented as a more abstract geometric object that can then be manipulated using mathematical operations, as in Figure 1. DrawBridge initialises in the leftmost position.

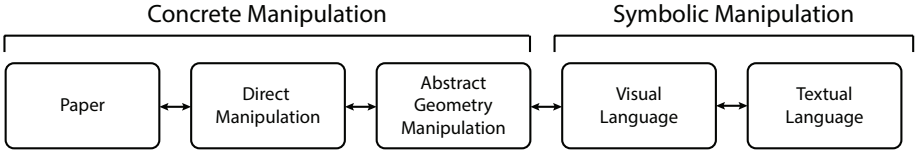


Figure 1 - DrawBridge Representations

Children find it highly motivating to draw their own characters. This has been noted as a limitation of Alice, where the complexity of creating 3D character models means that the system is restricted to providing (extensive) libraries of predefined characters. In Scratch, new sprites can easily be drawn

by children using a simple bitmap editor. DrawBridge simplifies the drawing process further, by supporting pen and paper with automated segmentation. However, as with Alice and Scratch, the introduction to programming occurs when the learner writes code to control the behaviour of the characters. Both Scratch and Alice use graphical syntax editors to support experimentation and minimise distracting syntax errors. However, as discussed already, this results in the trade-off that those languages do not explicitly teach the elements of conventional text language syntax that are central to much programming.

DrawBridge therefore adds a second scaffolding component, which allows learners to explore text syntax by use of an intermediary graphical syntax editor. As the user manipulates the hand-drawn characters to construct an animation program by demonstration, this program is initially displayed in a graphical block-syntax form that visually resembles Scratch or Alice (although with many small design differences, not discussed in detail in this paper). The user can experiment with behaviour modifications by manipulating the block syntax directly. However, users are also able to move on from block syntax to see and edit the same behaviour in conventional text syntax (here, valid JavaScript). The block syntax and JavaScript text syntax can be viewed side by side, with live, bidirectional interaction such that editing either of them also updates the other. The result is a scaffolded multi-representation editor that can be used to explicitly teach the principles of textual code syntax.

The remainder of this paper describes an initial study evaluating the design intentions of DrawBridge.

3. Research questions

There are two primary research questions in this study:

1. To assess the educational benefit of moving from concrete drawing to symbolic mathematical operations

2. To assess the educational benefit of using graphical block syntax in order to scaffold learning of text syntax conventions.

4. Experimental design

We explored these questions via a controlled experimental design, administered in a classroom setting. We manipulated two factors, via modifications to the DrawBridge system:

Concrete drawing (C): We created a version of the DrawBridge system in which there was no concrete drawing phase, so that participants started the task by manipulating geometric representation rather than their own drawings.

Visual-first versus Text-first (VF/TF): We created a version of the DrawBridge system in which participants used the conventional text representation before using the block-syntax representation.

The experimental measures tested participants' understanding of syntax use before and after using DrawBridge, administered as a classroom pre-test and post-test.

The study was conducted over a period of two weeks, involving two sessions with the same class of students. In the first session, the class was divided into four groups, with each group using a different version of DrawBridge as determined by a Latin square design based on the two experimental factors:

C-VF	C-TF
Concrete drawing phase	Concrete drawing phase
Visual block syntax first	Conventional text syntax first
VF	**TF**
No drawing phase	No drawing phase
Visual block syntax first	Conventional text syntax first

Table 1 – Experimental design.

All groups completed the same pre-test and post-test at the start and end of the session, which consisted of a multiple-choice component and a translation component. DrawBridge was also instrumented to record data concerning time spent on each representation pair, error information, button clicks and navigation.

In the second session, all students used the same version of DrawBridge, corresponding to the C-VF condition. The reason for this was that a large difference in educational outcomes had been observed during the first session, as discussed in more detail below. The second session was therefore intended to provide all students with access to the version of DrawBridge expected to be most educationally beneficial, in order that no student would have been disadvantaged through participation in the study.

At the end of the second session, a return test was administered, in the same format as the pre-test and post-test.

5. Participants

A complete class of twenty-one school students was recruited from an independent school in Cambridgeshire. Participants were studying in Year 7, aged 11-12, and had a range of academic aptitudes. The four conditions were allocated to participants according to blocks of seating in the classroom, so that participants were not distracted by neighbours carrying out different activities. In session 2, three members of the class were absent, and the session was conducted with 18 participants.

6. Procedure

In session 1, the procedure was:

1. Questionnaire regarding previous computing and programming experience
2. Pre-test assessing prior knowledge of JavaScript syntax (see appendix)
3. Task using DrawBridge, allocated to participants according to experimental condition
4. Post-test assessing knowledge of JavaScript syntax after using DrawBridge

In session 2, the procedure was:

5. All participants complete DrawBridge task in the C-VF condition
6. Return-test assessing knowledge of JavaScript syntax
7. Questionnaire assessing participants' experiences and attitude to DrawBridge (see appendix)

The experimental task in the Concrete + Visual-First (C-VF) condition corresponded the ordered sequence of stages in DrawBridge as follows:

a) Draw a character on paper, which is photographed and loaded into DrawBridge by the experimenter

b) Adjust the size and location of the character on screen using the image direct manipulation screen

c) Demonstrate and record an animation using the abstract geometry screen

d) Add an extra step to the animation using the visual block syntax screen

e) Modify and extend the animation using the text syntax screen

In the Concrete + Text-First (C-TF) condition, the same sequence is followed, but with the representation formats in steps d) and e) reversed.

In the two conditions without the concrete drawing phase (VF and TF), participants did not carry out any activity equivalent to stages a), b) and c). Instead, they were provided with a predefined program, which they were asked to enter into DrawBridge, using either the visual block syntax screen or the text syntax screen according to condition VF or TF respectively. After viewing the resulting conversion to text (from VF) or to visual blocks (from TF), they were then invited to recreate the program by translating it directly from the predefined program sheet.

7. Results

Most participants had previous experience of programming in Scratch, which is included in their school curriculum. Nevertheless, only 10/21 reported that they had programmed a computer before. Despite the fact that they did not consider this to be “programming”, 15/21 reported that they had created animations before, and that Scratch was the tool they had used. 18/21 reported either that they loved using computers, or found it OK. 11/21 reported a "good" level of computer skill, with 5 reporting skill below this level, and 5 reporting skill above.

An initial analysis was conducted to compare the four experimental groups, according to results obtained in the pre-test. Unexpectedly, participants allocated to the TF group performed significantly better in the syntax knowledge pre-test, with mean score 6.6/10, compared to means of 2.2, 3.2 and 3.8 in the VF, C-TF and C-VF groups ($p < .01$), see Figure 2. This data indicates a lack of balance in the sample groups rather than a treatment effect, given that it was collected before any treatment was administered. It is possible that this resulted from the decision to allocate participants to conditions according to the location of their seats in the classroom, and that more able students may have been grouped in one part of the class.

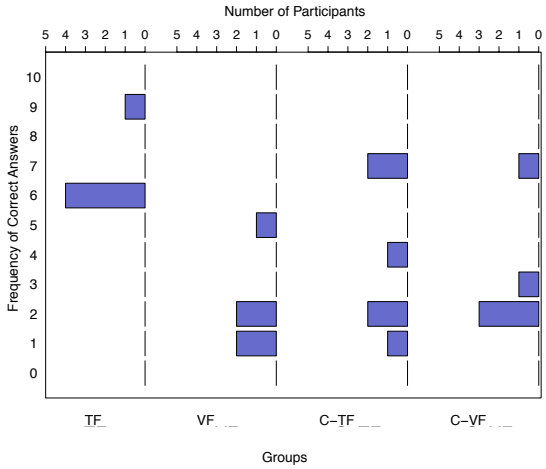


Figure 2 – Pre-Test Syntax Assessment

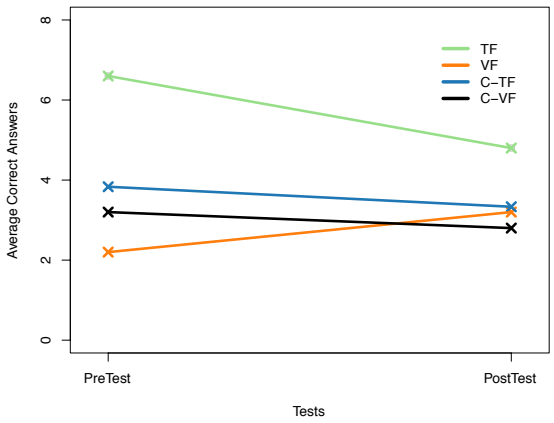


Figure 3 – Absolute Test Values

A comparison between pre-test and post-test results for the four experimental groups is shown in Table 1 and Figure 3. It is apparent that the high performance of the TF sample was not maintained in the post-test evaluation. On the contrary, performance on the post-test reduces substantially, although a pairwise t-test indicates the difference is not significant. Two possible explanations for this are either i) that the high pre-test scores were a coincidence, and that the lower post-test score represents reversion to the mean; or ii) that the TF condition impedes student's understanding of text syntax, such that otherwise capable students perform less well after carrying out this task. In order to explore

this further, we corrected for this pre-test sample difference by calculating change in test score between the pre-test and post-test, rather than absolute number of correct answers.

Group	Pre	Post	Return	Post-Pre	Return-Post	Return-Pre
TF	6.6 ± 1.67	4.8 ± 2.54	3.8 ± 1.62	-1.8 ± 2.96	-1.0 ± 2.22	-2.8 ± 5.01
VF	2.2 ± 2.04	3.2 ± 2.69	4.2 ± 3.09	1.0 ± 3.04	1.0 ± 3.51	2.0 ± 6.51
CTF	3.83 ± 2.77	3.33 ± 3.36	4.2 ± 2.83	-0.5 ± 3.74	0.87 ± 2.91	0.37 ± 5.00
CVF	3.2 ± 2.69	2.8 ± 2.04	3.67 ± 7.17	-0.4 ± 1.42	0.87 ± 5.74	0.47 ± 3.35

Table 1 – Assessment Marks and Change Scores (Green – above mean, Red – Below mean, Confidence Interval of 0.95)

After correcting for pre-test sample differences, we see that the visual-first treatment is associated with improved performance in the post-test, and that the concrete treatment is associated with slightly better performance than the conditions without the concrete task (see Figure 4). We also see that groups using concrete representations performed better on average in the post-test than groups not using concrete representations (4.0 vs. 3.1 mean respectively). However, the difference was not significant ($p = 0.38$) and there was little difference in the change score between the pre-test and post-test.

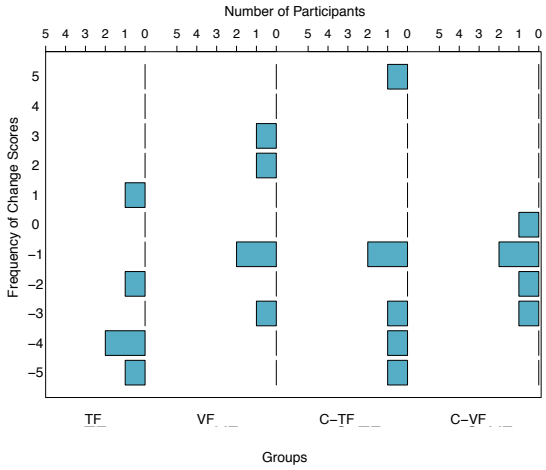


Figure 4 – Pre-Test to Post-Test Change Scores

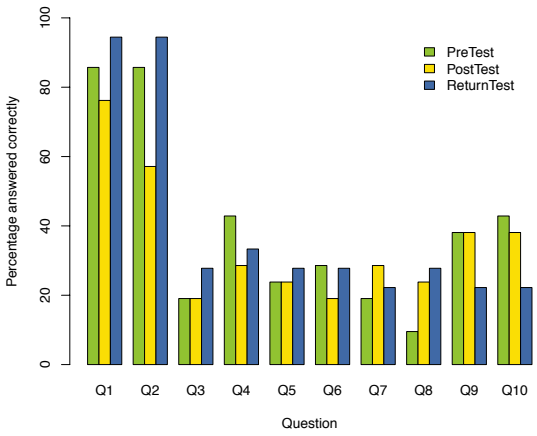


Figure 5 – Correct Answers by Question

In response to the overall reduction in performance that was observed between the pre-test and post-test, a further lesson was arranged, in which all participants carried out the C-VF task. The result of the return-test administered after this task is shown in Table 1 and Figure 6. On average, performance improved in the syntax component of the return-test, with the exception of the group originally assigned to the TF condition, which decreased even further. This suggests that, of the two explanations for the original fall in performance, reversion to the mean seems the most likely.

The percentage of correct answers for each question in the multiple-choice component is shown in Figure 5. In order to improve participant self-efficacy, we included two "easy" questions relating to the validity of a web address and email address. Participants' answers for these questions were significantly better than other questions ($p < .05$) as expected. However, despite this and other attempts to reduce anxiety and increase participation, test results suffer from poor reliability - many participants provided no answer to a substantial number of the translation questions, or gave the same answer to every multiple-choice question.

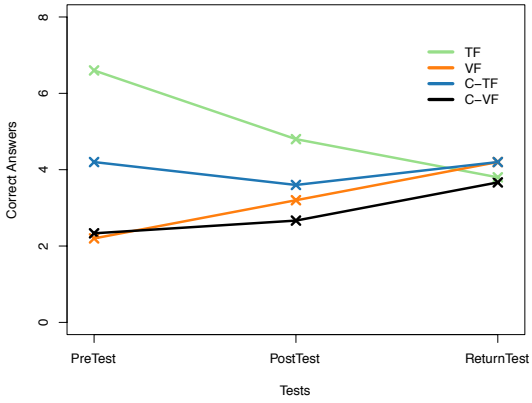


Figure 6 – Absolute Test Values

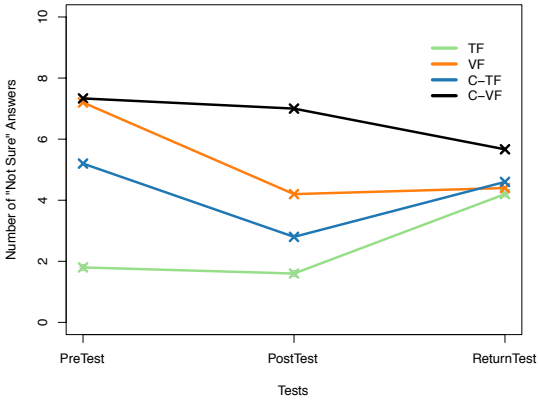


Figure 7 – "Not Sure" responses

The number of "not sure" responses in the multiple-choice component decreased for VF and C-TF between the pre-test and post-test, suggesting that self-efficacy improved for these groups (see Figure 7). Responses stayed reasonably constant for TF and C-VF in the post-test, before all groups converged towards a mean value of 4.5 in the return session.

A comparison of the time spent viewing symbolic representations is shown in Figure 8. Participants in without-concrete groups spent significantly longer viewing symbolic representations than with-concrete groups ($p < .001$, 3m 23s on average vs. 38m 35s). With-concrete groups spent most time (19m 51s on average) viewing the first pair of representations, which included direct manipulation of automatically segmented characters.

All but two participants present on the second visit reported that they loved, liked or did not mind using DrawBridge, with all but three happy to use DrawBridge again. When participants were asked to report the three best things about DrawBridge, they agreed that drawing their own characters and being able to animate was a clear strength - *"see your own animations. it's fun! it's different!" and "You can do some cool animation, it's fun to do, being able to control the characters."* When asked about the three worst things, reliability was noted as the main problem - *"sometimes it froze"*, and *"It's a bit slow sometimes"*.

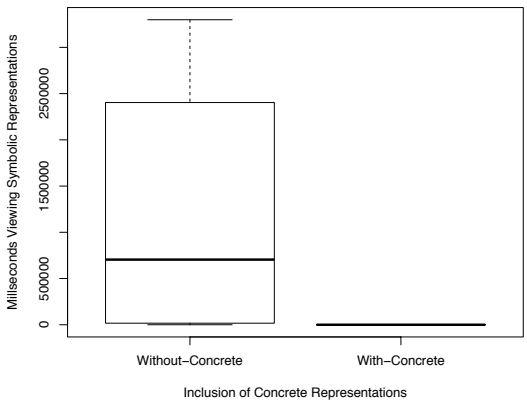


Figure 8 – Instrumentation Viewing Time

8. Discussion

Our aim in this study was i) to assess the educational benefit of moving from concrete drawing to symbolic mathematical operations and ii) to assess the educational benefits of using graphical block syntax in order to scaffold learning of text syntax conventions.

Our pre-test results suffered from imbalance, apparently caused by an association between participants' seating arrangements and ability. We therefore focused on change scores between tests, in accordance with best practice for non-equivalent group design analysis (May, 2012), and previous research (Levy, Ben-Ari, & Uronen, 2003) However, change-score analysis can be prone to regression artefacts (Reichardt, 2005), which occur when groups with extreme pre-test values regress towards the mean in the post-test. Given the extreme pre-test results of group TF, it is likely that the decline observed in post-test results is due to regression to the mean rather than exposure to DrawBridge.

The change scores relating to the benefits of the inclusion of concrete drawing representations suggest that there was no significant educational benefit to including concrete drawing representations. However, instrumentation results show that participants in without-concrete groups spent 11.43 times longer viewing symbolic representations than with-concrete groups, which could explain the lack of difference. Furthermore, results from the questionnaire given on the second visit showed that drawing on paper and manipulating characters for animation provided a major motivation for participants.

Results relating to the order of visual and text representations show that groups who encountered the visual representation first showed the highest change scores between the pre-test and post-test (see Figure 4), suggesting that exposure to and scaffolding from the visual representation before the text representation was beneficial to users. Return-test results show that visual-first groups showed the highest improvements, suggesting that longer exposure to visual-first groups also proved to be beneficial.

The difference between the number of answers in which participants responded "Not Sure" between the pre-test and return-test was significantly ($p < .05$) affected by order: visual-first groups VF and C-VF had a reduced number of "Not Sure" answers (-2.375 on average), while text-first groups had an increased number (0.9 on average), suggesting that the visual-first order may increase self-efficacy. However, is possible the difference could be due to faster maturity of participants in the visual-first groups, who had longer exposure to the visual-first version of DrawBridge used in the return test.

9. Conclusion

We presented an experiment that explored the classroom application of DrawBridge, a novice-programming environment that transitions users from concrete representations to abstract, symbolic representations in an effort to scaffold syntax acquisition. Results show that users who were able to use the visual representation first improved more than those encountering the text representation first. We also found that the use of a pen and paper as a starting point for programming provided major motivation to participants. Unfortunately, the experiment suffered from an imbalance of pre-test results and low response rates in some questions, resulting in weakened internal validity. However, steps were taken to address the initial imbalance by using a return test in addition to the pre-post comparison. Future work will explore students' experience of DrawBridge through more in-depth qualitative interviews, leading to refinement of the DrawBridge prototype and the evaluation tests used. Based on the outcome of that study, we expect to carry out further classroom evaluation.

10. References

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages and Computing* (pp. 1–51).

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the "Match-Mismatch" Conjecture. *ESP*, *91*(743), 121–146.

Guzdial, M. (1994). Software-Realized Scaffolding to Facilitate Programming for Science Learning. *Interactive Learning Environments*, *4*(1), 37–41.

Kay, A. (1990). User Interface: A Personal View. *The Art of Human-Computer Interface Design*.

Levy, R. B.-B., Ben-Ari, M., & Uronen, P. (2003). The Jeliot 2000 Program Animation System. *Computers & Education*, *40*(1), 1–15.

May, H. (2012). Nonequivalent Comparison Group Designs. *APA Handbook of Research Methods in Psychology, Vol 2: Research designs: Quantitative, Qualitative, Neuropsychological, and Biological* (Vol. 2, pp. 489–509). Washington, DC, US: American Psychological Association.

McKay, F., & Kölling, M. (2013). Predictive Modelling for HCI Problems in Novice Program Editors. *27th International British Computer Society Human Computer Interaction Conference*. London, UK.

Pane, J. F., & Myers, B. A. (1996). Usability Issues in the Design of Novice Programming Systems, (August).

Reichardt, C. (2005). Nonequivalent group design. *Encyclopedia of Statistics in Behavioral Science*.

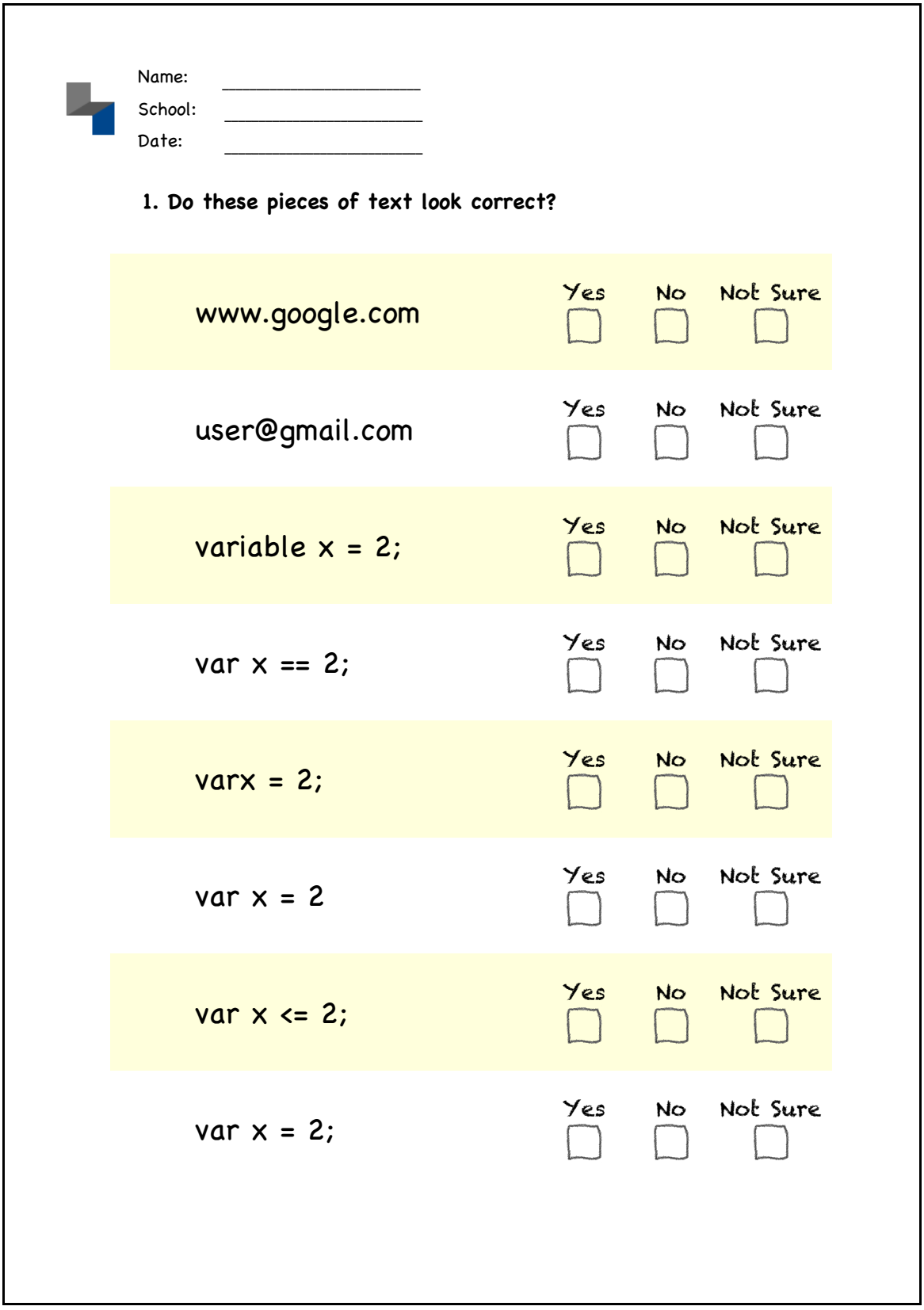
Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33.

Wing, J. M. (2008). Computational Thinking and Thinking about Computing. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, *366*(1881), 3717–3725.

Wood, D., Bruner, J. S., & Ross, G. (1976). The Role of Tutoring is Problem Solving. *Journal of Child Psychology and Psychiatry*, *17*(2), 89–100.

11. Appendix

11.1 Pre-Test



Name: _____

School: _____

Date: _____

1. Do these pieces of text look correct?

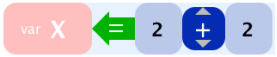
	Yes	No	Not Sure
www.google.com	☐	☐	☐
user@gmail.com	☐	☐	☐
variable x = 2;	☐	☐	☐
var x == 2;	☐	☐	☐
varx = 2;	☐	☐	☐
var x = 2	☐	☐	☐
var x <= 2;	☐	☐	☐
var x = 2;	☐	☐	☐

Pre-Test Page 2

setImagePosition 100; Yes ☐ No ☐ Not Sure ☐

setImagePosition(100); Yes ☐ No ☐ Not Sure ☐

2. **Can you fill in the missing pieces?**

Blocks	Code
	
	`setTimeToDestination(500);`
	
	`loadImage(2, 541, 173, 238, 168);`
	
	`document.getElementById("myCanvas");`
	
	`var X = 2 / 2;`

11.2 Return Questionnaire

Name: _____

School: _____

Date: _____

1. Overall, how much have you enjoyed using DrawBridge?

Hated it ☐ Disliked it ☐ Didn't mind it ☐ Liked it ☐ Loved it ☐

2. Would you like to use DrawBridge again?

Never ☐ Not really ☐ Don't mind ☐ OK ☐ Definitely ☐

3. What are the three best things about DrawBridge?

4. What are the three worst things about DrawBridge?

5. How can we improve DrawBridge?

Blinded by their Plight: Tracing and the Preoperational Programmer

Donna Teague

Queensland University of Technology
Australia
d.teague@qut.edu.au

Raymond Lister

University of Technology, Sydney
Australia
raymond.lister@uts.edu.au

Keywords: POP-I.B. barriers to programming, POP-II.A. novices, POP-V.A. Neo-Piagetian, POP-V.B. protocol analysis, POP-VI.E. computer science education research

Abstract

In this paper, we present evidence that some novice programmers have the ability to hand execute ("trace") small pieces of code and yet are not able to explain what that code does. That evidence is consistent with neo-Piagetian stage theory of programming. Novices who cannot trace code are working at the first stage, the sensorimotor stage. Novices who are working at the preoperational stage, the second stage, can trace code but do not yet have a well-developed ability to reason about the code's purpose, other than by induction from input/output pairs. The third stage, the concrete operational stage, is the first stage where novices can reliably reason about code. We present data from think aloud sessions that contrast the behaviour of preoperational and concrete students while they attempt to reason about code.

1. Neo-Piagetian Stages of Development

Lister (2011) proposed that we could describe novice programmers' behaviour using neo-Piagetian stage theory. This theory is based on the premise that there are consecutive, cumulative stages through which we develop increasingly more mature abstract reasoning and expertise in a domain.

1.1 Sensorimotor Stage

Sensorimotor is the first, and least mature, stage of development. It is at this stage that misconceptions about basic programming concepts most influence the novice's behaviour, like those misconceptions described by du Boulay (1989). A sensorimotor novice programmer has as yet minimal language skills in the domain and is still learning to recognise syntax and distinguish between the various elements of code. At this stage the novice requires considerable effort to trace code (i.e., hand execute), and only occasionally do they manage to do so accurately.

1.2 Preoperational Stage

At the next more mature stage, the preoperational novice has made headway into mastering basic programming concepts, with most misconceptions having now been rectified. This makes it possible for them to more consistently trace code accurately. However, the preoperational novice is heavily reliant on the use of specific values to trace, understand and write code. Preoperational novices are not yet able to perform abstract reasoning about a chunk of code, as their focus is quite narrow: limited to simply a single statement or expression at a time. They struggle to recognise the relationship between two or more statements.

1.3 Concrete Operational Stage

By the time a novice is at the concrete operational stage, their focus shifts from individual statements to small chunks of code which allows them to consider the overall purpose of code. Their ability to reason at a more abstract level allows them to understand short pieces of code simply by reading that code. When the concrete operational novice does trace, they can do so in an abstract manner rather than being reliant on the use of specific variable values. One of the defining characteristics of the

concrete operational novice is the ability to perform transitive inference: comparing two objects via an intermediary object. For example, if `A > B` and `B > C` then, by transitive inference, `A > C`.

1.4 Piaget -v- Neo-Piaget

Whereas Piaget himself focussed on the cognitive development of children, neo-Piagetian theory is concerned with cognitive development of people of any age, learning any new task. A person can thus concurrently exhibit characteristics from different stages in different knowledge domains. Using a methodology based on Piaget's theory of genetic epistemology, da Rosa (2007) witnessed in her research participants the transition of reasoning about relationships towards the construction of new recursive concepts. According to neo-Piagetian theory, time taken by individuals to transition through the stages varies, but there are conflicting theories about the nature of those transitions which we will discuss in the next section.

1.5 Staircase Model -v- Overlapping Wave Model

We previously alluded to conflicting theories about the nature of the transitions between neo-Piagetian stages. Although theorists agree that the neo-Piagetian stages are consecutive and cumulative, one view is that the stages are discrete, much like a stair-case model. However, there is a growing body of evidence suggesting that progress through the stages may not be so straightforward. How, for example, does one make the quantum leap from one stage to the next? An alternative to the stair-case model is that the transition through the stages can be seen as overlapping waves: where a person exhibits characteristics from two or more stages as they develop skills in the domain (Siegler, 1996; Boom, 2004; Feldman, 2004). In this overlapping waves model, characteristics of the earliest stage dominate behaviours initially, but as cognitive progress is made there is an increase in use of the next more mature level of reasoning and a decrease in the less mature. In this way, there is concurrent use of multiple stages of reasoning. This model is depicted in Figure 1. As will be apparent later in this paper, some novice programmers' behaviour that we have observed fits this overlapping waves model.

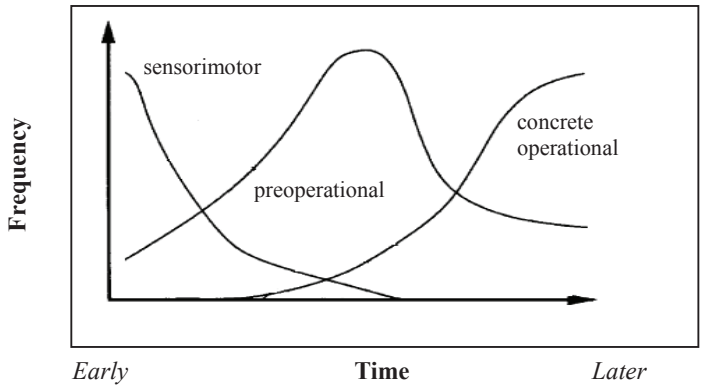


Figure 1 Overlapping Waves Model

2. Methodology

Previous studies have found evidence that novices find explaining code harder than tracing code (Lister, Simon, Thompson, Whalley, & Prasad, 2006; Whalley et al., 2006; Lister, Fidge, & Teague, 2009; Simon, Lopez, Sutton, & Clear, 2009). Philpott, Robbins and Whalley (2007) found that students who could not accurately trace were not able to explain similar code.

But if a novice has the skills to accurately *trace* a piece of code, shouldn't they then have an adequate understanding of it to be able to explain the purpose of that same piece of code? In this paper, we gathered empirical data to help us answer this question. We gave students some code and asked them to both *trace* and *explain* its purpose.

At most institutions, Explain in plain English (EPE) questions are not as familiar to most programming students as tracing and writing tasks. At our institution, however, students encountered EPE questions in their lectures and previous tests and therefore were familiar with what type of

answer was expected of them. They had not necessarily both traced and explained the same piece of code before. We asked students to do that in this study, first to establish that they understood the semantics of the code by being able to trace it, and second, if they could indeed trace it, to determine if they were also able to explain its purpose.

2.1 In-Class Testing

We tested introductory programming students with trace and explain tasks at our university during their lecture in four different (13 week) semesters. The students involved in these in-class tests had already completed one semester of programming. Students were asked to complete the tests individually, as if they were sitting an exam, but the tests did not contribute to their final grades.

In the sixth week of each of the four semesters, we gave the students the programming tasks shown below in Figure 2. (Note that the line numbers next to the code in Figure 2 were inserted by the authors of this paper for the readers' benefit, and were not part of the exercise given to the students.) Sample answers are provided in the shaded areas of the figure. The concepts that these tasks use (selection and output) were covered in week four of their first unit of study, so in effect, students had exposure to these concepts for 15 teaching weeks. In other words, the programming concepts in the tasks were quite familiar to them.

Consider the following block of code, where variables `a`, `b` and `c` each store integer values:

```
1      if (a > b) {
2          if (b > c) {
3              Console.WriteLine(c);
4          } else {
5              Console.WriteLine(b);
6          }
7      } else if (a > c) {
8          Console.WriteLine(c);
9      } else {
10         Console.WriteLine(a);
11     }
```

(a) In relation to the above block of code, which one of the following values for the variables will cause the value in variable `b` to be printed?

(i) a = 1; b = 2; c = 3; (ii) a = 1; b = 3; c = 2;
(iii) a = 2; b = 1; c = 3; (iv) a = 3; b = 2; c = 1;

Correct answer: (iii)

(b) In one sentence that you should write in the box below, describe the purpose of the above code (i.e. the `if/else if/else` block). Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code:

Sample answer: *To print the smallest of the three given values.*

Figure 2 - Trace and Explain Tasks

Although informally invigilated, our in-class testing was not conducted under formal exam conditions and students may have been less motivated to complete the tests than if those tests had contributed to their grades. On the other hand, our students also had less motivation to plagiarise.

2.2 Think Aloud Sessions

An issue with any type of written exam is that test scripts (i.e., the papers that the students hand in) are sometimes not an accurate indication of students' ability at all, and certainly rarely give any insight into the process they used to arrive at an answer (Teague et al., 2012). We wanted evidence of how students traced and reasoned about code. Artefacts from think aloud sessions are potentially a much richer source of data which describe the students' *process* of solving tasks (Ericsson & Simon,

1993; Atman & Bursic, 1998). So to complement any quantitative findings from our in-class tests, we also ran a series of think aloud sessions with volunteer students from our first two introductory programming classes and asked them to complete an exercise similar to that shown in Figure 2. Note that our think aloud students came from different cohorts using different programming languages, but essentially there were only syntactic differences in the code. The exercise was presented to them in the same manner as the in-class test, except the exercise was printed on special dot paper on which they wrote their answers with a SmartPen (LiveScribe, 2014). This allowed us to see what students wrote and record what they said as they did so.

3. Results

3.1 In-Class Testing Results

Table 1 shows the performance of students on the tasks in Figure 2, from four different cohorts, in four different semesters (each cohort is a row in the table). The last row of that table combines the four cohorts.

A great proportion of the students we tested over the four semesters were able to answer the tracing question correctly (see Col. 3 of Table 1). A much smaller percentage could actually explain the code (see Col. 5 of Table 1). A total of 29% of the students ***could*** trace the code and therefore had a working knowledge of the programming concepts involved, but ***could not*** explain what that code did (see Col. 2 of Table 1).

Considering students were working with the same code for both tasks, these results seem to be surprising. Why were so many students unable to explain that code when they could trace that code?

	Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6
n	Can trace (a) and can explain (b)	**Can trace (a) but cannot explain (b)**	Can trace	Cannot trace (a) but can explain (b)	Can explain	Can neither trace (a) nor explain (b)
51	31 (61%)	**10 (20%)**	41 (80%)	1 (2%)	32 (63%)	9 (18%)
113	40 (35%)	**31 (27%)**	71 (63%)	0 (0%)	40 (35%0	42 (37%)
53	27 (51%)	**21 (40%)**	48 (91%)	1 (2%)	28 (53%)	5 (9%)
86	51 (59%)	**26 (30%)**	77 (90%)	5 (6%)	56 (65%)	4 (5%)
303	149 (49%)	88 (29%)	237 (78%)	7 (2%)	156 (51%)	60 (20%)

Table 1 Comparison of students' performance on the trace and/or explain tasks in Figure 2

3.2 Think Aloud Sessions Results

As we have seen, a significant number of students in our in-class tests were not able to explain the code even though they could trace it. However, only two of the students who took part in think aloud sessions were able to trace the code but were not able to explain the code. Although two students is much too small a sample size from which to draw conclusions or generalise (about why students are not able to explain the purpose of code), an analysis of these two students' *process* of completing the tasks is insightful. It gives us evidence that their ability to do one task and not the other can be explained by neo-Piagetian theory.

For anonymity, our think aloud students chose aliases, by which we will refer to them. The think aloud sessions with four students are summarised in Table 2. Later in the paper, we will discuss in detail the difficulties encountered by two of those students, Michael and Charlotte, as they completed the tasks in Figure 2. But first, by way of comparison, we introduce the other two students, Lance and Briandan, who completed the exercise without difficulty, and did so in ways we had originally anticipated all of our students would complete it.

Alias	Weeks of prior programming instruction	Time taken to complete exercise (min:seconds)	Dominant neo-Piagetian stage demonstrated by behaviours
Lance	10	2:14	Concrete
Briandan	26	3:06	Preop/Concrete
Michael	6	7:26	Preoperational
Charlotte	6	8:03	Sensorimotor/Preoperational

Table 2 Summary of Think Aloud Sessions, with Subjects in Order of Reasoning Sophistication

Our detailed excerpts which follow use a format similar to that used previously in qualitative studies (Lewis, 2012; Teague & Lister, 2014a), where the interview data is presented separate to its analysis, so that the reader may more easily follow the think aloud session.

In this paper, pauses in speech are marked "...", as placeholders for dialog we have not included because we deemed that the excluded dialog added nothing to the context of the think aloud session. Utterances are italicised and where we have added our own annotations for clarification, these appear in square brackets in non-italicised text.

3.2.3 Lance

Summary

Lance completed a Python version of the exercise in Figure 2 without any fuss, in little more than 2 minutes. He did not trace with specific values. While reading the code in Figure 2 he spontaneously determined the purpose of the code.

Data

After very quickly reading the code almost in its entirety, Lance made the comment:

Lance: *Ah that's a bit of a mind warp.*

He made a mark in the code (line 5 in Figure 2) indicating the part of the code that needed to execute.

Lance: *... so basically to get there* [line 5] *we need* `a` *to be greater than* `b` *and we need* `b` *to be greater than* `c` *... oh no we need* `b` *to be ...* <u>*less*</u> *than* `c` *... so we need* `b` *to be the smallest number*

To verify his thinking, Lance then traced the code with the set of values in option (iii)

Lance: `a` *... is greater than ...* `b` *... yes ...* `b` *is ... greater than* `c` *... no ... so it doesn't print* `c` *... and then it goes to the* `else` *statement print* `b` *so ... yep so (iii)*

As part of answering part (a) in Figure 2, Lance had already explained the purpose of the code, so he was able to write his answer to part (b) without hesitation.

Analysis

Lance's comment about the code being "*a bit of a mind warp*" makes us believe that on first reading, he had not formed a clear understanding of the code. However, he then determined which conditions must be met in order for the required output statement to print. We refer to this as a "backward" trace. The ease with which he volunteered an explanation of the purpose of the code in part (b) of Figure 2 indicates that he had already processed a great deal of the code's semantics while completing part (a).

Lance had no real need to trace the code with the specific values in the options, as his abstract trace of the code and conclusion that "*we need* `b` *to be the smallest number*" was sufficient to identify the correct option. However, he traced the code using the values from option (iii) to confirm the answer at which he had arrived. He did not at any stage refer to any of the code after line 6 that is, the `else` branches of the first `if` block.

Lance formed a coherent understanding of the "big picture" as a by-product of his trace. It was his grasp of the concept of *transitive inference* (comparing two things via an intermediary) that allowed him to quickly determine the code's purpose, that is, that if `a > b` and `b < c` then `b` is the smallest. Transitive inference is a defining quality of the concrete operational stage.

3.2.1 Briandan

Summary

Briandan completed a Java version of the task in part (a) of Figure 2 by doing a "backward" trace, much like Lance had done. However, she was a little more reliant on specific values than Lance. Briandan eliminated two options based on the first condition not being met, then traced the code with the values given in the remaining two options to find the correct option. For the EPE task (part (b) in Figure 2) she did little else other than to re-read the given code, consider its purpose and then correctly describe that purpose. Whereas Lance had already spontaneously formed a clear understanding of the code as part of his trace, Briandan had not.

Data

Briandan read each section of the code, sometimes articulating a summary rather than each token of the syntax. For example, she said "*print*" to summarise the code "*system.out.println*". (Note that as Briandan was working in Java, the "print" statement was different to that shown in Figure 2.)

Briandan drew a line next to the code at line 5 in Figure 2 which prints `b`, to indicate the line that must be executed. She then determined that the first condition at line 1 in Figure 2 (i.e., `a > b)` must be true in order for `b` to print, and marked that condition with a dash. She said:

Briandan: *let's eliminate* [options] `if a` *... greater than* `b` *... we need* `a` *greater than ... no*

She then crossed out option (i) and proceeded to check the other three options in a similar way:

Briandan: *... is* `a` *greater than* `b` *yes ... that one* [option (iii)] *...* `a` *greater than* `b`*, no* [option (ii)] *...* `a` *greater than* `b` *... mm yes* [option (iv)] *... so we've got two options here*

By this stage she had eliminated options (i) and (ii) and placed a mark under options (iii) and (iv). However, she then changed tactic, and traced the values provided in options (iii) and (iv):

Briandan: *Now if* `b` *greater than* `c b` *... greater than* `c` *... no it's not ... so we're going to go to the* `else` *one ... so that would be possible* [option (iii)]

She then tested the final option as well in order to confirm her choice:

Briandan: *and other one* [option (iv)]*... yeah it would print* `c`

Briandan circled the correct option (iii).

To answer the EPE task in part (b) of Figure 2, Briandan read the code again then said:

Briandan: *hold on ... we printed the ... smallest number ... so assuming ... this didn't work right* [i.e., the condition (`b > c)` at line 2 in Figure 2 failed]*... if* `c` *greater than* `b` *its going to go up here* [line 5 in Figure 2]*... so printing the smallest number*

Analysis

That Briandan substituted some meaning for expressions as she read the code is evidence that she was processing the code, rather than simply reading it (i.e., when she said "*print*" in lieu of the code "*system.out.println*").

Briandan's initial approach to doing the tracing task was a short-cut elimination of two options based on the conditions that must be met. She did not test each answer option in a linear fashion. However, having eliminated two options by this approach, she then changed to tracing the remaining two options to determine their outcome. She did not make a transitive inference. Like Lance, Briandan found no need to refer to the code after line 6 in Figure 2.

In part (b), and unlike Lance, Briandan had to think further about the meaning of the code: she had not deduced the meaning of the code as she traced in part (a) in Figure 2. It hadn't occurred to her during the trace that the code would always print the smallest number. That is, she did not see that a transitive inference could be made about the variables.

3.2.2 Michael

Summary

Michael was given the C# exercise in Figure 2. His approach to the tracing task was different. He started with the first option and (forward) traced its values. When he noticed the `(a > b)` condition (at line 1 in Figure 2) was not met, he was able to eliminate both option (i) and option (ii). He then traced with the values from option (iii), and determined that `b`'s value would be printed. He chose option (iii) as the correct answer, and did not trace option (iv). Even though he accurately traced the code, he then had difficulty explaining what the code did autonomously. To explain the code, he required scaffolding from the interviewer.

Data

Michael read the code verbatim, including each output statement and punctuation ("*console dot writeline...*"). He then took the values in option (i) and started tracing:

Michael: *`a` equals to 1 `b` equals 2 `c` equals 3. 1 is more than 2 no. So this `if` statement would not run* [i.e., the condition `(a > b)` at line 1 in Figure 2 would evaluate to false].

Recognising that the `if (a > b)` block needed to execute for `b` to be printed, and therefore `a`'s value needed to be greater than `b`'s, he eliminated options (i) and (ii).

Michael: *so just left with the third option and fourth option*

He then looked at the next option, (iii):

Michael: *So `a` ... is more than `b` ... 2 is more than 1. You jump to next statement where 1 is more than 3 console dot writeline `c`. ... doesn't happen* [i.e., the condition `(b > c)` at line 2 in Figure 2 fails] *... so we write `b`. ... Yeah. ... so it's (iii)*

After requiring clarification that a line by line description was not required for part (b), Michael gave his first and incorrect explanation of the code:

Michael: *Display the values of `a` `b` and `c`?*

Interviewer: *Would it display all of them?*

Michael: *Not all of them. It depends on what starting values they have*

Interviewer: *Under what conditions would `a` print?*

Michael: *from the code ... as long as ... `a` is the smallest number ... comparison to ...*

At this point, Michael seemed to have figured out the purpose of the code. However, when asked under what conditions would `b` print he attempted to answer in terms of each of the conditions in the code that would need to be met:

Michael: *`b` would print when um ... `a` is smaller, `a` is larger than `b`, plus smaller than `c` ... and ... but ... oh wait wait wait I take that back. Uh ... `a` is ah ... more than `b` ... but less than `c` ... and `c` is larger than `b` ... ah `c` is ... larger than `b` yeah... to print `b`*

The interviewer asked a similar question about the code printing `c`, to which he gave a similar, preoperational answer. Then he was asked a more general question about the code:

Interviewer: *What can you say about it printing `a` or `b` or `c`? Is there anything in common?*

Michael: *Yeah all depends on values of `a` ... ah ... I'm very confused*

The interviewer then gave Michael a set of values for `a` `b` and `c` to trace: 10, 2, and 7 respectively.

Interviewer: *Which one will print?*

Michael: *... um 2 ...*

The interviewer then gave Michael another set: 2, 5, and 12 respectively:

Michael: *... `a` ... yeah `a` ... I've figured it out <laugh> as long as ... it's the smallest digit it will be printed*

Analysis

Unlike Briandan, Michael's inclination was to read every token of the code including output statements and punctuation verbatim ("*console dot writeline...*"). This is a remnant of the sensorimotor stage, where the language in the domain is still developing. At this stage, processing the meaning of many elements of code is necessary, and tends to be a time consuming process that requires a great deal of cognitive effort.

Michael traced the values in the options in a mechanical manner. He chose the first option, determined the outcome, eliminated another option based on the outcome of the first, then traced option (iii) before deciding it was the correct one.

Michael continually referred to the specific values of the variables while he traced. For example, "*1 is more than 2 no*", "`a` *... is more than* `b` *... 2 is more than 1*". Reliance on specific values to trace is consistent with preoperational behaviour.

Michael did not attempt to reason about the code as he traced. Unlike Lance, he was not building an understanding of the code's purpose while he traced it. His attempt to explain the code (part (b) in Figure 2) showed he had very little ability to reason about the code's purpose which is, again, indicative of someone at the preoperational stage. His explanation relied on inductive reasoning based on various input/output pairs. He itemised which conditions in the code needed to be met before the line to print `b` would be executed. As with any preoperational novice, Michael was preoccupied with evaluating individual statements rather than developing a more abstract "big picture".

It was only after several prompts by the interviewer, which lead Michael through additional traces with specific values, that he understood the purpose of the code. Until what appeared to be a "light bulb moment", Michael had used specific values when asked which variable would print: "*... um 2 ...*", but after his enlightenment, his responses became abstract: "*...* `a` *... yeah* `a` *...*". It was as if he had only just come to realise that the code's outcome, when expressed abstractly, was invariant.

3.2.3 Charlotte

Summary

Charlotte was given the C# exercise in Figure 2. Her method of tracing the code involved substituting specific values for each of the variables. She rewrote the code in specific rather than abstract terms. After an initial self-corrected error, she determined that the first condition (i.e., `(a > b)` at line 1 in Figure 2) must be met in order for the correct output statement to be executed, and then eliminated options (i) and (ii). She rewrote the code using values from option (iii), decided it was correct, but also checked (iv) before eliminating it. Charlotte's attempt to explain the purpose of the code was also heavily reliant on the use of specific values.

Data

Charlotte read the code including "*curly brace ... writeline ...*", and circled the "b" in the output statement (at line 5 in Figure 2) as a "note to self". She then proceeded to rewrite the code with specific values (from option (i)) substituted for the variables. She said:

Charlotte: *I'm going to write them out*

Charlotte then wrote what can be seen in Figure 3. Immediately she made an incorrect conclusion about option (i):

Charlotte: *so this one* [option (i)] *could be it*

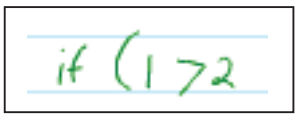


Figure 3 Charlotte's trace of option (i)

She said of option (iii):

Charlotte: *if 2 is greater than 1 ... this won't run so (iii) is already not an option*

At this point Charlotte paused (for 32 seconds) and silently read through the problem again. She then self-corrected:

Charlotte: *so* `a` *has to be greater than* `b` *for* `b` *to print*

She then eliminated options (i) and (ii) and marked option (iv) as a possibility. She traced option (iii) again by substituting specific values for the variables and wrote what appears in Figure 4.

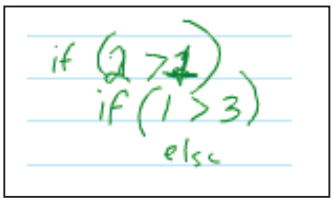


Figure 4 Charlotte's trace of option (iii)

Charlotte: *if 1 is greater than 3, which it's not, write* `c` *... else ...* `b`*, so it's (iii)*

After deciding this, she also checked (iv) "*just to be sure*":

Charlotte: *3 is greater than 2 which it is, if 2 is greater than 1 ...* `c` *... no its (iii)*

When Charlotte began part (b) of Figure 2, her initial "*gut instinct*" (as she described it) was incorrect:

Charlotte: *showing which number is ... I guess the biggest?*

However, she was not convinced that this was correct:

Charlotte: *Ok .. don't be lazy. 'cause I don't want to go through it all again*

But she did go through it again. This time, instead of rewriting the code with values, she wrote the specific values for option (iii) above each of the variables in the given code (see Figure 5).

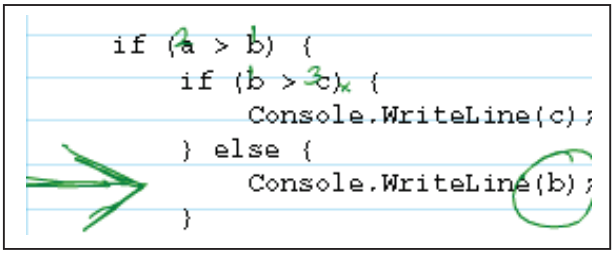


Figure 5 Charlotte's second trace of option (iii)

Then she asked herself:

Charlotte: *... what's so special about* `c`*?*

After a short time she concluded:

Charlotte: *In this case* [referring to the line of code to output `c`, line 3 of Figure 2]*,* `c` *would be the smallest number ... in this case* `b` [referring to line of code to output `b`, line 5 of Figure 2] *would be the smallest number ... in this case* `c` [referring to line of code to output `c`, line 3 of Figure 2] *... yeah, it's to find out what the smallest number is.*

Analysis

Like Michael, Charlotte's inclination was to read every token of the code including punctuation ("*curly brace*"), which suggests that she too is manifesting remnants of sensorimotor habits.

However, some of her behaviour is clearly preoperational. She successfully traced the code, but her reliance on specific values caused her to rewrite the code substituting a specific value for each of the variables. So rather than trying to reason about abstract code, she rewrote it in a language she understood: specific values. At other times when she was tracing or verifying the correctness of her answer, she wrote values above each of the variables.

Charlotte's initial attempt at reasoning about the code's purpose (i.e., that it finds the biggest) was intuitive, and surprisingly inaccurate given that she had previously concluded that "*...* `a` *has to be greater than* `b` *for* `b` *to print*". As novices at the preoperational stage attempt to reason about code, they tend to make guesses based on intuition, and those intuitions can be inconsistent.

After suspecting she was wrong, Charlotte actually considered *not* retracing. She thought better of it and admonished herself for being lazy. She then traced the code correctly. A sensorimotor novice finds any tracing task to be non-trivial and for that reason is reluctant to do so more than the minimum necessary.

In answer to part (b) of the task shown in Figure 2, it was only after, again, making extensive use of specific values and finding a pattern via inductive reasoning that she was able to make a conclusion about the purpose of the code.

Charlotte is manifesting behaviours of both the sensorimotor and preoperational stages. This behaviour fits with the overlapping waves theory as described in the previous section where her sensorimotor behaviours, although diminishing and no longer dominant, are still evident as she starts to reason at the preoperational stage.

4. Discussion

Preoperational novices are heavily reliant on specific values. They talk about code in terms of specific values and trace with specific values, to the extent of replacing variables with values as they trace code like Michael did: "*a ... is more than b ... 2 is more than 1. You jump to next statement where 1 is more than 3*". Similarly, Charlotte wrote "`if 1 > 2`". Novices at an early phase of the preoperational stage are keenly focused on using the knowledge accumulated in the sensorimotor stage (i.e., the semantics of programming constructs) to mechanically trace code. The ability to trace in abstract terms, like Lance did, is usually beyond the preoperational novice. Michael and Charlotte are working mostly at this preoperational level.

Also beyond the capacity of the preoperational novice is the ability to reason about the purpose of the code. Preoperational novices are preoccupied with the detail of a tracing task. They have developed the ability to determine the functional outcome of each line of code and trace to completion. However the mental effort of doing so exhausts them, which obscures from them the abstract purpose of the code. They are in effect, tracing blind.

There is a stark difference between the concrete operational behaviour of Lance and the preoperational behaviour of Michael and Charlotte. With the help of the think aloud sessions we have come to understand that what Lance was doing when he traced the code was something that neither Charlotte nor Michael did when they traced the code. He was reasoning about the parts of the code as he read and traced the code. Briandan also showed some evidence of processing the code as she read it, by summarising complicated output sequences simply as "*print*". The speed with which Briandan solved the EPE task, "*hold on ... we printed the ... smallest number ... so...*" is reasonable evidence that although she had not previously drawn this conclusion verbally, the process of tracing the code had provided some insight into the code's purpose. This behaviour exhibited by both Lance and, to a lesser extent, Briandan is indicative of the concrete operational stage.

Charlotte would have been awarded full marks for her answer if it had been provided in an exam. We doubt that Michael would have completed the EPE question in an exam, as he was unable to do so without intervention in his think aloud session. We suspect that many of the students who completed the in-class test are much like Charlotte or Michael. Their correct test answers belie the difficulty they had with the task. (We speculate that this difficulty might explain why some students can trace code, yet not be able to write similar code.)

It is interesting that none of the think aloud students referred to code after line 6. We could account for this in a number of ways. First, students may have assumed that any code we supplied would be "purposeful" code, which is indeed the case. Second, we could attribute their behaviour to inductive reasoning. That is, they drew conclusions about the purpose of the code based on input and output combinations. By backwards tracing (i.e., finding which conditions needed to be met in order to print the value of variable `b`), they saw no need to investigate the latter section of code as it was of no consequence to the outcome in this particular instance. For example, even if lines 8 and 10 in the code were swapped, it would *still* print the smallest value when the smallest value was stored in `b`.

If we assume that the reasoning processes of the students in the in-class test are consistent with the reasoning processes of the think aloud students, then we can make some inferences about the in-class test results. Students who could neither trace nor explain (see Col. 6 of Table 1) are exhibiting behaviours that are consistent with the sensorimotor stage. They manifest limited ability to reason

logically and abstractly about the code's purpose. Therefore their attempt at an EPE task is most likely a guess.

Students who traced the code correctly but then could not explain it (see Col. 2 of Table 1) fall into the preoperational category (at best). As we have discussed previously in this paper, students working at the preoperational level, like Michael and Charlotte, have developed the skills to trace code but as yet do not have the ability to reason abstractly about its purpose. As part (a) of the task (see Figure 2) was a multiple choice question, some students would have simply guessed the correct answer in the in-class test. In that case, and if the guess was because those students were unable to trace the code, then they are students who are at the sensorimotor stage.

That there are students who could not trace the code, but yet were able to explain it (see Col. 4 of Table 1), is an anomaly, for neo-Piagetian theory. (Jean Piaget referred to such anomalies as *decalage*.) However, those students are a very small proportion of the students. We suspect they had an accurate idea of the code's purpose, but merely made a careless mistake on part (a). It is less likely that they guessed the correct explanation, as this is more difficult to do for a short answer question than a multiple choice question.

Students who were able to complete both the tracing and explaining tasks successfully (see Col. 1 of Table 1), like Lance and Briandan, *may* be working at the concrete operational level. However, it is difficult to make a conclusion based on their answer alone. It is the *process* that identifies concrete operational reasoning, not the final answer. Charlotte's think aloud session in particular argues this point. It is at the concrete operational level we would *like* all of our students to be working, and it is certainly where most of our teaching and learning material is aimed. However, as we can see from our results, many of our students fall short of this level of cognitive development because they are still preoperational, and are not yet capable of working at a concrete level, with abstractions.

Our results support previous findings that explaining code is more difficult than tracing code. Neo-Piagetian theory offers an explanation of why that is so.

5. Conclusion

There are important pedagogical implications that can be drawn from this research. Many of our students are not reasoning at the concrete operational level required of the type of programming tasks we expect them to complete. If they cannot reason about code given to them, then they are probably incapable of writing similar code. From our data, about a third of our students are reasoning at the preoperational stage, so to them we may as well be talking in a foreign language when we pitch our teaching resources at the concrete operational level. Our preoperational students require exposure to reading and tracing tasks which are constituted from a minimal number of parts and which give them the freedom to use a less abstract level of reasoning. With sufficient practice, and with a slow increase in the sophistication of the code they read and trace, these students will eventually reach the concrete operational stage.

Neo-Piagetian theory offers a coherent framework for explaining our data. Readers might argue that our empirical results are not entirely new, and we have cited several other similar findings. However, our use of a neo-Piagetian framework to explain such data is new. Our use of neo-Piagetian theory also has methodological implications. Knowing that tracing code does not require concrete operational skills, students who can trace code accurately are not necessarily capable of tasks that require abstract reasoning, such as explain in plain English tasks, and also writing code.

Neo-Piagetian theory suggests interesting problems on which to study students. In this particular paper, we have used a problem intended to study transitive inference. In other papers, we have used problems intended to study other aspects of concrete operational reasoning, such as reversibility and conservation (Teague & Lister, 2014a, 2014b).

4. References

Atman, C. J., & Bursic, K. M. (1998). Verbal Protocol Analysis as a Method to Document Engineering Student Design Processes. *Journal of Engineering Education, 87*(2), 121-132.

Boom, J. (2004). Commentary on: Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 239-247.

da Rosa, S. (2007). *The Learning of Recursive Algorithms from a Psychogenetic Perspective*. Paper presented at the Psychology of Programming Interest Group (PPIG) 19th Annual Workshop 2007, Joensuu, Finland.

du Boulay, B. (1989). Some Difficulties of Learning to Program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the Novice Programmer* (pp. 283-300). Hillsdale, NJ: Lawrence Erlbaum.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology.

Feldman, D. H. (2004). Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 175-231.

Lewis, C. M. (2012). *The importance of students' attention to program state: a case study of debugging behavior*. Paper presented at the 9th Annual International Conference on International Computing Education Research (ICER 2012), Auckland, New Zealand.

Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011), Perth, WA.

Lister, R., Fidge, C., & Teague, D. (2009). *Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming*. Paper presented at the ITiCSE 09: Proceedings of the 14th annual conference on Innovation and technology in computer science education, Paris.

Lister, R., Simon, B., Thompson, E., Whalley, J., & Prasad, C. (2006). *Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy.* Paper presented at the Eleventh Annual Conference on Innovation Technology in Computer Science Education (ITiCSE'06), Bologna, Italy.

LiveScribe. (2014). Retrieved March 17, 2014, from https://www.smartpen.com.au/

Philpott, A., Robbins, P., & Whalley, J. (2007). *Accessing the Steps on the Road to Relational Thinking*. Paper presented at the 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07), Port Nelson, New Zeland.

Siegler, R. S. (1996). *Emerging Minds*. Oxford: Oxford University Press.

Simon, Lopez, M., Sutton, K., & Clear, T. (2009). *Surely We Must Learn to Read before We Learn to Write!* Paper presented at the 11th Australasian Computing Education Conference (ACE 2009), Wellington, New Zealand.

Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A., & Lister, R. (2012). *Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming*. Paper presented at the Australasian Association for Engineering Education Conference (AAEE 2012), Melbourne.

Teague, D., & Lister, R. (2014a). *Manifestations of Preoperational Reasoning on Similar Programming Tasks*. Paper presented at the Australasian Computing Education Conference (ACE 2014), Auckland, New Zealand.

Teague, D., & Lister, R. (2014b). *Programming: Reading, Writing and Reversing*. Paper presented at the ITiCSE '14, Uppsala, Sweden.

Whalley, J., Lister, R., Thompsopn, E., Clear, T., Robbins, P., Kumar, P., & Prasad, C. (2006). *An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies*. Paper presented at the 8th Australiasian Computing Education Conference, Hobart, Australia.

Exploring Problem Solving Paths in a Java Programming Course

Roya Hosseini
Intelligent Systems Program
University of Pittsburgh
roh38@pitt.edu

Arto Vihavainen
Dep. of Computer Science
University of Helsinki
avihavai@cs.helsinki.fi

Peter Brusilovsky
University of Pittsburgh
135 North Bellefield Ave.
peterb@pitt.edu

Abstract

Assessment of students' programming submissions has been the focus of interest in many studies. Although the final submissions capture the whole program, they often tell very little about how it was developed. In this paper, we are able to look at intermediate programming steps using a unique dataset that captures a series of snapshots showing how students developed their program over time. We assessed each of these intermediate steps and performed a fine-grained concept-based analysis on each step to identify the most common programming paths. Analysis of results showed that most of the students tend to incrementally build the program and improve its correctness. This finding provides us with evidence that intermediate programming steps are important, and need to be taken into account for not only improving user modelling in educational programming systems, but also for providing better feedback to students.

Keywords: POP-II.C. Working Practices; POP-III.B. Java; POP-VI.E. Computer Science Education Research

1. Introduction

Over the last 10 years, the popularity and the use of systems for automated assessment of programming assignments has been growing constantly. From the pioneer Ceilidh system (Benford et al. 1993) to the modern popular Web-CAT (Edwards et al. 2008) system, these systems offer instructors and students some highly valuable functionality - the ability to assess the correctness and other important features of student programs without investing a large volume of manual labour. These systems introduced a number of changes in the way programming is usually taught. They allowed instructors to increase the number of programming assignments in a regular course and offer multiple submission attempts for each assignment (Douce et al. 2005). Additionally, they enabled teaching assistants to focus on more creative ways in assisting the students. As a result of these changes, the automatic assessment systems remarkably increased students' chances to receive feedback on their work.

There is, however, another important impact of automatic assessment systems that is being discussed much more rarely: their ability to increase our understanding of how humans solve programming problems. Course by course, automatic assessment systems accumulate a large number of program submissions. These submissions open a window to the students' problem solving process on both the personal and community level. On the *personal level*, the analysis of single student program submissions could reveal how the student progresses to the correct solution through several incorrect ones and how her knowledge grows from assignment to assignment. On the *community level*, the analysis of submissions for a single problem could reveal multiple correct and incorrect ways to solve the same problem.

Active work on community-level analysis has already started, to a large extent propelled by the availability of a very large volume of problem solutions collected by several programming MOOC courses that apply automatic assessment. Two interesting papers demonstrate how the MOOC data could be used to analyse the landscape of student problem solving solutions for individual problems (Huang et al. 2013, Piech et al. 2012), while another work shows how this data could be used to build an intelligent scaffolding system (Rivers and Koedinger 2013).

In our work we would like to contribute to the analysis of program submission on the *personal level*. This work is enabled by a unique set of data from University of Helsinki programming classes. Unlike other automatic assessment systems, the Helsinki system collected not just final program states submitted for grading, but also many intermediate states. It offered a much broader window to examine the progress of individual students on their way to problem solutions. In addition, we applied a unique concept-based analysis approach to examine and compare student partial and complete solutions.

Armed with a unique dataset and a novel analysis approach, we examined a large body of individual student paths to success. The remaining part of the paper is structured as follows. The next section reviews a few streams of related work. After that we introduce the key assets - the dataset and the concept-based analysis approach. The following sections present the details of our analysis. We conclude with an overview of the performed work and plans for future work.

2. Related Work

Understanding and identifying the various challenges novice programmers face has been in the center of interest of computer science educators for decades. Work on identifying traits that indicate tendency toward being a successful programmer dates back to the 1950s (Rowan 1957), where psychological tests were administered to identify people with an ability to program. Since the 1950s, dozens of different factors have been investigated. This work included the creation of various psychological tests (Evans and Simkin 1989, Tukiainen and Mönkkönen 2002) and investigation of the effect of factors such as mathematical background (White and Sivitanides 2003), spatial and visual reasoning (Fincher et al. 2006), motivation and comfort-level (Bergin and Reilly 2005), learning styles (Thomas et al. 2002), as well as the consistency of students' internal mental models (Bornat and Dehnadi 2008) and abstraction ability (Bennedssen and Caspersen, 2008). While many of these factors correlate with success within the study contexts, more recent research suggests that many of these might be context-specific, and urges researchers to look at data-driven approaches to investigate e.g., the students' ability to solve programming problems and errors from logs gathered from programming environments (Watson et al. 2014). Analysis of submission and snapshot streams in terms of student compilation behaviour and time usage has been investigated in, for instance, Jadud (2005) and Vee (2006).

Perhaps the most known work on exploring novice problem-solving behaviour is by Perkins et al., whose work classifies students as "stoppers," "movers" or "tinkerers" based on the strategy they choose when facing a problem (Perkins et al. 1986). The stoppers freeze when faced with a problem for which they see no solution, movers gradually work towards a correct solution, and tinkerers "try to solve a programming problem by writing some code and then making small changes in the hopes of getting it to work." Some of the reasons behind the different student types are related to the strategies that students apply as they seek to understand the problem; some students approach the programming problems using a "line by line" approach, where they often fail to see larger connections between the programming constructs (Winslow 1996), while others depend heavily on the keywords provided in the assignments that mistakenly led to incorrect or inefficient solutions (Ginat 2003).

The importance of students' constant and meaningful effort has been acknowledged by programming educators, who have invested a considerable amount of research in tools and systems that support students within the context they are learning, providing, e.g,. additional hints based on the students' progress. Such tools include systems that provide feedback for students' as they are programming (Vihavainen et al. 2013), systems that can automatically generate feedback for struggling students (Rivers and Koedinger 2013), and systems that seek to identify the situations when support should be provided (Mitchell et al. 2013).

3. The Dataset

The dataset used in this study comes from a six-week introductory programming course (5 ECTS) held at the University of Helsinki during fall 2012. The introductory programming course is the first

course that students majoring in Computer Science take; no previous programming experience is required. Half of the course is devoted to learning elementary procedural programming (input, output, conditional statements, loops, methods, and working with lists), while the latter part is an introduction to object-oriented programming.

The course is taught using the Extreme Apprenticeship method (Vihavainen et al. 2011) that emphasizes the use of best programming practices, constant practise, and bi-directional feedback between the student and instructor. Constant practice is made possible with the use of a large amount of programming assignments; the students work on over 100 assignments, from which some have been split into multiple steps (a total of 170 tasks) – over 30 tasks are completed during the first week. The bi-directional feedback is facilitated both in aided computer labs, where students work on course assignments under scaffolding from course instructors and peers, as well as by a plugin called Test My Code (TMC) (Vihavainen et al. 2013), which is installed to the NetBeans programming environment that is used throughout the course. TMC provides guidance in the form of textual messages that are shown as students run assignment-specific tests; the detail of the guidance depends on the assignment that the student is working on, as well as on the goal of the assignment, i.e., whether the goal is to introduce new concepts (more guidance) or to increase programming routine (less guidance). In addition to guidance, the TMC-plugin gathers snapshots from the students' programming process, as long as students have given consent. Snapshots (time, code changes) are recorded every time the student saves her code, runs her code, runs tests on the code, or submits the code for grading.

The dataset we used for the study has 101 students (65 male, 36 female), with the median age of 21 for male participants and 22 for female participants. Students who have either not given consent for the use of their data or have disabled the data gathering during some part of the course have been excluded from the dataset, as have students who have not participated in the very first week of the course. Additional filtering has been performed to remove subsequent identical snapshots (typically events where students have first saved their program and then executed it, or executed the program and then submitted it for grading), resulting with 63701 different snapshots (avg. 631 per student). Finally, as snapshots are related to assignment that are all automatically assessed using assessment-specific unit tests, each snapshot has been marked with correctness of the student's solution. The correctness is a value from 0 to 1, which describes the fraction of assignment specific tests this snapshot passes.

When comparing the dataset with other existing datasets such as the dataset from the Blackbox-project (Brown et al. 2014), the dataset at our disposal is unique due to (1) the scaffolding that students received during the course, (2) it provides a six-week lens on the struggles that novices face when learning to program in a course emphasizing programming activities, (3) the assignment that the student is working on is known for each snapshot, making the learning objectives visible for researchers, (4) the test results for each snapshot is available, providing a measure of success, and (5) students have worked within an "industry-strength" programming environment from the start.

4. Approach

The main goal of this paper is to study students' programming behaviour. The approaches we use in this paper are based on two unique aspects. First, we use an enhanced trace of student program construction activity. While existing studies focus on the analysis of students' final submissions, our dataset provides us with many intermediate steps which give us more data on student paths from the provided program skeleton to the correct solutions. Second, when analysing student progress and comparing the state of the code in consecutive steps, we use a deeper-level conceptual analysis of submitted solutions. Existing work has looked at each submitted program as a text and either examined student progress by observing how students add or remove lines of code (Rivers and Koedinger 2013) or attempted to find syntactic and functional similarity between students' submissions (Huang et al. 2013). We, on the other hand, are interested in the conceptual structure of submitted programs. To determine which programming concepts have been used in a specific solution, we use a concept extraction tool called JavaParser (Hosseini and Brusilovsky 2013).

```
import java.util.Scanner;
public class BiggerNumber {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("Type a number:  ");
      int firstNumber = Integer.parseInt(input.nextLine());
      System.out.println("Type another number:  ");
      int secondNumber = Integer.parseInt(input.nextLine());
      if (firstNumber > secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + firstNumber);
      if (firstNumber < secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + secondNumber);
      else
         System.out.println("\nNumbers were equal: ");
   }
}
```

(a)

```
import java.util.Scanner;
public class BiggerNumber {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("Type a number:  ");
      int firstNumber = Integer.parseInt(input.nextLine());
      System.out.println("Type another number:  ");
      int secondNumber = Integer.parseInt(input.nextLine());
      if (firstNumber > secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + firstNumber);
      else if (firstNumber < secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + secondNumber);
      else
         System.out.println("\nNumbers were equal: ");
   }
}
```

(b)

Figure 1 - Implementation of the 'Bigger Number' program: (a) first snapshot, (b) second snapshot

The JavaParser tool extracts a list of ontological concepts from source code using a Java ontology developed by PAWS laboratory1; concepts have been extracted for each snapshot.

Table 1 presents the set of concepts that are extracted by Java Parser for an assignment called 'Bigger Number' at two consecutive snapshots of a single user (Figure 1a and Figure 1b). In this assignment, the goal of the student was to create an application that asks for two numbers and then prints out the larger one of the two. More extensive description on this problem is available on the course website2. This program is tested using 3 tests that verify that output is right when the first number is smaller than the second (Test 1); the output is right when the second number is smaller than the first (Test 2); and the student does not print anything unnecessary (Test 3).

Figure 1 shows that the student first wrote a program in Figure 1a that passes Test 1 and Test 2 when the first number is smaller than the second or vice versa. However it does not pass Test 3 since it prints additional information when the second number is smaller than the first one. After receiving this feedback, the student expands the code by adding the 'else if' statement as shown in Figure 1b. Now the program also passes Test 3 since it does not print any unnecessary outputs when the numbers differ.

Snapshot	Extracted Concepts
(a)	ActualMethodParameter, MethodDefinition, ObjectCreationStatement, ObjectMethodInvocation, PublicClassSpecifier, PublicMethodSpecifier, StaticMethodSpecifier, StringAddition, StringDataType, StringLiteral, LessExpression, java.lang.System.out.println, java.lang.System.out.print, ClassDefinition, ConstructorCall, FormalMethodParameter, GreaterExpression, **IfElseStatement**, **IfStatement**, ImportStatement, IntDataType, java.lang.Integer.parseInt, VoidDataType
(b)	ActualMethodParameter, MethodDefinition, ObjectCreationStatement, ObjectMethodInvocation, PublicClassSpecifier, PublicMethodSpecifier, StaticMethodSpecifier, StringAddition, StringDataType, StringLiteral, LessExpression, java.lang.System.out.println, java.lang.System.out.print, ClassDefinition, ConstructorCall, FormalMethodParameter, GreaterExpression, **IfElseIfStatement**, **IfElseStatement**, ImportStatement, IntDataType, java.lang.Integer.parseInt, VoidDataType

Table 1 - Distinct concepts extracted by JavaParser for snapshot (a) and (b) of the program shown in Figure 1. Differences are highlighted in bold.

1 http://www.sis.pitt.edu/~paws/ont/java.owl

2 http://mooc.cs.helsinki.fi/programming-part1/material-2013/week-1#e11

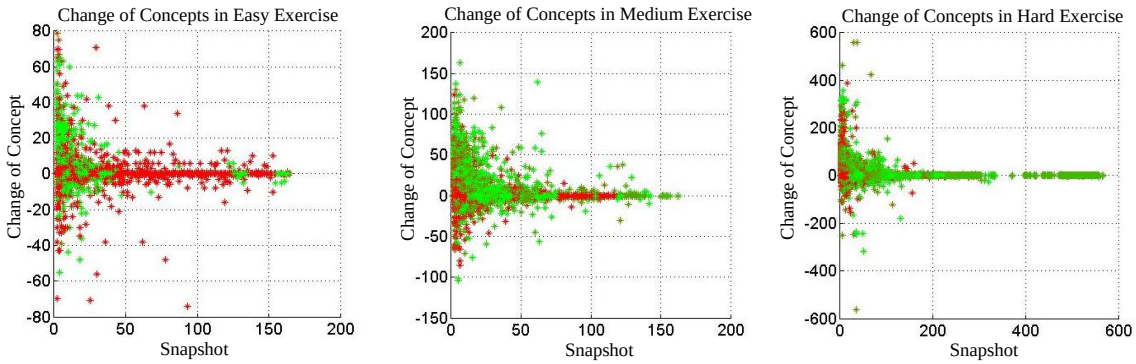
Using this tool we can examine conceptual differences between consecutive submissions – i.e., observe which concepts (rather than lines) were added or removed on each step. We can also examine how these changes were associated with improving or decreasing the correctness of program. Our original assumption about student behaviour is that students seek to develop their programs incrementally. We assume that a student develops a program in a sequence of steps adding components to its functionality on every step while gradually improving the correctness (i.e., passing more and more tests). For example, a student can start with a provided program skeleton which passes one test, then enhance the code so that it passes two tests, add more functionality to have it pass three tests, and so on. To check this assumption we start our problem solving analysis process with the global analysis that looks into common programming patterns and then followed with several other analysis approaches.

5. Analysis of Programming Paths

5.1. Global Analysis

Given the dataset that has all intermediate snapshots of users indexed with sets of concepts, we have everything ready for finding global patterns in students programming steps (snapshots). To examine our incremental building hypothesis, we started by examining the change of concepts in each snapshot from its most recent successful snapshot. We defined a successful snapshot as a program state that has the correctness either greater than or equal to the greatest previous correctness. For simplicity the concept change was measured as numerical difference between the number of concepts in two snapshots. Since student-programming behaviour might depend on problem difficulty, we examined separately the change of concepts per three levels of exercise complexity: easy, medium, and hard. The difficulty level of exercise is determined by averaging students' responses over the question "*How difficult was this exercise on a scale from 1 to 5?"* which was asked after each exercise. An easy exercise has the average difficulty level of [1.00,2.00); a medium exercise has the average difficulty level of [2.00,2.50); and a hard exercise has the average difficulty level of [2.50,5.00].

Figure 2 shows the change of concepts for all easy, medium, and hard exercises, respectively. It should be mentioned that each figure only shows change of concepts from the most recent successful snapshots. Each data point represents one snapshot taken from one user in one exercise. The horizontal position of the point represents the number of this snapshot in the sequence of snapshots produced by this user for this exercise. The vertical position of the point indicates the change in the number of concepts from the previous successful snapshot. If a point is located above the 0 axe, the corresponding snapshot added concepts to the previous snapshot (this is what we expected). If it is located below the 0 axe, concepts were removed. The colour of each point indicates the correctness of the user's program at that snapshot using red-green gradient to visualize [0-1] interval. Light green is showing the highest correctness (1) where all tests are passed and light red showing no correctness (0) where none of the tests are passed. The summary of the same data is presented in Table 2.



(a) (b) (c)

Figure 2 - Change of concepts from the most recent successful snapshot in all (a) easy, (b) medium, and (c) hard exercises for all students.

	1st Quartile			2nd Quartile			3rd Quartile			4th Quartile		
Exercise	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**
Easy	13440 (2482/894)	2.82 ±8.04	0.49 ±0.46	462 (132,60)	2.86 ±5.84	0.09 ±0.26	223 (37,24)	1.69 ±6.04	0.05 ±0.20	107 (21,16)	1.29 ±2.39	0.24 ±0.43
Medium	20449 (6071/1526)	5.74 ±13.62	0.46 ±0.41	1182 (363,210)	8.26 ±19.93	0.45 ±0.38	287 (56,56)	2.47 ±6.70	0.46 ±0.33	74 (24,11)	2.23 ±5.42	0.70 ±0.18
Hard	31229 (10882/3935)	11.26 ±35.18	0.47 ±0.35	636 (319,65)	3.78 ±10.59	0.50 ±0.28	193 (45,44)	2.30 ±6.13	0.65 ±0.17	142 (24,14)	1.05 ±3.73	0.70 ±0.09

Table 2 - The scale of program changes at different stages of program development. Total is the total number of points in each quartile; +/-: the number of positive/negative changes; Avg,SD: the average and standard deviation of change respectively; Cor,SD: is the average and standard deviation of correctness respectively

The analysis of the data presented on Figure 2 and Table 2 leads to two important observations. First, we can see that the amount of conceptual changes from snapshot to snapshot is relatively high at the beginning of students' work on each exercise and then drops rapidly, especially for hard problems. It shows that on average significant changes are introduced to the constructed program relatively early within the first quartile of its development. To confirm this observation, we used Spearman's Rank Order correlation to determine the relationship between number of changed concepts and snapshot number for 567 snapshots, i.e. the maximum number of snapshots in student-exercise pairs. Each snapshot, contained the sum of absolute changes of concepts in the snapshot with this number for all student in all exercises. The analysis confirmed the presence of strong, statistically significant ($r_s(565) = -0.82$, $p < .000$) negative correlation between changed concepts and snapshot number.

The second observation is about the nature of changes. As the data shows, all development changes include large changes in both directions – both adding and removing concepts. It shows that our original hypothesis about incremental program expansion is not correct – there are a lot of cases when concepts (frequently many of them) are removed between snapshots. While the figure provides an impression that snapshots associated with removal tend to be less successful (more "red" in the lower part of the figure), there is still a sufficient number of successful concept-removal steps. To examine the relationship between the scale of removal and the success of result, we ran Spearman's Rank Order correlation between *the number of removed concepts* and *correctness* in 4058 snapshots which have removed concepts. The results confirmed the observation: there is a very weak, negative correlation between removed concepts and correctness, however, it is not statistically significant ($r_s(4056) = -0.03$, $p=.071$). These diagrams, however, integrate the work of many students over many exercises, so it doesn't explain why and when so many reduction steps appeared. Could it be that some specific group of exercises encourages this concept-removal approach while the remaining exercises are generally solved by progressively adding concepts? Could it be that some specific categories of students construct programs by throwing a lot of concepts in and then gradually removing? To check this hypothesis we examined adding/removal patterns for different exercises and groups of students.

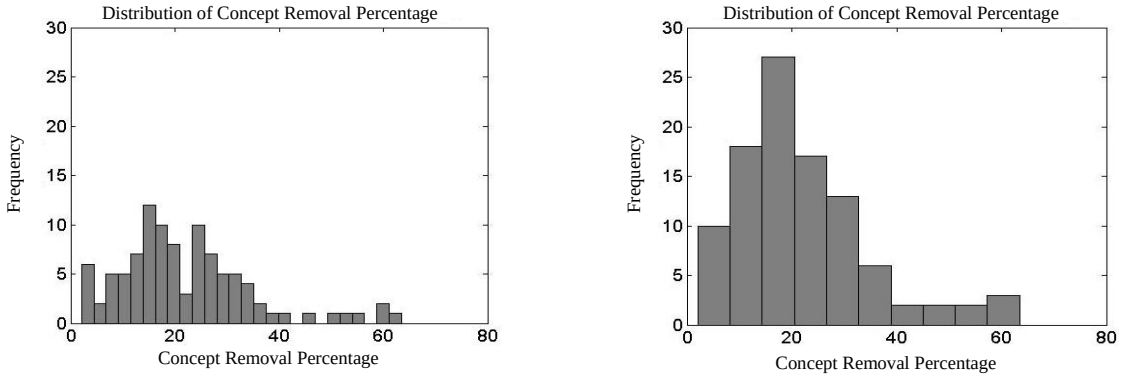


Figure 3 - The distribution of concept-removing steps over exercises. Bins represent group of exercises with same fraction of removal steps, the height of each bar represents the number of exercises in each bin. The left and right diagrams have different number of bins.

Figure 3 shows the distribution of concept-removal behaviour over problems. As the data shows some tasks do provoke larger fraction of removing actions, however, there are very few tasks on the extreme sides (very small or relatively large fraction of removals) and there are no problems with less than 3% or more than 64% of removals. Overall, the distribution shows that the majority of problems have very similar fraction of removal steps. This data hints that the frequency of removal steps is more likely caused by some student-related factor, not by the nature of specific exercises.

Since we have no personal data about students, the only student-related factor we can further examine was the level of programming skills that could be approximated by analysing the individual speed in developing the code. The presence of green dots in the first quartile indicates that some good number of students are able to pass all tests relatively fast, while the presence of the long tail (very long for hard exercises) indicates that a considerable number of students struggle to achieve full correctness for a relatively long time. That is, slower students tend to develop the program in more snapshots while faster students write program in fewer snapshots. To further investigate whether students' programming speed leads to different programming behaviours, we clustered students based on their number of snapshots on their path to success. For each exercise, we separated students into three different groups by comparing the number of their snapshots with the median of students' snapshots in that exercise. We then classified students with snapshots less than the median to be in 'cluster 1' and greater than median to be in 'cluster 2'. We ignored students with the number of snapshots exactly equal to median to improve the cluster separation.

Figure 4 shows changes of concepts per all easy, medium, and hard exercises for all students in the two clusters. Overall, this figure exhibits a similar pattern to the one observed in Figure 2. We see larger-scale changes in the beginning of program development are replaced by small changes in the middle and end of the process. We also see that students on all skill levels working with all problem difficulty levels apply both concept addition and concept reduction when constructing a program. This hints that the patterns of program construction behaviour are not likely to be defined by skill levels and that we need to examine the behaviour of individual students to understand the nature of pattern differences.

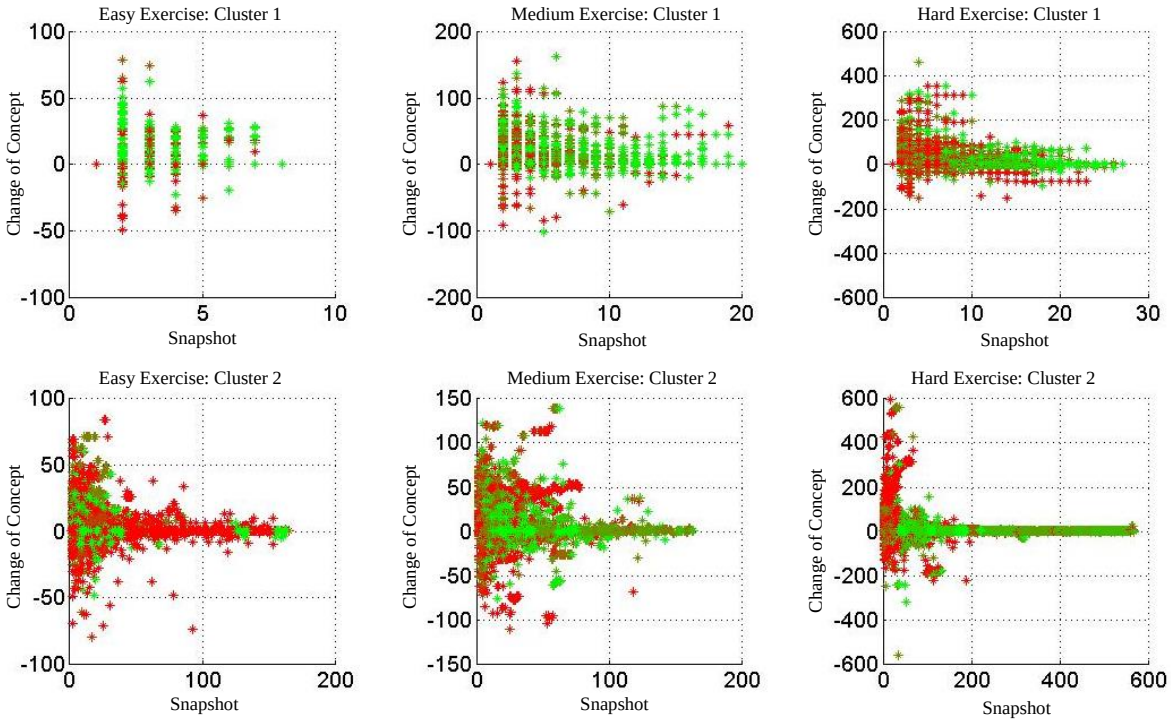


Figure 4 - Change of concepts and correctness for students with different programming speeds. Cluster 1 represents students who need less than the median number of snapshots and Cluster 2 represents students who need more than the median number of snapshots to develop the program.

5. 2. Analysis of programming behaviour on individual level

At the final analysis step, we examined programming behaviour on the finest available level exploring the behaviour of individual students on a set of selected assignments. The analysis confirmed that most interesting differences in problem solving patterns could be observed on the individual level where it likely reflects differences in personal problem solving styles. Figure 5 shows several interesting behaviour patterns we discovered during the analysis. We call the students exhibiting these specific patterns Builders, Massagers, Reducers and Strugglers. This classification is inspired by the work of Perkins et al. (1986), who categorized students as "stoppers", "movers" and "tinkerers;" however, in our dataset, the students rarely "stop" or give up on the assignment, and even the tinkering seems to be goal-oriented. Among these patterns, Builders that roughly correspond to "movers" behave exactly as expected; they gradually add concepts to the solution while increasing of correctness of the solution in each step (Figure 5a). In many cases we observe that students who generally follow a building pattern have long streaks when they are trying to get to the next level of correctness by doing small code changes without adding or removing concepts in the hope of getting it to work (Figure 5b). We call these streaks as code massaging and students who have these streaks as Massagers.

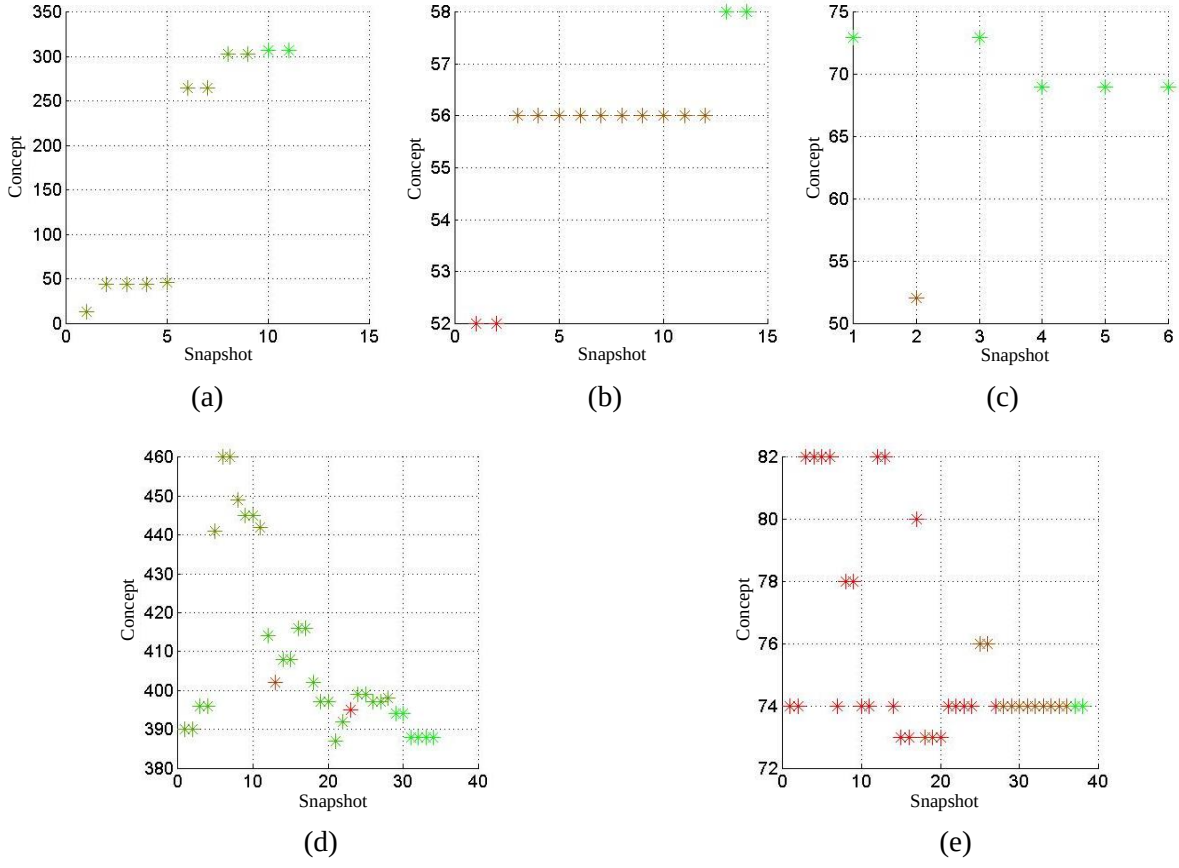


Figure 5 - Patterns of individual programming behaviour for different students. The X-axis represents the snapshot number and the Y-axis represents the number of concepts in the snapshot. The colour of each point ranges from green to red representing increase or decrease in the correctness compared to the previous successful snapshot, respectively. (a) A Builder who solves a hard problem incrementally by adding concepts and increasing the correctness; (b) A Massager who has long streaks of small program changes without changing concepts or correctness levels; (c) A Reducer who tries to reduce the concepts after reaching the complete correctness in a medium exercise; (d) A Reducer who reduces concepts and gradually increases correctness in a hard exercise (e); A Struggler who struggles with the program failing to pass any tests for a long time.

We can also observe behaviour opposite to building when students remove concepts while maintaining or reducing the correctness level. In many cases reducing steps could be found at the end of regular building stage where it apparently represents the attempts to make the code more efficient (Figure 5c). However, we also identified some fraction of students who use concept reduction as the main problem solving approach. These students start by throwing in large amounts of code and then gradually work by reducing the set concepts until reaching the correct solutions (Figure 5d). Massagers and Reducers could be probably considered as a mixture of Perkins' movers and tinkerers. Finally, we identified a considerable fraction of students we call Strugglers (Figure 5e). These students spend considerable amounts of time to pass the first correctness test. They do all kinds of code changes, but probably have too little knowledge to get the code working. We probably could consider these students as a mix of tinkerers and stoppers; they may freeze on a problem (not going towards the "correct" solution) for a long while, but through experimentation and movement typically finally end up getting the solution right.

The observed global patterns provide interesting insight into the variety of student problem behaviour, yet it is very hard to use them in the observed form to classify students in clear-cut categories and analyse how frequently these patterns can be found in student behaviour. To recognize students with different kinds of behaviour we decided to perform micro-analysis of student behaviour. Starting with the main patterns of programming behaviours we observed, we decided to explore the frequency of such patterns inside programming steps of each user in each exercise. Instead of classifying the whole student path as Builder, Massager, Reducer, or Struggler, we started by classifying each problem solving step with these labels based on the change of concept and correctness in each snapshot. Table 3 lists the labels used for the labeling process. Next, we applied sequential pattern mining to identify the most common problem solving patterns among the 8458 students-exercise pairs that we had in the dataset. Specifically, we used software called PEX-SPAM which is an extension of SPAM algorithm (Ayres et al. 2002) that supports pattern mining with user-specified constraints such as minimum and maximum gap for output feature patterns. The tool was originally developed for biological applications but it can be easily employed to extract frequent patterns in other domains, too (Ho et al. 2005). Table 4 summarizes the common patterns obtained by PEX-SPAM with the support (the probability that the pattern occurs in the data) greater than 0.05.

The table shows interesting results related to the frequency of specific patterns. The first important observation is the dominance of the Builder pattern. In particular, the most common retrieved pair is 'BB' with support of 0.304 indicating that this pattern occurs in about ⅓ of the student-exercise pairs. This indicates that our original assumption of regular program building is not really that far from reality. While only a fraction of students could be classified as straightforward Builders, streaks of incremental building represent most typical behaviour. Another frequent kind of pattern is struggling sequences that may or may not end with the building step. It is also interesting to observe that reducing steps are rarely found inside frequent sequences.

To further investigate the most dominant programming pattern inside each of the 8458 students-exercise pairs, we grouped each of these pairs based on their most frequent pattern(s). Table 5 shows the results of the grouping and lists the frequency and percentage of each group. According to this table, about 77.63% of the students are Builders who tend to add concepts and increase the correctness incrementally.

Correctness\Concepts	**Same**	**Increase**	**Decrease**
Zero	Struggler	Struggler	Struggler
Decrease	Struggler	Struggler	Struggler
Increase	Builder	Builder	Reducer
Same	Massager	Builder	Reducer

Table 3 - Labeling sequence of students' snapshots based on change of concept and correctness

Pattern	Support	Pattern	Support
BB	0.304	SSB	0.123
SB	0.281	SSSS	0.088
BS	0.215	BSS	0.083
SS	0.199	SSSB	0.07
SSS	0.128	SSSSS	0.067
BR	0.127	BBBB	0.063
BBB	0.123	BBS	0.058

B: Builder, S: Struggler, R: Reducer

Table 4 - Common programming patterns with support greater than 0.05.

Group	Frequency	Percentage	Group	Frequency	Percentage
B	6566	77.63	BM	60	0.71
BS	1084	12.82	BMS	26	0.31
BR	309	3.65	BRMS	11	0.13
R	173	2.05	RS	11	0.13
BRM	135	1.60	M	11	0.12
BRS	70	0.83	MS	2	0.02

B: Builder, S: Struggler, R: Reducer, M: Massager

Table 5 - Details of dominant group(s) in student-exercises pairs

6. Discussion & Conclusion

To explore behaviour patterns found along student problem paths, we performed macro- and micro-level analysis of student program construction behaviour. Contrary to our original hypothesis of student behaviour as incremental program building, by adding more concepts and passing more tests, the macro view of student behaviour indicated a large fraction of concept reduction steps. Further analysis demonstrated that both building and reduction steps happen in all stages of program development and across levels of problem difficulty and student mastery. The analysis of program building patterns on the micro-level demonstrated that while the original building hypothesis is not strictly correct, it does describe the dominant behaviour of a large fraction of students. Our data indicate that for the majority of students (77.63%) the dominant pattern is building, i.e. incrementally enhancing the program while passing more and more tests. At the same time, both individual pattern analysis and micro-level pattern mining indicated that there are a number of students who do it in a very different way. Few students tend to reduce the concepts and increase the correctness (2.05%) or they might have long streaks of small program changes without changing concepts or correctness level (0.12%). There are also some students who show different programming behaviour, showing two or more patterns (20.2%), hinting that students' behaviour can change over time. Apparently, the original hypothesis does not work for everyone, but it does hold for a considerable number of students. Therefore, we have now enough evidence to conclude that there exists a meaningful expansion programming that implies starting with a small program, and then increasing its correctness by incrementally adding concepts and passing more and more tests in each of the programming steps.

In future work, we would like to see to what extent the introduced programming behaviours are stable over different datasets and even in different programming domains. It would be also interesting to see how information such as age, gender and math performance, as well as many of the traits that have been previously used to predict programming aptitude influences programming behaviour. We believe

having such step-wise analysis of student programming could enable us to have more advanced and accurate user modelling.

8. References

Ayres, J., Flannick, J., Gehrke, J., & Yiu, T. (2002, July). Sequential pattern mining using a bitmap representation. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 429-435). ACM.

Benford, S., Burke, E., & Foxley, E. (1993). Learning to construct quality software with the Ceilidh system. Software Quality Journal, 2(3), 177-197.

Bennedssen, J., & Caspersen, M. E. (2008, September). Abstraction ability as an indicator of success for learning computing science?. In Proceedings of the Fourth international Workshop on Computing Education Research (pp. 15-26). ACM.

Bergin, S., & Reilly, R. (2005, June). The influence of motivation and comfort-level on learning to program. In Proc. 17th Workshop of the Psychology of Programming Interest Group (pp. 293-304).

Bornat, R., & Dehnadi, S. (2008, January). Mental models, consistency and programming aptitude. In Proceedings of the tenth conference on Australasian computing education-Volume 78 (pp. 53-61). Australian Computer Society, Inc.

Brown, N. C., Kölling, M., McCall, D., & Utting, I. (2014, March). Blackbox: A Large Scale Repository of Novice Programmers' Activity. In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 223-228). ACM.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. Journal on Educational Resources in Computing (JERIC), 5(3), 4.

Edwards, S. H., & Perez-Quinones, M. A. (2008, June). Web-CAT: automatically grading programming assignments. In ACM SIGCSE Bulletin(Vol. 40, No. 3, pp. 328-328). ACM.

Evans, G. E., & Simkin, M. G. (1989). What best predicts computer proficiency?. Communications of the ACM, 32(11), 1322-1327.

Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., de Raadt, M., ... & Tutty, J. (2006, January). Predictors of success in a first programming course. In Proceedings of the 8th Australasian Conference on Computing Education-Volume 52 (pp. 189-196). Australian Computer Society, Inc..

Ginat, D. (2003, June). The novice programmers' syndrome of design-by-keyword. In ACM SIGCSE Bulletin (Vol. 35, No. 3, pp. 154-157). ACM.

Ho, J., Lukov, L., & Chawla, S. (2005). Sequential pattern mining with constraints on large protein databases. In Proceedings of the 12th International Conference on Management of Data (COMAD) (pp. 89-100).

Hosseini, R., & Brusilovsky, P. (2013, June). JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (p. 60).

Huang, J., Piech, C., Nguyen, A., & Guibas, L. (2013, June). Syntactic and functional variability of a million code submissions in a machine learning mooc. In AIED 2013 Workshops Proceedings Volume (p. 25).

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. Computer Science Education, 15(1), 25-40.

Mitchell, C. M., Boyer, K. E., & Lester, J. C. (2013, June). When to Intervene: Toward a Markov Decision Process Dialogue Policy for Computer Science Tutoring. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (p. 40).

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012, February). Modeling how students learn to program. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 153-160). ACM.

Rivers, K., & Koedinger, K. R. (2013, June). Automatic generation of programming feedback: A data-driven approach. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (p. 50).

Rowan, T. C. (1957). Psychological Tests and Selection of Computer Programmers. Journal of the Association for Computing Machinery, 4, 348-353.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002, February). Learning styles and performance in the introductory programming sequence. In ACM SIGCSE Bulletin (Vol. 34, No. 1, pp. 33-37). ACM.

Tukiainen, M., & Mönkkönen, E. (2002). Programming aptitude testing as a prediction of learning to program. In Proc. 14th Workshop of the Psychology of Programming Interest Group (pp. 45-57).

Vee, M. H. N. C., Meyer, B., & Mannock, K. L. (2006). Understanding novice errors and error paths in object-oriented programming through log analysis. In Proceedings of Workshop on Educational Data Mining at the 8th International Conference on Intelligent Tutoring Systems (ITS 2006) (pp. 13-20).

Vihavainen, A., Paksula, M., & Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. In Proceedings of the 42nd ACM technical symposium on Computer science education (pp. 93-98). ACM.

Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013, July). Scaffolding students' learning using test my code. In Proceedings of the 18th ACM conference on Innovation and technology in computer science education(pp. 117-122). ACM.

Watson, C., Li, F. W., & Godwin, J. L. (2014, March). No tests required: comparing traditional and dynamic predictors of programming success. In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 469-474). ACM.

White, G., & Sivitanides, M. (2003). An empirical investigation of the relationship between success in mathematics and visual programming courses. Journal of Information Systems Education, 14(4), 409-416.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. ACM SIGCSE Bulletin, 28(3), 17-22.

Falling Behind Early and Staying Behind When Learning to Program

Alireza Ahadi & Raymond Lister

University of Technology, Sydney
Australia
raymond.lister@uts.edu.au

Donna Teague

Queensland University of Technology
Australia
d.teague@qut.edu.au

Keywords: *POP-I.B. barriers to programming, POP-II.A. novices, POP-V.A. Neo-Piagetian, POP-VI.E. computer science education research*

Abstract

We have performed a study of novice programmers, using students at two different institutions, who were learning different programming languages. Influenced by the work of Dehnadi and Bornat, we gave our students a simple test, of our own devising, in their first three weeks of formal instruction in programming. That test only required knowledge of assignment statements. We found a wide performance difference among our two student cohorts. Furthermore, our test was a good indication of how students performed about 10 weeks later, in their final programming exam. We interpret our results in terms of our neo-Piagetian theory of how novices learn to program.

1. Introduction

Many computing educators have conjectured that success in learning to program requires a special talent or mental inclination. Some recent work in that area by Dehnadi and Bornat (Dehnadi and Bornat, 2006; Dehnadi, 2006) has generated much interest. They devised a test, consisting of nothing but assignment statements, which they gave to students who (it was believed) had no prior instruction in programming. They found that the test was a reasonable predictor of success in a first programming course. However, attempts at replicating those results with students from other institutions have met with mixed results (Caspersen, Larsen and Bennedsen, 2007; Bornat, Dehnadi and Simon, 2008; Lung, Aranda, Easterbrook and Wilson, 2008). Those mixed results have led to some clarifications and refinements on the original work (Bornat, Dehnadi, and Barton, 2012).

A key concept in Dehnadi and Bornat's work is consistency in the application of an algorithmic model of program execution. They believe that, prior to formal instruction in programming, a student's model of program execution need not be a correct model. Instead, what matters is whether a student applies a model consistently. For example, when hand executing assignment statements, prior to receiving formal instruction on programming, a student might mistakenly assign the value from left to right rather than right to left, but as long as the student applied that model consistently, prior to receiving formal instruction on programming, Dehnadi and Bornat found that the student stood a better chance of learning to program from subsequent formal instruction.

While our work was influenced by Dehnadi and Bornat, our pedagogical interest is in the early identification of students (as early as weeks 2-3 of formal instruction) who are in danger of failing their introductory programming course. Since, after 2-3 weeks, our students have been taught the correct model for assignment statements, our interest is in whether the students have learnt that correct model. We have therefore designed our own test, which we describe in the next section.

2. The Test

The specific test questions shown below are from the Python version of the test, used at the Queensland University of Technology (QUT). Given that this test is only concerned with assignment statements on integer variables, the Java version of the test used at the University of Technology, Sydney (UTS) is almost the same. The most common change in the Java version is that each line of

code ends with a semicolon. The UTS Java version has been presented in full elsewhere (Ahadi and Lister, 2013).

The design of our test was influenced by our prior work on applying neo-Piagetian theory to programming (Lister, 2011; Teague and Lister, 2014). In that work, we proposed a three stage model of the early stages of learning to program, which are (from least mature to most mature):

- **Sensorimotor:** The novice programmer has an incorrect model of program execution.
- **Preoperational:** The novice can reliably manually execute ("trace") multiple lines of code. These novices often make inductive guesses about what a piece of codes does, by performing one or more traces, and examining the relationship between input and output.
- **Concrete operational:** The novice programmer reasons about code deductively, by reading the code itself, rather than using the preoperational inductive approach. This stage is the first stage where students begin to show a purposeful approach to writing code.

2.1 Semantics of Assignment Statements — Questions 1(a) and 1(b)

Questions 1(a) and 1(b) tested whether a student understood the semantics of assignment statements—that is, the value on the right of the assignment is copied to the left, overwriting the previous value. The specific questions were as follows:

Q1 (a). In the boxes, write the values in the variables after the following code has been executed:

```
a = 1
b = 2
a = 3
```

The value in `a` is ☐ and the value in `b` is ☐

Q1 (b). In the boxes, write the values in the variables after the following code has been executed:

```
r = 2
s = 4
r = s
```

The value in `r` is ☐ and the value in `s` is ☐

Dehnadi and Bornat used similar questions. Du Boulay (1989) summarised a number of problems that students have with variables and assignment statements. One of the problems he described was the analogy of a variable as a "box", leading to the misconception that a variable may hold more than one value, and thus the novice does not realize that the old value in a variable is overwritten by a new value. The above two questions are intended to detect students who have misconceptions like these. According to our neo-Piagetian model, students struggling with the above two questions are at the sensorimotor stage of learning to program.

2.2 Effect of a Sequence of Statements — Question 1(c)

Question 1(c) tested whether a student understood the effect of a sequence of assignment statements; that the statements were executed one at a time. The specific question was as follows:

Q1 (c). In the boxes, write the values in the variables after the following code has been executed:

```
p = 1
q = 8
```

The value in `p` is ☐ and the value in `q` is ☐

```
q = p
p = q
```

Dehnadi and Bornat used a similar question. Some students mistakenly interpret this code as swapping the values in variables `p` and `q`, which Pea (1986) called an "intentionality bug", where novices believe "there is a hidden mind somewhere in the programming language that has intelligent interpretive powers". (While sequence also matters in Q1(a) and Q1(b), the intentionality bug in Q1(c) exposes novices who apply sequence weakly.) A student who can do Q1(a) and 1(b), but who struggles with this question, is working at the late sensorimotor / early preoperational stage.

2.3 Tracking Intermediate Variable Values — Questions 1(d) and 1(e)

Even when students understand both assignment statements and the effect of a sequence of statements, they can still make frequent errors because they cannot reliably track the changing values in variables through a series of assignment statements. In many cases this is because the students try to retain the variable values in their mind (i.e. working memory), rather than write down those values. Another source of error is that students use arbitrary and error prone ways of recording the variable values on paper (Teague and Lister, 2014). Questions 1(d) and 1(e) were designed to test whether a student could track the changing values of three variables in a sequence of three assignment statements (i.e. the three statements following the initialization of the three variables):

Q1 (d). In the boxes, write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 3
```

The value in `x` is ☐ `y` is ☐ and `z` is ☐

```
x = y
z = x
y = z
```

Q1 (e). In the boxes, write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 0
```

The value in `x` is ☐ `y` is ☐ and `z` is ☐

```
z = x
x = y
y = z
```

Dehnadi and Bornat used similar questions. According to our neo-Piagetian model, a student who can do these questions is at least in the middle range of the preoperational stage.

2.4 Inductive Reasoning — Question 1(f)

A novice who answers Question 1(f) correctly but who answers incorrectly Question 2 (see next section) is probably reasoning about code inductively and thus is working at the preoperational stage:

Q1 (f). In part (e) above, what do you observe about the final values in `x` and `y`? Write your observation (in one sentence) in the box below.

Sample answer: *the original value in x is now in y, and vice versa.*

2.5 Deductive Reasoning and Code Writing — Questions 2 and 3

Questions 2 and 3 were aimed at the novice who reasons in a concrete operational fashion:

Q2. The purpose of the following three lines of code is to swap the values in variables `a` and `b`, for any set of possible values stored in those variables.

```
c = a
a = b
b = c
```

In one sentence that you should write in the box below, describe the purpose of the following three lines of code, for any set of possible initial integer values stored in those variables. Assume that variables `i`, `j` and `k` have been declared and initialized.

```
j = i
i = k
k = j
```

Sample answer: *it swaps the values in variables i and k.*

Q3. Assume the variables `first` and `second` have been initialized. Write code to swap the values stored in `first` and `second`.

Sample answer:
```
  temp = first
 first = second
second = temp
```

Du Boulay (1989, p. 290) described some of the problems novices might have when attempting Q3: *Many students get these assignment statements in the wrong order and express individual assignments back to front. Difficulty in expressing the overall order of the assignments may be due to a lack of regard for the sequential nature of the three* [lines of code, that] *look a lot like three equations which are simultaneous statements about the properties of* [the three variables] *rather than a recipe for achieving a certain internal state ...*" According to our neo-Piagetian model, students who struggle with Q2 and Q3 in these (and other) ways are students at the preoperational stage (or lower). Students who correctly answer both Q2 and Q3 are probably at the concrete operational stage.

3. The Conduct and Grading of the Test — including some threats to validity

The introductory programming courses at both institutions comprised a 13 week semester where classes each week comprised a two hour lecture and, commencing in week 2, 2-3 hours of tutorial and/or lab classes. Students completed our test at the start of their lecture in either week 2 (at QUT) or week 3 (at UTS). The test was presented to the students on a single piece of paper, printed on both sides. At QUT, we eliminated from our data the small number of students who scored zero on the test, as those students had probably not attended the week 1 lecture. For consistency, at UTS we also eliminated data from students who scored zero. The UTS test contained an extra question that, for consistency, we have subsequently ignored. Some results for the UTS test with that extra question have been published earlier (Ahadi and Lister, 2013). As the test did not contribute to a student's final grade, the students may have had little motivation to perform well on the test, but equally they had little motivation to cheat. We stopped the test after around 15 minutes. Very few if any students were still working on the test when we called a stop.

Questions 1(a) to 1(f) were all worth 1 point, as were Q2 and Q3, for a total of 8 points. No fractional points were awarded — answers were treated as being either right or wrong, but English language issues in Q2, and syntactic errors in Q3, were ignored as long as a student's intention was clear.

4. Results: Falling Behind Early...

There are many differences in what and how the UTS and QUT cohorts are taught. For example, the UTS cohort was taught Java while the QUT cohort was taught Python. Our primary interest is in finding patterns in our results that are common to both institutions, as those patterns are more likely to generalise to other institutions.

Figures 1 and 2 show the distribution of student scores on the test, at UTS and QUT respectively. While the respective distributions have a different shape, a common feature of both distributions is the wide variation in test scores, spanning the entire range of possible marks.

As is always the case when grading students, it is one thing to assign a score to a student, but it is another thing entirely to know what that score means — for example, are the students who scored 4 on this test qualitatively different, as a general rule, from students who scored 6? Tables 1 (for UTS) and 2 (for QUT) address exactly that sort of question. These tables show the percentage of students who answered correctly each part of the test, broken down by total test score. In the remainder of this

section, we describe the results in those two tables. (Sections 4.1 and 4.2 below go to some pains to introduce and explain the information displayed in those tables.)

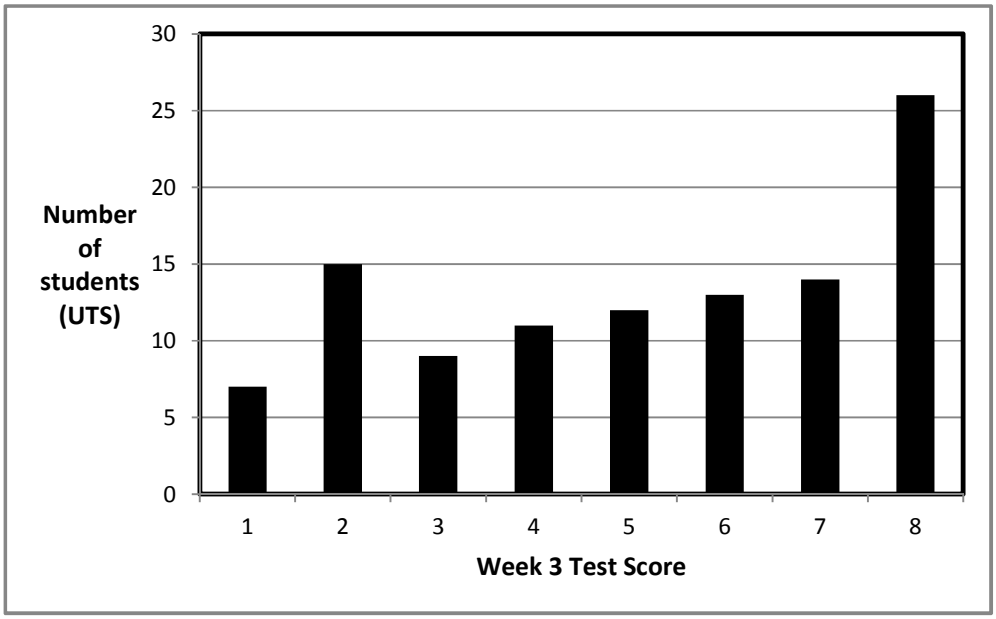


Figure 1 — Distribution of student total scores on the test at UTS (*N = 107*)

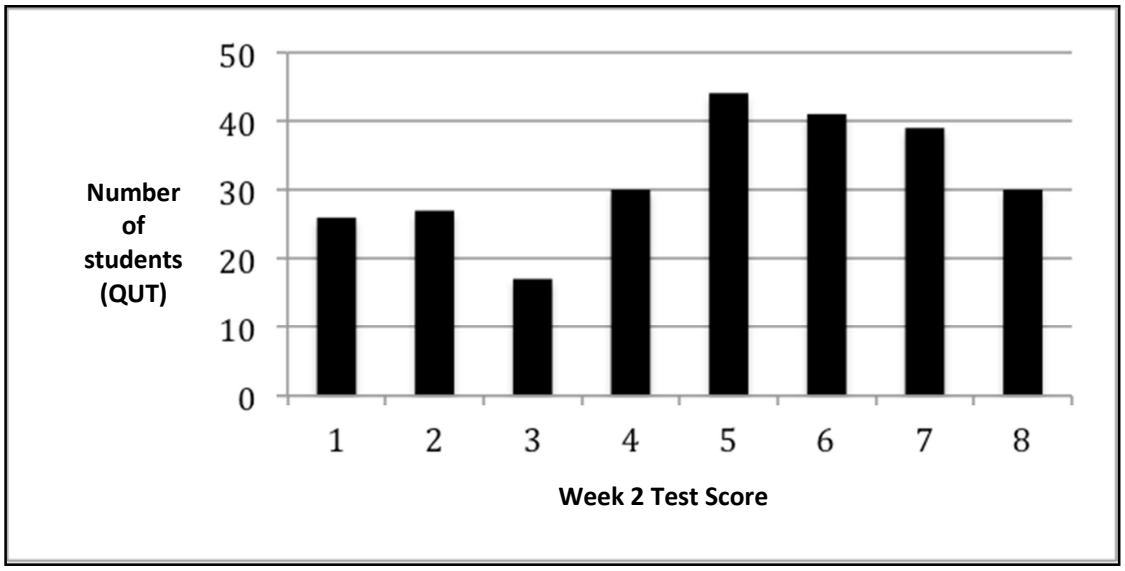


Figure 2 — Distribution of student total scores on the test at QUT (*N=254*)

4.1 Semantics of Assignment Statements — Questions 1(a) and 1(b)

At UTS (see Table 1), among the 15 students who scored 2 out of the possible 8 on the test, 93% answered Q1 (a) correctly (i.e. only one student answered incorrectly). All 15 students answered Q1 (b) correctly. However, for these 15 students, a performance difference of 93% and 100% on Q1 (a) and (b) is not statistically significant.

Of the 11 UTS students who scored 4 on the test, all answered Q1 (a) correctly and all but one student answered Q1 (b) correctly (i.e. 91%). In general, irrespective of their total score on the test, UTS students did very well on questions 1(a) and 1(b).

Week 3 Test Score	N	UTS							
		Sensorimotor			Preoperational			Concrete Operational	
		assignment		sequence & tracking values			induction	deduction	
		Q1(a)	Q1(b)	Q1(c)	Q1(d)	Q1(e)	Q1(f)	Q2	Q3
2	15	93%	100%	7%	0%	0%	0%	0%	0%
χ^2				*	***	***	***	***	
4	11	100%	91%	45%	55%	27%	27%	36%	18%
χ^2				**	**	**			
6	13	92%	92%	92%	100%	85%	54%	31%	54%
χ^2							***	***	***
8	26	100%	100%	100%	100%	100%	100%	100%	100%
1 to 8	107	95%	89%	64%	64%	56%	50%	44%	53%

*Table 1 — The percentage of UTS students who answered correctly each part of the test, broken down by total score. Cells containing asterisks indicate a statistically significant difference in the two percentages above and below the asterisk(s) (χ^2 test, * $p \leq 0.05$, ** $p \leq 0.01$ and *** $p \leq 0.001$). A thick vertical bar indicates a statistically significant difference in the two percentages to the left and right of the bar (χ^2 test, but only at the $p \leq 0.1$ level). All χ^2 tests were performed on the raw numbers from which the percentages were calculated.*

Week 2 Test Score	N	QUT							
		Sensorimotor			Preoperational			Concrete Operational	
		assignment		sequence & tracking values			induction	deduction	
		Q1(a)	Q1(b)	Q1(c)	Q1(d)	Q1(e)	Q1(f)	Q2	Q3
2	27	59%	70%	26%	26%	15%	4%	0%	0%
χ^2		*		***	**	**		*	
4	30	87%	83%	80%	63%	53%	13%	13%	7%
χ^2			*	*	***	***	***	*	*
6	41	88%	98%	98%	98%	88%	66%	34%	32%
χ^2		*				*	***	***	***
8	30	100%	100%	100%	100%	100%	100%	100%	100%
1 to 8	254	78%	83%	76%	72%	65%	42%	33%	31%

Table 2 — The percentage of QUT students who answered correctly each part of the test, broken down by total score. The cells containing asterisks, and also the thick vertical bars between some cells, indicate the same types of statistically significant differences as in Table 1.

The QUT students represented by Table 2 also did fairly well on questions 1(a) and 1(b). The only exception is the performance on 1(a) of students who scored 2 on the test. Only 59% of those 27 students answered that question correctly. Of the 30 QUT students who scored 4 on the test, 87%

answered Q1 (a) correctly. In Table 2, between those two percentages for Q1(a) (i.e. 59% and 87%), there is a grey cell containing an asterisk, which indicates that the difference in these two percentages is statistically significant (χ^2, $p \leq 0.05$). In the two columns for questions 1(a) and 1(b), there are two other grey cells containing asterisks; thus, while the QUT students represented by Table 2 did fairly well on questions 1(a) and 1(b), those QUT students with higher overall scores on the entire test did statistically better on those questions.

In summary, on inspection of both Table 1 and Table 2, among the students at both institutions who scored 2 or higher on the test, most had a good grasp of the semantics of assignment statements.

4.2 Effect of a Sequence of Statements — Question 1(c)

At UTS (see Table 1), among the 15 students who scored 2 out of the possible 8 on the test, only 7% (i.e. 1 student) answered Q1 (c) correctly. Among the students who scored 4 on the test, 45% answered Q1 (c) correctly. As indicated by the grey cell between those two percentages, which contains an asterisk, the difference in these percentages is statistically significant. Below that 45% in Table 1, another grey cell, containing two asterisks, indicates that there is a statistically significant difference ($p \leq 0.01$) between the students who scored 4 and students who scored 6 (i.e. 45% vs. 92%).

While the percentages for Q1 (c) at QUT are different (see Table 2), the test for statistical significance shows the same pattern at both institutions — students who scored 2 did poorly on Q1(c), while students who scored 4 did significantly better, but not as well as students who scored 6 or 8.

In both Table 1 and 2, in the row for the students who scored 2, there is a thick vertical bar between the cells representing Q1 (b) and (c). This vertical bar, and the other vertical bars like it throughout both tables, indicate a statistically significant difference between the two horizontally adjoining cells (χ^2, but $p \leq 0.1$). There is another thick vertical bar in Table 1 between cells in the columns for Q1 (b) and Q1 (c), in the row for students who scored 4, but there is no corresponding vertical bar in Table 2.

In summary, on inspection of both Table 1 and Table 2, most students who scored 2 had a poor grasp of sequence. Most of the students with higher scores on the test had a better grasp of sequence.

4.3 Tracking Intermediate Variable Values — Questions 1(d) and (e)

As described in section 2, questions 1(d) and 1(e) were designed to test whether a student could track the changing values of three variables in a sequence of three assignment statements. In summary, on inspection of both Table 1 and Table 2, only the students who scored 6 or higher could reliably track the values in variables. Thus most students scoring 6 or higher were preoperational or higher.

4.4 Inductive Reasoning — Question 1(f)

As described in section 2, question 1(f) was designed to identify students who can make reasonable inductive guesses about the function of a piece of code based upon the input/output behaviour. Since a student could not be expected to answer Q1 (f) correctly if that student had answered Q1 (e) incorrectly, it is the difference in percentages between Q1 (f) and Q1 (e) that is of interest, especially statistically significant differences (i.e. the thick vertical bars between those two table columns).

At both institutions, most students who scored 2 on the test performed poorly on both Q1 (e) and Q1(f). At QUT, among students who scored 4, there is a statistically significant difference between performance on Q1 (e) and Q1 (f), but not at UTS. There is, however, a statistically difference at both institutions among students who scored 6.

At both institutions, when looking down the table column for Q1 (f), it is apparent that most students who scored 2 or 4 did very poorly on this question, while the students who scored 6 exhibited mixed performance. Only the group of students who scored 8 on the test did very well on this question.

For this question, the only clear result that applies across both institutions is that students who scored 6 on the test tended to do well on the Q1 (e) tracing question but did significantly worse on the Q1 (f) inductive reasoning question.

4.5 Deductive Reasoning and Code Writing — Questions 2 and 3

On none of the overall test scores, at either institution, was there a statistically significant difference in the performance on Q2 and Q3. In both Tables 1 and 2, the only group of students who did well on both Q2 and Q3 were the students who scored a perfect 8 on the test.

We did not survey our students to establish any prior knowledge in programming, since self reporting is notoriously unreliable, but the results for Q2 and Q3 suggest that most students who scored 6 or less on this test are unlikely to have had any useful prior experience of programming.

4.6 A Neo-Piagetian Summary of Tables 1 and 2

On inspection of both Table 1 and Table 2, students with a total score of:

- 2 tended to have a grasp of the semantics of individual assignment statements but a poor grasp of sequence, and were thus working at the late sensorimotor / early preoperational stages.
- 4 were showing some ability to track values but many struggled with inductive reasoning, so we characterise this group of students as being early to mid-range preoperational.
- 6 were usually successfully tracking values and a majority could perform inductive reasoning, so we characterise this group of students as being late preoperational.
- 8 were the only group of students who performed consistently well on Q2 and Q3, so we characterise this group of students as being concrete operational.

5. Results: ... and Staying Behind

This section examines the relationship between performance on the test held early in semester and performance on the final exam at the end of the 13 week semester.

5.1 UTS Multiple Choice Exam

At UTS, the exam was entirely multiple choice. Figure 3 shows the probability that a UTS student would finish in the top half of the class, as a function of their performance on the week 3 test. The size of each black disc indicates the number of students who received that week 3 test score (i.e. the size of the discs is proportional to the size of the bars in Figure 1). The linear regression calculation represented by the dashed line was weighted according to the size of the discs. This was done by

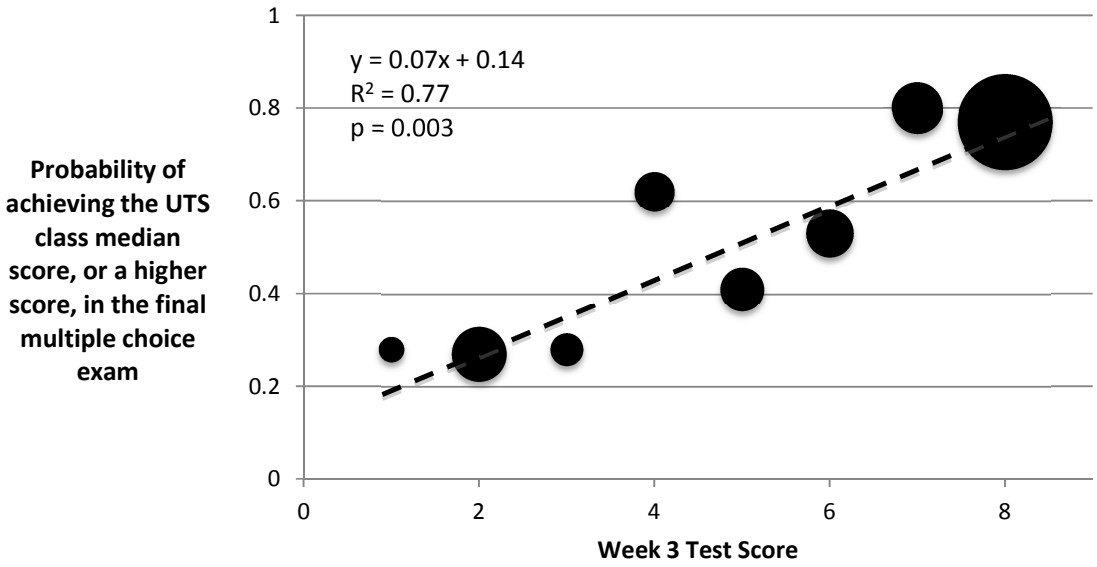


Figure 3 — UTS student scores on the test versus performance on the final exam (N=107).

performing the regression with 26 duplicate data points for test score = 8, 13 duplicate data points for test score = 6 and so on, for all test scores. All regression lines in subsequent figures were calculated this same way.

The use of the median in Figure 3 facilitates comparisons across institutions, since student ability and exam difficulty varies across institutions. Of course, approximately half of all students must perform above the median, and half below. However, that condition would still be satisfied if the regression line in Figure 3 was horizontal. In fact, from a pedagogical point of view, it would be best if that line of regression was horizontal, since the end-of-semester fate of a student should not be strongly attributable to their performance as early as the third week of a 13 week semester — but on the contrary, Figure 3 shows that many students did not recover from their slow start to the semester.

By the end of a 13 week semester, students had of course covered many more programming topics than the assignment statement tested in week 3. At UTS, approximately half the final exam covered basic object-oriented concepts, while the other half emphasized common 3GL searching algorithms and quadratic sorting algorithms. The following question indicates the general level of difficulty:

> This question refers to the Linear Search algorithm, studied in lectures, for an array “s” where the elements are stored in ascending order, and the final position in the array is stored in a variable “last”. The search should terminate as soon as either the value in variable "e" is found in the array, or it is established that the value is not in the array. Using a variable “pos” to scan along the array, the correct loop is:
>
> (a) `while ( (pos<=last)  &&  (pos < e) )  ++pos;`
> (b) `while ( (pos<=last)  &&  (s[pos] < e) )  ++pos;`
> (c) `while ( (s[pos]<=s[last])  &&  (pos < e) )  ++pos;`
> (d) `while ( (s[pos]<=s[last])  &&  (s[pos] < e) )  ++pos;`

5.2 QUT Multiple Choice Questions

The final exam at QUT comprised two parts: a set of multiple choice questions and a set of questions that required students to write Python code. Figure 4 shows the probability that a student would finish in the top half of the class for the multiple choice part of the exam, as a function of their performance on the week 2 test. As was also the case for UTS, Figure 4 shows that QUT students who performed poorly on the test — held in week 2! — were unlikely to overcome their poor early start to the semester and finish in the top half of the class.

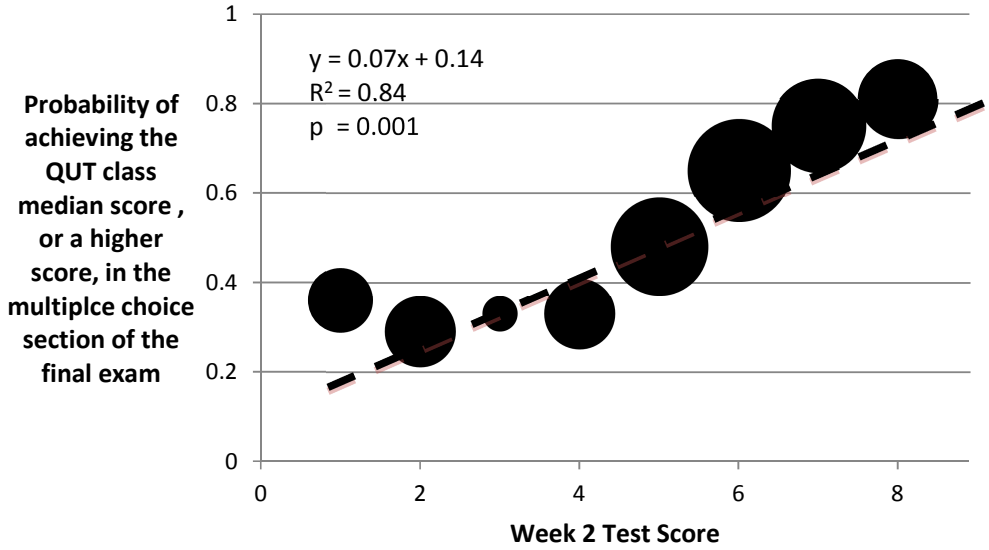


Figure 4 — QUT student scores on the test versus performance on the multiple choice component of the final exam (N=254).

5.3 QUT Short Answer Questions

While Figures 3 and 4 both describe a clear relationship between the test near the start of the semester and performance on the end-of-semester multiple choice questions, perhaps that relationship can be attributed to the relatively simple nature of multiple choice questions? For example, multiple choice questions do not require a student to write code. Figure 5 tests that idea. It shows the probability that a QUT student would finish in the top half of the class, as function of their performance on the week 2 test, for the short answer part of the QUT exam. This figure is similar to Figures 3 and 4 — the resemblance between Figure 5 and Figure 4 is uncanny.

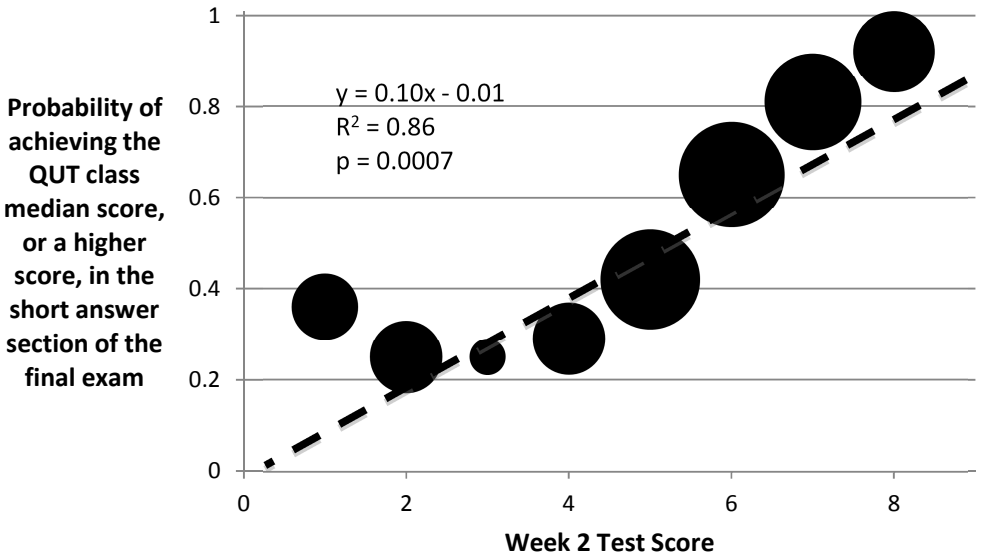


Figure 5 — QUT student scores on the test versus performance on the short answer section of the final exam (N=254).

One of the short answer questions in the exam is illustrated by the code shown below. The students were given the code on the left, which "rotates" the values in the list `items` one place to the left, with the leftmost value moving to the rightmost position. The students were required to write code to do the opposite transformation; that is, write code to rotate the values in array `items` one place to the *right*, with the rightmost item moving to the leftmost position. The solution is shown below:

"Rotate Left" Code Given to the Students

```
temp = items[0]
for index in range(len(items)-1):
    items[index] = items[index + 1]
items[len(items) - 1] = temp
```

"Rotate Right" Code Required from Students

```
temp = items[len(items) - 1]
for index in range(len(items)-1,0,-1):
    items[index] = items[index - 1]
items[0] = temp
```

As part of the instructions for this question, students were effectively given the **for** loop header required in their answer, so the question was marked out of 3, with one point for each of the remaining three lines of code. Figure 6 shows the probability that a student scored 2 or 3 for this question, as a function of their performance on the week 2 test. This graph is similar to the three earlier graphs for performance on final exams — our results are robust, across the two institutions, and also across multiple choice and short answer questions.

In neo-Piagetian terms, this "Rotate Right" short answer question requires the student to manifest concrete operational reasoning (Lister, 2011). In section 4.6, we provided a neo-Piagetian summary for Tables 1 and 2. Building on that earlier summary, we now provide a neo-Piagetian summary where we contrast the performance of students in Table 2 (i.e. at week 2 of semester) and their end of

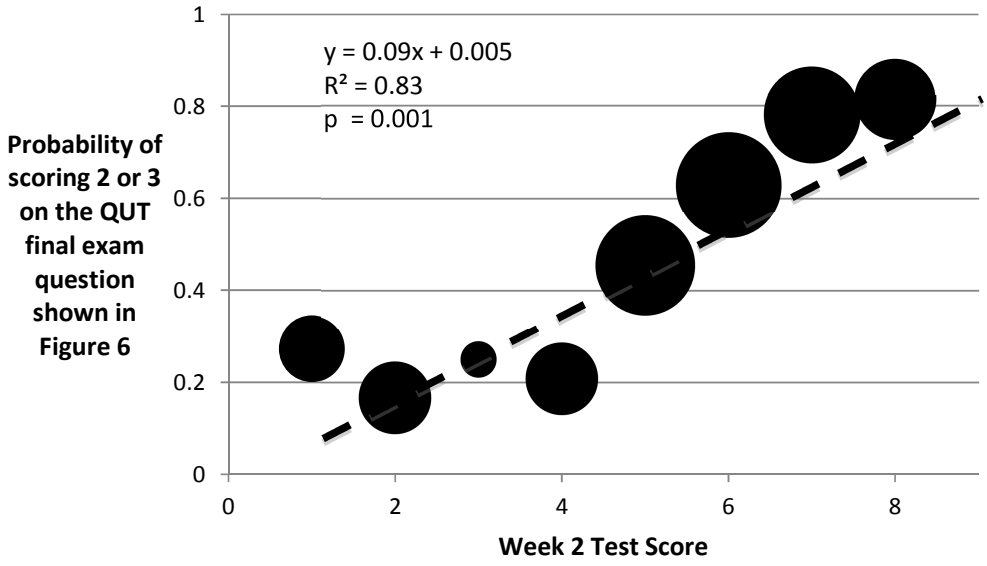


Figure 6 — QUT student scores on "Rotate Right" short answer question (N=254).

semester performance at QUT on the "Rotate Right" problem (as shown in Figure 6). Students with a total score in Table 2 of:

- 2 or 4 were characterised as being spread from sensorimotor to mid-preoperational. By the end of semester, around 20% of those students manifested concrete operational reasoning on the "Rotate Right" problem.
- 6 were characterised as late preoperational. By the end of semester, around 60% of these students manifested concrete operational reasoning on the "Rotate Right" problem.
- 8 were characterised as concrete operational. By the end of semester, around 80% of these students manifested concrete operational reasoning on the "Rotate Right" problem. (N.B. the remaining 20% did not "go backwards", as the week 2 test was on assignment statements only, whereas the "Rotate Right" problem tested more demanding concepts and skills.)

6. Conclusion

In this paper, we have described a test on novice programmers, in weeks 2 and 3 of semester, with students from two different institutions, where (among other pedagogical differences) the students are taught two different programming languages. At both institutions we found a wide performance difference among each student cohort on that test. Furthermore, that early test is a good indication of how students performed about 10 weeks later, in their final exam. In terms of neo-Piagetian theory, students who exhibit lower neo-Piagetian stages in the early test are unlikely to manifest the higher concrete operational stage of reasoning in the final exam.

People who believe that programming requires an innate talent may feel justified by our results. While our results do not disprove the existence of an innate talent for programming, we do not subscribe to that view. As we have summarised in this paper, in earlier work we have developed a neo-Piagetian theory of how novices learn to program. Neo-Piagetian theory is based upon the constructivist principle that cognitive skills are primarily learnt, not innate. Our neo-Piagetian perspective leads us to view the curriculum for programming as comprising two dimensions. On one dimension are the nuts and bolts of how programming languages work. That dimension is emphasised in today's classroom. The other and more neglected dimension comprises the skills for reasoning about programs, sometimes referred to as the notional machine (du Boulay, 1989), but which we think of in neo-Piagetian terms. This dimension is often not explicitly taught, especially in the first few weeks of

learning to program. We believe that, with every increment along the "nuts and bolts" dimension (i.e. with every new programming construct taught), all the neo-Piagetian stages of reasoning need to be explicitly reprised. In the test we used in weeks 2 and 3 of semester, questions 1(f), 2, and 3 represent the types of learning exercises that students need when they are introduced to assignment statements. As a further example, the Bubblesort algorithm could be introduced well before loops are explicitly taught, using implicit "uncompressed" loops (Milner, 2008).

We close by speculating, from a neo-Piagetian perspective, on Dehnadi and Bornat's claim that people who apply a consistent model of program execution are more likely to learn to program, even when their model is wrong. Perhaps those people enjoy an early affective advantage, not a cognitive advantage. That is, people who show an early preference for consistency may be especially well motivated to perform the deep learning required to push through the earlier neo-Piagetian stages and gain the consistency of reasoning that only begins at the concrete operational stage.

7. Acknowledgements

The grant for this work was from the Office of Learning & Teaching, of the Australian Government.

8. References

Ahadi, A. and Lister, R. (2013). *Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant*? Ninth International Computing Education Research Workshop (ICER '13). ACM, USA, pp. 123-128. http://doi.acm.org/10.1145/2493394.2493416

Bornat, R., Dehnadi, S., and Simon (2008) *Mental models, consistency and programming aptitude*. Tenth conference on Australasian Computing Education (ACE '08). pp. 53-61. http://crpit.com/confpapers/CRPITV78Bornat.pdf

Bornat, R., Dehnadi, S., and Barton, D. (2012) *Observing Mental Models in Novice Programmers*. 24th Annual Workshop of the Psychology of Programming Interest Group, London. http://www.ppig.org/papers/24/8.Observing_mental_models-Richard%20Bornat.pdf

Caspersen , M., Larsen , K., Bennedsen, J. (2007) *Mental models and programming aptitude*. Innovation and Technology in computer science education (ITiCSE '07), Scotland, pp. 206-210. http://doi.acm.org/10.1145/1269900.1268845

Dehnadi, S., and Bornat, R. (2006) *The camel has two humps* (*working title*) https://www.cs.kent.ac.uk/dept_info/seminars/2005_06/paper1.pdf

Dehnadi, S. (2006) *Testing programming Aptitude*. 18th Annual Workshop of the Psychology of Programming Interest Group, Brighton, pp. 22-37. http://www.ppig.org/papers/18th-dehnadi.pdf

Du Boulay, B. (1989). *Some difficulties of learning to program*. In E. Soloway and J. C. Sphorer (eds), Studying the novice programmer, New Jersey: Lawrence Erlbaum. pp. 283-300.

Lister, R. (2011) *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Thirteenth Australasian Computing Education Conference (ACE '11), pp. 9-18. http://crpit.com/confpapers/CRPITV114Lister.pdf

Lung, J., Aranda, J., Easterbrook, S., and Wilson, G. (2008) *On the difficulty of replicating human subjects studies in software engineering*. 30th international conference on Software engineering (ICSE '08), pp. 191-200. http://doi.acm.org/10.1145/1368088.1368115

Milner, W. (2008) *A Loop is a Compression*. 20th Annual Workshop of the Psychology of Programming Interest Group, Lancaster. http://www.ppig.org/papers/20th-milner.pdf

Pea, R. (1986) *Language-Independent Conceptual "Bugs" in Novice Programming*. Journal of Educational Computing Research, Vol. 2(1), pp. 25-36.

Teague, D. and Lister, R. (2014) *Longitudinal Think Aloud Study of a Novice Programmer*. Sixteenth Australasian Computing Education Conference (ACE '14), pp. 41-50. http://www.crpit.com/Vol148.html

Affective Learning with Online Software Tutors for Programming

Amruth N. Kumar

Computer Science
Ramapo College of New Jersey
amruth@ramapo.edu

Keywords: POP-I.A Learning to program, POP-II.A Individual differences, POP-II.B. Program comprehension, POP-V.B Cross-sectional studies, POP-VI.E. Computer science education research

Abstract

We conducted a study to see if there was any difference in the affective learning of students who used software tutors to learn programming concepts. We used two software tutors – on arithmetic expression evaluation and tracing selection statements for this study. Data was collected over five semesters with the arithmetic tutor and over four semesters with the selection tutor, yielding a sample size in the thousands. The data was analysed using ANOVA, with sex (male versus female), representation (Caucasians and Asians versus other races), and discipline (Computer Science versus other disciplines) as fixed factors. We found no difference in the affective learning of any demographic group - all the groups felt in equal measure that they had learned using the software tutors. This is a positive result since affective learning complements cognitive learning. But, we did find statistically significant difference between the sexes and between the representation groups on prior-preparedness on arithmetic tutor, but not selection tutor. This may be because arithmetic tutor covered concepts that students had been exposed to in high school, but the concepts covered by selection tutor were unique to programming and unlikely to have been seen by students who had had no prior programming experience. Finally, non-Computer Science majors learned significantly more concepts than Computer Science majors on both the tutors, even though no significant difference was found between the two disciplinary groups on prior-preparedness. This may allude to difference in perceived self-efficacy of the two groups towards learning from software tutors.

1. Motivation

Essentialism permeates the discussion of participation of women in computing. Essentialism posits that people have properties essential to their composition, and that all members of a demographic group share common characteristics (Frieze & Quesenberry, 2013). Gender essentialism has been supported by studies that suggest that girls are concerned about the passivity of their interactions with the computer as a (learning) tool, and that they dislike *narrowly and technically focused* programming classes ((AAUW, 2000). Whereas men view computers as machines, women view them as tools for use within a *societal and/or interdisciplinary context* (Margolis & Fisher, 2002). In response, National Center for Women & Information Technology (NCWIT) recommends that one strategy for retaining undergraduates (women) in computing is to align assignments and coursework with student interests and career goals (Barker & Cohoon, 2009).

All along, there have also been arguments against using essentialism to explain the differences in the attitudes of men and women towards computing, and therefore, the difference in their participation levels in the discipline. Klawe warned about the widespread *misconceptions (*Klawe, 2002) that “girls don’t like using computers” and that “girls are people people and computer people are not people people”. Explaining the differences based on culture rather than essentialism has been recently proposed as a more effective way of increasing the participation of women in computing (Frieze & Quesenberry, 2013).

We have been developing and deploying software tutors called problets (problets.org) that help students learn programming concepts by solving drill-and-practice problems. The problems presented by these tutors are structured around individual programming constructs, and therefore, are *narrowly and technically focused*. Since the problems are automatically generated as instantiations of templates, they lack any *societal or interdisciplinary context* recommended in essentialist proposals for increasing the participation of women in computing, e.g., (Margolis & Fisher, 2002). Therefore, we have been especially interested in the differential effects, if any, of our software tutors on women versus men vis-à-vis cognitive and affective learning. However, users of our software tutors are asked to voluntarily identify their sex (biological notion of male/female) rather than their gender (social/cultural notion of man/woman) (Sears, 1999). Therefore, henceforth, we will be referring to male versus female students rather than men versus women.

In Computer Science, in addition to Caucasians who are traditionally represented, Asians have also been positively stereotyped because of its quantitative nature (Kao, 1995). The other racial groups, viz., Black/African American, Hispanic/Latino, Native American, Native Hawaiian/Pacific Islander and Other, are traditionally classified as under-represented. *Myths* of genetic determinism have been considered to be one of the barriers for participation of under-represented groups in computing (Eglash, Gilbert, & Foster, 2013). Stereotype threat (Steele & Aronson, 1995) has also been listed as one of the factors that could be contributing to problems with recruitment and retention of female and minority students (Beyer, Rynes, Perrault, Hay, & Haller, 2003; Peckham et al., 2007). Here again, culture-based approaches to computing education have been proposed as a way to increase participation of under-represented groups in computing education (Eglash et al., 2013).

Drill-and-practice problem-solving as supported by our software tutors is devoid of any cultural context. So, we have also been interested in differential effects, if any of our tutors on the cognitive and affective learning of traditionally-represented versus under-represented racial groups.

1.1. Prior Work with our Software Tutors

During earlier studies, we had found that female students learned just as well as male students using our software tutors (Kumar, 2007). They rated the usability, ability to learn from, and usefulness of the tutors more favourably than male students (Kumar, 2006a) and this was not an artifact of the design of the survey instrument (Kumar, 2008b). They had lower prior self-confidence (Kumar, 2011), but using the software improved their self-confidence to be on par with that of male students (Kumar, 2006b, 2008a).

We had found that students from under-represented groups rated the tutors more favourably than those from traditionally represented groups (Kumar, 2009b). They had less prior preparation and lower prior self-confidence than students from traditionally represented racial groups (Kumar, 2011). However, students needed and benefited from the tutors in the same proportion, regardless of sex or racial group (Kumar & Kaczmarczyk, 2013).

One study of under-represented groups in computing found intersectionality, i.e., interaction between gender and race to be critical, i.e., under-represented males may have less in common with under-represented females than say, Caucasian females (Trauth, Cain, Joshi, Kvasny, & Booth, 2012). This concurs with our own observations that when evaluating educational interventions in computing, significant interactions exist among demographic groups (Kumar, 2009a).

Given these prior results on self-confidence, assessment of tutor, and to a lesser extent, cognitive learning and prior preparation, we focused on affective learning in this study, but included supplemental investigation of prior-preparation (how well the students were prepared before using our software tutors) and number of concepts learned (cognitive learning) using our software tutors.

1.2. Affective Learning

Affective learning relates to emotional aspects of learning, such as:

- Motivation, which mediates learning by increasing or decreasing cognitive engagement (Gottfried, 1990);

- Attitude towards learning – positive or negative; and
- Emotions such as anxiety, confidence, and boredom, which affect learning (e.g., (Kort, Reilly, & Picard, 2001)).

Researchers are increasingly acknowledging the importance of affective factors in computer-mediated learning (Picard et al., 2004), and are trying to address affective issues in the design of software tutors (e.g., (Baker, D'Mello, Rodrigo, & Graesser, 2010)).

Self-efficacy is one's judgment about what one can or cannot do (Bandura, 1977). It affects one's motivation, influences one emotionally, and is one of the more easily measurable aspects of affective learning. In our work, we focused on self-efficacy, i.e., learners' judgment of what they could or could not do with their knowledge after a cognitive learning session with our tutor.

2. Study Protocols and Instruments

For this study, we used two tutors – on arithmetic expression evaluation and tracing programs containing selection statements. These two tutors were selected because they promised large sample sizes – they are both typically used early in the semester and are used by larger numbers of students than tutors on more advanced topics such as functions and arrays. At the same time, the two tutors are not duplicative – they have different user interfaces, cover different programming concepts and require different problem-solving skills.

Each tutor was configured to administer pretest-practice-post-test protocol as follows:

- **Pretest** – During pretest, the tutor presented one problem per concept. If a student solved a problem correctly, the student was given credit for the corresponding concept. No feedback was provided to the student, and no more problems on the concept were presented to the student. On the other hand, if the student solved a problem incorrectly, feedback was presented to the student immediately after the student submitted his/her solution to the problem. Additional problems were presented on the concept during the subsequent stages.
- **Adaptive practice** – During this stage, additional problems were presented to the student on only the concepts on which the student made mistakes when solving problems during the pretest. For each such concept, the student was presented multiple problems until the student mastered the concept, i.e., solved at least 60% of the problems correctly. On each problem, the student received feedback explaining the correct answer step by step.
- **Post-test** - During this stage, the student was presented test problems on the concepts that the student had mastered during adaptive practice.
- **Affective Learning Survey:** During this stage, the student was asked to respond to 5 - 8 statements on affective learning, using a 5-point Likert-scale questionnaire.
- **Demographics** - Students were provided the option to identify their demographic information, including sex and race. Demographic information was solicited after the pretest-practice-post-test protocol to avoid the effects of stereotype threat (Kumar, 2012).

The entire protocol was administered online, back-to-back, with no break in between, all by the software tutor. The pretest-practice-post-test problem-solving session was limited to 30 minutes.

If a student solved all the problems correctly during pretest, the student was not presented any practice or post-test. If the student did not meet the minimum percentage correctness on a concept during practice, regardless of how many problems the student had solved on the concept during practice, the student was not presented any post-test problems on that concept. If a student did not get to practice or post-test stage, it could also have been because the student ran out of time. The concepts on which a student solved the problem incorrectly during pretest, demonstrated mastery during practice and solved a post-test problem are *practiced concepts.* Each practiced concept on which a pre-post increase in score was observed is also a ***learned concept****.*

2.1. Arithmetic Expression Evaluation Tutor

Tutor on Arithmetic Expression evaluation covered 25 concepts such as correct evaluation, precedence and associativity of the five arithmetic operators, divide-by-zero error, use of parentheses and coercion of data types. Students were presented with an expression involving one or more arithmetic operators, and were asked to evaluate it one operator at a time, i.e., pick the operator and the operands to which it applied, and enter the intermediate result of its evaluation (See Figure 1).

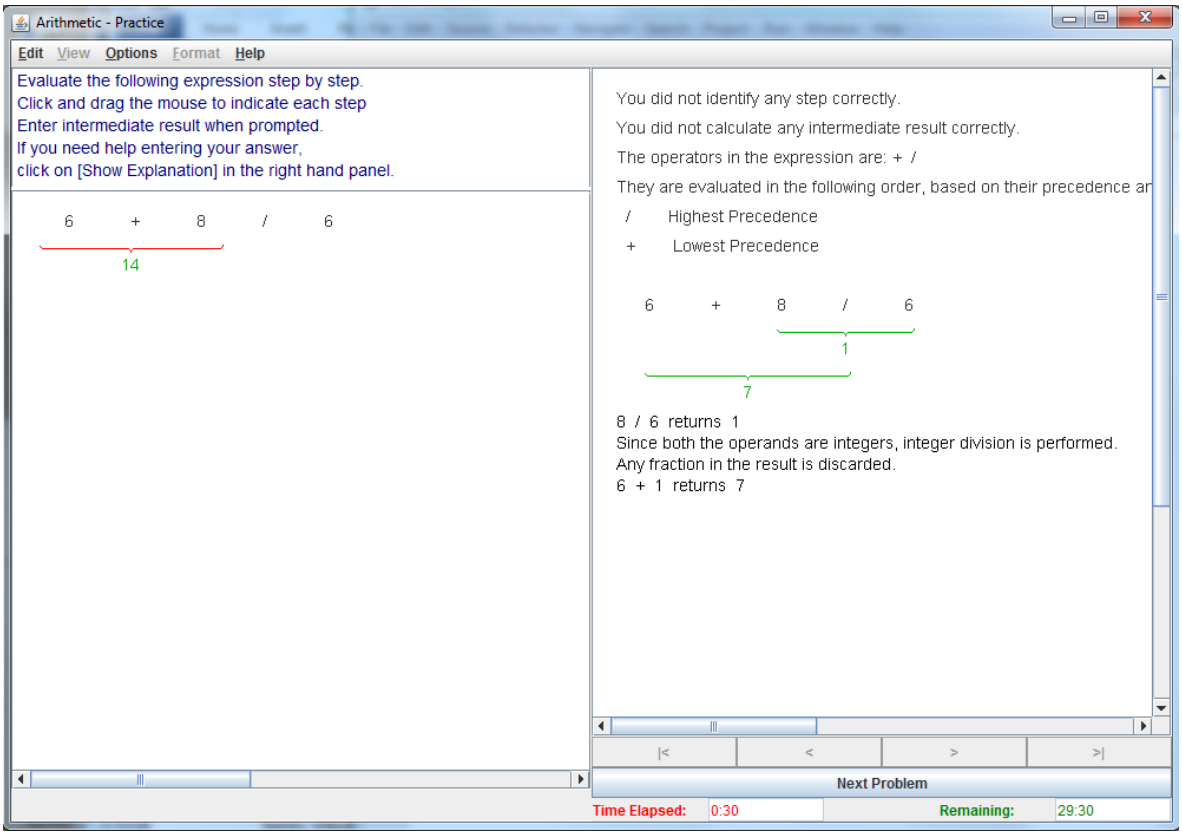


Figure 1: Snapshot of Arithmetic Tutor

The pretest contained 16 problems. The problems were selected to together cover all 25 concepts at least once. For example, the problem in the left panel of Figure 1 covers 4 concepts - correct evaluation of addition and integer division operators, and their relative precedences.

The affective survey administered with the arithmetic tutor consisted of the following 5 statements:

After using the tutor, I can do the following better:

1. Read arithmetic expressions
2. Evaluate arithmetic expressions
3. Find errors in arithmetic expressions
4. Write arithmetic expressions
5. Critique arithmetic expressions

These statements were picked to reflect the levels of revised Bloom's Taxonomy (Anderson & Krathwohl, 2001) – applying (2), analysing (1, 3), evaluating (5) and creating (4). Students responded on a 5-point Likert scale of Strongly Agree (coded as 1), Agree (2), Neutral (3), Disagree (4) and Strongly Disagree (5). Students had the option to skip the survey altogether or respond to only some of the statements.

2.2. Selection Statement Tutor

Tutor on selection statements covered 12 concepts such as one-way and two-way selection statements, nested selection statements, execution of the statement when the condition is true/false, etc. Students were presented a program containing selection statement(s) and were asked to identify the output of the program one at a time, along with the line number of the code that produced each output (See Figure 2).

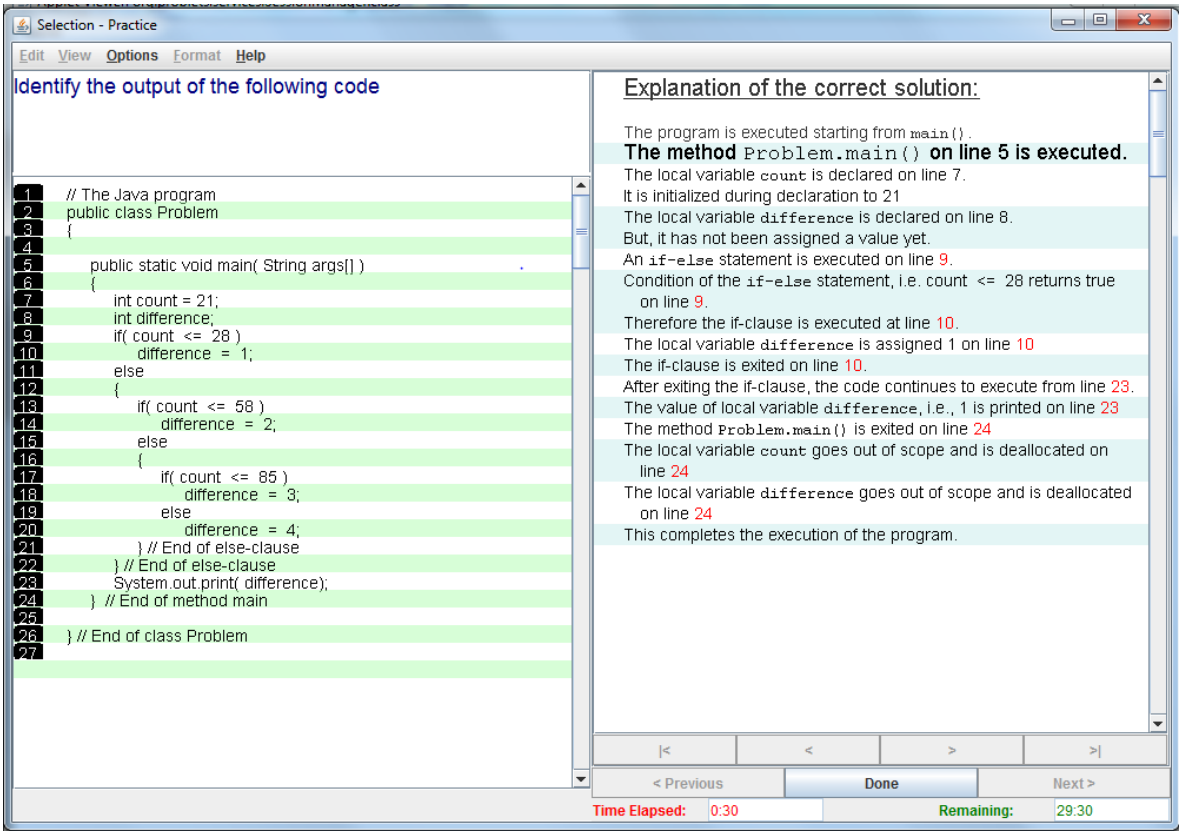


Figure 2: Snapshot of Selection Tutor

The pretest contained 12 problems, one for each of the 12 concepts covered by the tutor. For example, the program in the left panel of Figure 2 covers the concept of nested `if-else` statements.

The affective survey administered with selection tutor consisted of the following 8 statements:

After using the tutor, I can do the following better:

1. Understand the grammar rules of if-else statements
2. Understand the meaning of if-else statements
3. Read if-else statements
4. Predict the output of if-else statements
5. Debug if-else statements
6. Write if-else statements
7. Design if-else statements
8. Critique if-else statements

These statements were also picked to reflect the levels of revised Bloom's Taxonomy – understanding (1, 2), applying (4), analysing (3, 5), evaluating (8) and creating (6, 7). Once again, students

responded on a 5-point Likert scale and had the option to skip the survey altogether or respond to only some of the statements.

3. Data Collection and Analysis

Data Collection: The two tutors were used by students of instructors at several institutions each semester. Typically, these were students in introductory programming courses. Students typically used the tutors after class, on their own time, over the web, without instructor supervision. Data was collected over multiple semesters when the versions of the two tutors were kept unchanged.

Data Filtering: Students could use each tutor as many times as they wished. When they used a tutor more than once, only the first attempt when the student had solved most of the pretest problems was considered for analysis. Only those students were considered for analysis who had solved at least a minimum number of pretest problems.

Data Analysis: The following data was analysed for this study:

1. **Prior preparation:** The score per problem on the **pretest**, which is indicative of prior-preparation of the student, i.e., how well prepared the student was before using the tutor. Score per problem was used rather than the raw score in order to eliminate the effect of difference in the number of problems solved by students during pretest. The range of values for score per problem was 0 → 1.0.

2. **Learned concepts:** The number of learned concepts, i.e., the concepts on which the student solved the pretest problem incorrectly, solved sufficient practice problems to demonstrate mastery and solved a post-test problem correctly.

3. **Affective learning:** The aggregate Likert-scale score on the affective survey statements presented after using each tutor – 5 statements for Arithmetic tutor and 8 statements for Selection tutor. So, the range of possible values was 5 → 25 for Arithmetic tutor and 8 → 40 for Selection tutor.

The following demographic groupings were used for analysis:

1. **Sex**: male or female as self-identified by the student;
2. **Representation**: Based on the race self-identified by the students, Caucasians and Asians were combined into traditionally represented group and all other racial groups were combined into under-represented group;
3. **Discipline**: Based on the major self-identified by the students, Computer Science majors were compared against students from all other majors, which included Information Systems, Engineering, Basic Sciences, Business, Arts, Humanities, Social Sciences, and Other.

Since providing demographic information was optional, not all the students entered all their information. Since the tutors were adaptive and timed, not all the students got credit for learned concepts – some knew all the concepts before using the tutor and some ran out of time before being able to practice any concept all the way from pretest to post-test. Since filling out the affective learning questionnaire was not mandatory, not all the students filled it out. Therefore, N value varies with each analysis and has been individually noted in the ANOVA results in the next section.

3.1. Results of Analysis – Arithmetic Tutor

Data was collected over five semesters - Fall 2011 through Fall 2013. A controlled study of the effects of visualization was being conducted during these semesters. While the hypothesis of the controlled study is not relevant to the current study, treatment was taken as one of the factors during analysis of the collected data. Anyone who had solved fewer than 10 of the 16 pretest problems was dropped from analysis. After this, 1065 students remained in the control group and 1304 students in the test group, from 56 different institutions.

The pre-post change in score on learned concepts was as follows, all pre-post changes being statistically significant ($p <= 0.05$):

Arithmetic Tutor		**Pretest**	**Post-test**
Control (N=415)	Mean	0.05 ± 0.0185	0.9228 ± 0.0185
	Std-Deviation	0.1926	0.1923
Test (N=485)	Mean	0.0424 ± 0.0139	0.9349 ± 0.0155
	Std-Deviation	0.1557	0.1746

We conducted a 2 x 2 x 2 x 2 ANOVA analysis, with sex (male versus female), representation (Caucasians and Asians versus other races), discipline (Computer Science versus other majors) and treatment (without versus with visualization) as fixed factors.

On affective learning (sum of the five Likert-scale responses), we found no significant main effect for sex, representation or discipline. When we eliminated students who had scored greater than 95% on the pretest, we found a marginal main effect for representation [$F(1,971) = 3.54$, $p = 0.06$]: Caucasians and Asians reported greater affective learning (9.353 ± 0.374 – confidence intervals being at 95% confidence level) than other racial groups (10.049 ± 0.622).

On the pretest average score per problem, we found:

- Significant main effect for sex [$F(1,1729) = 5.26$, $p = 0.022$]: male students scored higher on the pretest (0.882 ± 0.09 points) compared to female students (0.859 ± 0.18 points). However, once all the students who had scored more than 95% on the pretest were eliminated, the difference was no longer significant. In other words, more male students scored greater than 95% on the pretest than female students.
- Significant main effect for representation [$F(1,1729) = 17.637$, $p < 0.001$]: Caucasians and Asians scored higher on the pretest (0.891 ± 0.1 points) than students from under-represented groups (0.849 ± .18 points).
- No significant main effect for discipline, and no significant interaction among the factors

In other words, male students and traditionally represented students were better prepared than their counterparts before using the software tutors. On the number of concepts learned we found:

- Significant main effect for discipline [$F(1,665) = 5.611$, $p = 0.018$]: non-Computer Science students learned more concepts (1.794 ± 0.157) than Computer Science majors (1.497 ± 0.189)
- No significant main effect for sex or representation

3.2. Results of Analysis – Selection Tutor

Data was collected over four semesters – Fall 2010 through Spring 2012. All the students got exactly the same treatment while using the tutor. Anyone who had solved fewer than 6 of the 12 pretest problems was dropped. After this, 908 students remained in the study from 37 different institutions.

The pre-post change in score on learned concepts was as follows, all pre-post changes being statistically significant ($p <= 0.05$):

Selection Tutor		**Pretest**	**Post-test**
(N=438)	Mean	0.0413 ± 0.0080	0.9156 ± 0.0127
	Std-Deviation	0.1226	0.1949

We conducted a 2 x 2 x 2 ANOVA analysis, with sex (male versus female), representation (Caucasians and Asians versus other races) and discipline (Computer Science versus other majors) as fixed factors:

- On affective learning (sum of the 8 Likert-scale responses) and average pretest score, we found no significant main effect for sex, representation or discipline.
- On the concepts learned, the only significant main effect we found was for discipline [$F(1,310) = 4.482$, $p = 0.035$]: once again, non-Computer Science students learned more concepts (1.95 ± 0.21) than Computer Science majors (1.488 ± 0.375).

3.3. Discussion of Results

Whereas arithmetic tutor covers concepts that students would have been exposed to in high school, selection tutor covers concepts that are unique to programming and are entirely new to anyone who has never programmed before. This might explain why we found male and traditionally represented students to be better prepared than their counterparts on arithmetic tutor concepts, but not on selection tutor concepts.

On both the tutors, non-Computer Science majors learned more concepts than Computer Science majors. This was true even though no significant difference was found between the two disciplinary groups on prior-preparedness (pretest average score) on either tutor. So, the greater learning of non-Computer Science majors cannot be attributed to greater need. Instead, it may allude to difference in perceived self-efficacy of the two groups towards learning from software tutors. We plan to explore this in the future.

Regardless of the differences found between groups on prior-preparation and concepts learned, no difference was found between demographic groups on affective learning on either tutor. In other words, all the demographic groups felt in equal measure that they had learned using the software tutors, which in itself is encouraging, since affective learning complements cognitive learning (Hurd, 2008). In the future, we plan to repeat this study with tutors for higher level concepts such as loops and functions.

One confounding factor of this study is that the survey instruments used for affective learning were not validated. The survey instruments were designed to elicit students' self-efficacy – their judgement about what they could or could not do on the topic of the tutor, in terms of revised Bloom's taxonomy. Self-efficacy associated with the use of computers is called Computer Self-efficacy (Compeau & Higgins, 1995). Numerous factors have been identified as contributing to computer self-efficacy, and are classified into 12 categories: enactive mastery, task characteristics, perceived effort, situation support, degree/quality of feedback, emotional arousal, vicarious experience, verbal persuasions, assigned goals/anchors, degree of professional orientation, age and attribution of cause (Marakas, Yi, & Johnson, 1998). The instruments used in this study deal only with enactive mastery, i.e., gaining relevant experience with the problem-solving task.

On the other hand, a supporting factor of the current study is that it was conducted with students using the tutors on their own time, unsupervised, i.e., *in-natura.* This reduced the likelihood of Hawthorne effect (Franke & Kaul, 1978) affecting the results. That these students chose to answer the survey on affective learning on their own time after solving problems with the tutor for up to 30 minutes, even though answering the survey was optional, adds credence to the results.

4. Acknowledgements

Partial support for this work was provided by the National Science Foundation under grant DUE-0817187.

5. References

AAUW. (2000). *Tech-Savvy: Educating Girls in the New Computer Age*: American Association of University Women.

Anderson, L. W., & Krathwohl, D. R. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Longman.

Baker, R. S. J. d., D'Mello, S. K., Rodrigo, M. M. T., & Graesser, A. C. (2010). Better to be frustrated than bored: The incidence, persistence, and impact of learners' cognitive-affective states during interactions with three different computer-based learning environments. *International Journal of Human-Computer Studies, 68*, 223-241.

Bandura, A. (1977). Self-Efficacy: Toward a Unifying Theory of Behavioral Change. *Psychological Review, 8*(2), 191-215.

Barker, L., & Cohoon, J. M. (2009). Key Practices for Retaining Undergraduates in Computing National Center for Women and Information Technology.

Beyer, S., Rynes, K., Perrault, J., Hay, K., & Haller, S. (2003). *Gender differences in computer science students*. Paper presented at the 34th SIGCSE Technical Symposium.

Compeau, D. R., & Higgins, C. A. (1995). Computer self-efficacy: Development of a measure and initial test. *MIS Quarterly, 19*(2), 189-211.

Eglash, R., Gilbert, J. E., & Foster, E. (2013, July). Broadening Participation - Toward Culturally Responsive Computing Education. *Communications of the ACM, 56*, 33-36.

Franke, R. H., & Kaul, J. D. (1978). The Hawthorne experiments: First statistical interpretation. *American Sociological Review 43*, 623-643.

Frieze, C., & Quesenberry, J. L. (2013). *From difference to diversity: including women in the changing face of computing*. Paper presented at the Proceeding of the 44th ACM technical symposium on Computer science education, Denver, Colorado, USA.

Gottfried, A. (1990). Academic intrinsic motivation in young elementary school children. *Journal of Educational Psychology, 82*, 525-538.

Hurd, S. (2008). Affect and strategy use in independent language learning. In S. Hurd & T. Lewis (Eds.), *Language learning strategies in independent settings* (pp. 218-236). Bristol: Multilingual Matters.

Kao, G. (1995). Asian Americans as model minorities? A look at their academic performance. *American Journal of Education, 103*, 121-159.

Klawe, M. (2002, June). Girls, Boys and Computers. *SIGCSE Bulletin Special Issue on Women and Computing, 34*, 16-17.

Kort, N., Reilly, R., & Picard, R. (2001). *An affective model of interplay between emotion and leaning: reengineering educational pedagogy - building a learning companion.* Paper presented at the IEEE International Conference on Advaned Learning Technology, Madison, WI.

Kumar, A. N. (2006a). *Do female students feel differently than male students about using software tutors?* . Paper presented at the Frontiers in Education Conference (FIE 2006), San Diego, CA. http://pages.ramapo.edu/~amruth/r/c/fie/06/paper.pdf

Kumar, A. N. (2006b). *The Effect of Using Problem-Solving Tutors on the Self-Confidence of Students*. Paper presented at the 18th Annual Psychology of Programming Workshop (PPIG 06), Brighton, U.K. http://pages.ramapo.edu/~amruth/r/wkshp/ppig/06/paper.pdf

Kumar, A. N. (2007). *Software Tutors Help Female Students Learn Programming Concepts Just as Well as Male Students*. Paper presented at the E-LEARN 2007, Quebec City, Canada. http://pages.ramapo.edu/~amruth/r/c/elearn/07/paper.pdf

Kumar, A. N. (2008a). *The Effect of Using Problem-Solving Software Tutors on the Self-Confidence of Female Students*. Paper presented at the 39th SIGCSE Technical Symposium, Portland, OR. http://pages.ramapo.edu/~amruth/r/c/sigcse/08/paper.pdf

Kumar, A. N. (2008b). *Female Students Assess Software Tutors More Positively Than Male Students.* Paper presented at the Frontiers in Education Conference (FIE 2008), Saratoga Springs, NY. http://pages.ramapo.edu/~amruth/r/c/fie/08/paper.pdf

Kumar, A. N. (2009a). *Need to Consider Variations within Demographic Groups When Evaluating Educational Interventions*. Paper presented at the Innovation and Technology in Computer Science Education, Paris, France. http://pages.ramapo.edu/~amruth/r/c/iticse/09/paper.pdf

Kumar, A. N. (2009b). *Patterns in Student Assessment of Problem-Solving Software*. Paper presented at the Frontiers in Education Conference, San Antonio, TX. http://pages.ramapo.edu/~amruth/r/c/fie/09/paper.pdf

Kumar, A. N. (2011). *Results from Repeated Evaluation of an Online Tutor on Introductory Computer Science*. Rapid City, SD.

Kumar, A. N. (2012). *A Study of Stereotype Threat in Computer Science*. Haifa, Israel.

Kumar, A. N., & Kaczmarczyk, L. C. (2013). *Programming Tutors, Practiced Concepts, and Demographics*. Oklahoma City, OK.

Marakas, G. M., Yi, M. Y., & Johnson, R. D. (1998). The multilevel and multifaceted character of computer self-efficacy: Toward clarification of the construct and an integrative framework for research. *Information Systems Research, 9*(2), 126-163.

Margolis, J., & Fisher, A. (2002). *Unlocking the clubhouse: women in computing*: MIT Press.

Peckham, J., Harlow, L. L., Stuart, D. A., Silver, B., Mederer, H., & Stephenson, P. D. (2007). *Broadening participation in computing: issues and challenges*. Paper presented at the Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, Dundee, Scotland. http://dl.acm.org/citation.cfm?doid=1268784.1268790

Picard, R. W., Papert, S., Bender, W., Blumberg, B., Breazeal, C., Cavallo, D., . . . Strohecker, C. (2004). Affective learning - a manifesto. *BT Technology Journal, 22*(4), 253-269.

Sears, J. (1999). Teaching queerly: Some elementary propositions. . In W. J. Letts & J. Sears (Eds.), *Queering elementary education: Advancing the dialogue about sexualities and Schooling* (pp. 97-110). Lanham, MD: Rowman & Littlefield, Inc.

Steele, C. M., & Aronson, J. (1995). Stereotype threat and the intellectual test performance of African Americans. *Journal of Personality and Social Psychology, 69*, 797-811.

Trauth, E. M., Cain, C. C., Joshi, K. D., Kvasny, L., & Booth, K. (2012). *Embracing intersectionality in gender and IT career choice research.* Paper presented at the Proceedings of the 50th annual conference on Computers and People Research, Milwaukee, Wisconsin, USA. http://dl.acm.org/citation.cfm?doid=2214091.2214141

Linking Linguistics and Programming: How to start? (Work in Progress)

J. E. Rice[1], I. Genee[2], and F. Naz[3]

[1] Dept. of Math and Computer Science, University of Lethbridge, Canada j.rice@uleth.ca
[2] Dept. of Modern Languages, University of Lethbridge, Canada inge.genee@uleth.ca
[3] Dept of Math and Computer Science, University of Lethbridge, Canada fariha.naz@uleth.ca

Abstract. Sociolinguistics is described as "the study of how language and social factors such as ethnicity, social class, age, gender, and educational level are related". Social factors may result in differences in language use, which may then be referred to as sociolinguistic differences. The goal of this project is to answer the following question: is it possible to identify sociolinguistic differences in the way people use programming languages? How might we go about answering our question? We propose that techniques from corpus linguistics may be of use. Computer-based techniques such as text mining allow the use of software in the identification of trends in language use that would otherwise require an enormous amount of human effort to discover. Once various sociolinguistic differences have been identified by computational means, traditional critical analysis methods can then be applied for further analysis.

1 Introduction

In 1982 Misek-Falkoff introduced a new area of research that she referred to as "Software Linguistics" [1]. She proposed treating computer programs as text, and described the use of "natural language textual criticism" as applied to software. To our knowledge this is the first work to suggest that the use of linguistics principles as applied to computer programs could provide useful information. She did not follow up on this work, however, and since then it appears that linguistic analysis has been mainly restricted to natural languages (with a few exceptions such as [2] and [3]).

Our work proposes to build upon Misek-Falkoff's original notion of software linguistics and make use of tools, ideas, and concepts from linguistics in application to texts produced using artificial languages: in other words, computer programs, or code. In the remainder of this paper we discuss the potential of such an approach, the areas where it might apply, and provide some initial thoughts from the work that we have begun.

2 Context

2.1 Natural Languages and Sociolinguistics

Wardhaugh [4] defines a society as "any group of people who are drawn together for a certain purpose or purposes", and a language as "what the members of a particular society speak". We could then say that the community of programmers is a society drawn together for the purposes of creating software. In this particular society we could examine either the natural language used in the course of carrying out software creation, or the artificial languages used in the actual software creation. Studies have already examined ways in which natural language is used in this society, e.g. [5, 6]. We propose to instead examine how people use the programming languages themselves.

In natural language one might break prescriptive syntax rules e.g. in order to emphasize a point, or phrase a sentence in a somewhat ambiguous or redundant way. These choices are not desirable when writing a computer program, and one would hope that the option to do so was removed as a possibility when designing the programming language. Yet there is still a great deal of variation possible, for instance in the choice of identifier names for functions/methods

and/or variables, or simply in the way the problem has been decomposed. It is this variation that we are interested in and which may provide information about the society of programmers.

The major insight offered by modern sociolinguistics is that social variability may influence or determine linguistic variability. The (independent) social variables most exhaustively studied in sociolinguistics are age, gender and socio-economic status (SES). The (dependent) linguistic variables include phonetic variables such as the pronunciation of the (ing) sequence at the end of words like "'working" and "playing"; morphosyntactic variables such as the use of double negatives, and stylistic variables such as the use of indirect requests vs. direct commands and forms of address. However another, complementary view is that linguistic variability may in turn influence social variability; thus the language itself shapes the social relations [7, 8].

One area where these ideas can be examined is that of gender differences, and in fact researchers have developed computer-based techniques that show evidence of gender differences in natural language use, e.g. [9–12]. However, rather than implying a one-way causality ("because one is a man/woman one must speak and/or write in a particular way") many researchers have shown how the language is shaping the society [13]. In other words, researchers examine how language use by individuals or groups contributes to the formation of those groups, or the treatment of those individuals within the community under examination.

2.2 Digital Humanities/Corpus Linguistics

In areas where large quantities of text are available analysis of those texts can be carried out by computers, rather than by humans. This allows much faster assessment, and in some cases the identification of trends that would not otherwise have been found. Discourse and linguistic analysis can both be aided by computer-based approaches. For this to happen text samples must be collected from some source and then stored in a format that allows the analysis to take place. The samples are then often referred to as a corpus. This type of approach has been followed by Argamon and Koppel [9–11] as well as others such as [12] and [14].

Having seen the benefits in the humanities we are now proposing to turn these techniques around to focus again on computer science and the societies within this field. This idea fits specifically into the digital humanities area of critical code studies, an area which examines the social and cultural implications of computer code, focusing particularly on gender, race, and class [15]. Indeed, one question being addressed by critical code studies is "How do issues of race, class, gender and sexuality emerge in the study of source code?" [16] We believe that as we attempt to address sociolinguistic categorization of source code we will be contributing knowledge towards this question, as well as furthering the body of knowledge surrounding how people use these types of languages.

2.3 Meaning to Society

Overall what does this project mean to society? Who cares if we can identify whether sociolinguistic differences exist in the use and creation of programming languages? The answer is that this research may have impact in a number of areas, from government to industry to private individuals. Identifying sociolinguistic differences can allow us to infer information about society, the groups, and/or the individuals using these languages – and here we are talking about programming languages. Programming languages could arguably be considered some of the most powerful languages on earth, given that the software that drives our daily lives is written using these languages. It is imperative that we learn more about their use and the groups of people using them; we want to know what this information can tell us in regards to the social and cultural implications of computer programs, and by extension, the societies that work with them.

Moreover, this research can contribute to the comprehension of computer programs; we may be able to identify markers, trends or beacons (e.g. [17]) that can enhance understanding

of programs, and patterns in their use. This information could then be utilised to teach students to write "good" programs, and also to use different approaches to teaching programming, approaches that might appeal to wider groups than are currently being reached.

3 Directions

The question is how to go about this. We are particularly interested in the work by Argamon et al. [9–11], partly because their approach was successfully applied to text samples in more than one (natural) language. Their text categorization approach, based on machine learning algorithms, achieved more than 80% accuracy in categorizing fiction and non-fiction samples according to author gender, and identified previously unknown patterns in male and female language use. Thus, could we not borrow these techniques for classification of programming language samples? In order to do so we must follow the components of text categorization [9]:

Document Representation: *choose a large set of text features which might be useful for categorizing a given text (typically words that are neither too common nor too rare) and represent each text as a vector consisting of values representing the frequency of each feature in the text.*

Dimension Reduction: *optionally, use various criteria for reducing the dimension of the vectors – typically by eliminating features which don't seem to be correlated with any category*

Learning Method: *use some machine learning method to construct one or more models of each category.*

Testing Protocol: *use some testing protocol to estimate the reliability of the system.*

Thus the first problem we must address is document representation.

3.1 Can we tag code with existing POS-taggers?

As described above, document representation consists of choosing a large set of features which might be useful for categorization. In Koppel et al.'s work, the features include 405 function words which appear at least once in the British National Corpus (BNC), and n-grams of parts-of-speech and punctuation marks. All 405 function words plus the 100 most common triples, 500 most common ordered pairs, and 76 parts of speech were used as features, for a total of 1081 features. The documents were then represented as a vector of length 1081, with each entry representing the number of appearances in the document for that particular feature. Thus we need a list of features applicable to code. Can we use an automated part-of-speech (POS) tagger on code samples to generate similar information? We've selected C++ as our programming language of choice, although this discussion could equally be applied to any programming language. In Figure 1 we've run a simple code sample through the CST's POS tagger[1]. NN means

```
main()
{
    int first_number = 1;
    int second_number = 3;
    int sum_of_numbers;
    sum_of_numbers=first_number + second_number;
    cout<<"Sum is = "<<sum_of_numbers<<endl;
}
```

(a)

```
main()/NN {/( int/NN first_number/NNP =/NNP 1/CD ;/:
int/NN second_number/NNP =/NNP 3/NNP ;/:
int/NN sum_of_numbers/NNS ;/:
sum_of_numbers=first_number/NN +/SYM second_number/NN ;/:
cout<<"Sum/NN is/VBZ =/NNP "/" <<sum_of_numbers<<endl/NNP ;/:
}/)
```

(b)

Fig. 1. (a) Untagged code sample. (b) Code sample with POS tagging from CST.

that the preceding token is a noun, NNP refers to a proper noun, NNS refers to a plural noun, CD refers to a cardinal number, VBZ refers to a verb (3rd person present) and SYM refers to a symbol. So we can see that our program is mostly nouns, including, strangely enough,

[1] Center for Sprogteknologi, University of Copenhagen, http://cst.dk/tools/index.php

the "=" sign and number 3. This tool also separated the text into segments (one per line) and tokens (separated by spaces), although these segments and tokens don't necessarily match with what a programmer might identify as relevant segments and/or tokens. A more complex sample generated labelings identifying adverbs and preposition/subordinating conjuctions. So how useful is this? One problem is that the tool does not ignore white space, so " = " with spaces around it is treated differently than the same symbol with no whitespace surrounding it. Another problem is the treatment of data inside quotes, which was broken up by the tagger, as was the "!=" symbol, so the tagger is not tokenizing the code correctly. This could easily be corrected, however, given that these programs are written in language designed to be parsed by a computer! A possible larger problem is that "`int`" is labeled as a noun, when maybe it should be an adjective given that it is being used to describe what type of variable or function is given following, and it is unclear why "`val`" and "`int(val[i])-int(0)`" were labeled as adverbs given the apparent lack of nearby verbs.

Although this is an interesting exercise, it is evident that a straightforward (brute-force) use of automated unmodified POS taggers on code is unlikely to be a useful way to go about tagging the constituent parts of a program for classification or analysis, given that little to no thought was put into how the "words" in a program are behaving.

3.2 Beginning the feature list

So we have ruled out automatically tagging code with POS-labels. In fact, we may want to consider also that the POS labels themselves may not apply at all. Since the elements of a programming language are already categorized, we can begin with a list of the elements to be found in a C++ program[2]. A C++ parser within the compiler will recognize these types of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. An identifier is a sequence of characters (word?[3]) that denotes a "thing"; this might be an object, variable, class, member of a class, or macro. Keywords are predefined identifiers that have special meanings which cannot be redefined. Punctuators are used to separate values or other tokens, while an operator appears as part of an expression, which is a sequence of operators and operands that appears for the purpose of computing a value, designating an object or function, or modifying the value of an object (this might take place as a side effect of computing a value). Operators include mathematical and logical operators that we might already be familiar with (e.g. `+ - * / < >`), combinations and repetitions of these (e.g. `+= -= *= /= ++ --`), and others such as array subscripts, scope resolution operators and class/pointer operators that resolve which part of a class variable (object) is being manipulated. Literals consist of invariant program data. These may be numeric values, characters or string constants, and may appear as 1 (numeric literal), `'c'` or `'\0'` (both are character literals), or "Hello!" (string literal).

3.3 Text Categorization

We have begun some preliminary work to test whether it is possible to use some of these items to represent a document, and then use a learning method to construct a model based on these items. We've created a program that identifies some very basic items: two data types (int, float); main functions, cout statements, and `{}` symbols. We then generated, for 10 sample programs, the metrics indicating how many times each of these items appeared in sample and produced a representation of each sample based on these metrics. We were then able to train, using a

[2] The following information can be found in any C++ reference. We used the descriptions and information in the C++ language reference available from the Microsoft Developer's Network (http://msdn.microsoft.com/en-us/library/3bstk3k5.aspx).

[3] What is a "word", in a program, and what information is that "word" conveying? [3] explored this discussion somewhat, deciding that "Together keywords, operators, and identifiers constitute the set of tokens which may be considered *words* in a programming language vocabulary."

support vector machine (SVM) implementation as described in [18], our system to identify categories within these samples. This was a little surprising to us given that we had so few samples to train on and that we were using such basic items; however it was a very promising result as a proof-of-concept.

3.4 Moving on

Our next steps are to add more features to use in categorization, and gather additional samples along with corresponding sociological data. Our current set of samples do not provide sufficient data for e.g. categorization according to author gender or first language learned (natural), although we can examine variables such as years of experience and possibly first language learned (artificial). Concurrently with this we must expand our feature list to incorporate a more complete list of the known C++ program elements.

3.5 Other Considerations

We had originally intended to determine a "code to POS mapping" and generate a feature set based on this, but as discussed above it seems reasonable to start with the known elements of our artificial language(s) and attempt text categorization with these. However it is still interesting to examine these elements and consider how they might map to what we call "parts of speech". For instance, literals are in most cases the equivalent of nouns, or more generally, "things" (rather than actions). Rather than attempt to find direct mappings to parts of speech, it might be more useful to generally categorize each token into "thing" words, "modifying" words, "action" words and so on. Identifiers are tricky, as they refer to anything a programmer needs to label with a name, including memory storage and functions. Depending on the context these could be "things" or "actions". If we were to use this type of labeling then the standard libraries would provide many identifiers that could likely be pre-labeled (e.g. cout). Finally, there are several keywords, reserved for use by the C++ language. These are quite varied. For instance, the keyword `false` is a logical literal (and so, likely to be considered a "thing") while the keyword `return` forms part of a return statement indicating the value a function will return when called, e.g. `return false;` – the behaviour of this keyword then is more in the nature of an action.

This approach could be valuable if we are to try to link our findings to those in the literature for natural language text categorization. For example [9] states that "[t]he picture that emerges is that the male indicators are largely noun specifiers (determiners, numbers, modifiers) while the female indicators are mostly negation, pronouns and certain prepositions". If we can categorize code samples according to gender author, might we see similar patterns?

4 Conclusion

In this paper we have explored the need for a sociolinguistic analysis of computer programs. We've pointed out that software is such a huge part of our daily lives that it really makes sense to understand how different groups might make differing uses of the artificial languages that are used to create programs. However, this isn't easily done. While we can leverage the approaches used in application to natural languages, the tools will likely need to be at the very least modified, if not developed from the ground up. To provide some guidance and structure to our problem we are leveraging approaches known to work in natural languages, and that have the potential to be altered (or trained) for additional languages. However identifying useful and meaningful features to work with and understanding fully how to apply these approaches in this new setting are large projects which we have only just begun researching. This paper offers both a methodology to follow in this investigation, plus an initial and tentative set of features that we hope to refine and then use in training learning algorithms in categorization of code samples according to sociological variables.

References

1. L. D. Misek-Falkoff. The new field of "software linguistics": An early-bird view. *SIGMETRICS Performance Evaluation Review*, 11(2):35–51, August 1982.
2. Abram Hindle, Earl Barr, Mark Gabel, Zhendong Su, and Prem Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE Press Piscataway, NJ, USA, 2012.
3. D. P. Delorey, C. D. Knutson, and M. Davies. Mining programming language vocabularies from source code. In *21st Annual Psychology of Programming Interest Group Conference - PPIG*, 2009.
4. R. Wardhaugh. *An Introduction to Sociolinguistics*. Blackwell Publishers, Oxford, UK, 6th edition, 2010.
5. F. Taïani, P. Grace, G. Coulson, and G. Blair. Past and future of reflective middleware: Towards a corpus-based impact analysis. In *Proceedings of the 7th Workshop on Reflective and Adaptive Middleware (ARM)*, pages 41–46. ACM, 2008.
6. Nicolas Bettenburg and Ahmed E. Hassan. Studying the impact of social structures on software quality. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '10)*, pages 124–133. IEEE, 2010.
7. W. Labov. *Principles of Linguistic Change, Cognitive and Cultural Factors*. Language in Society. Wiley, 2011.
8. J.K. Chambers. *Sociolinguistic Theory: Linguistic Variation and Its Social Significance*. Language in Society. John Wiley & Sons, 2003.
9. M. Koppel, S. Argamon, and A. Shimoni. Automatically categorizing written texts by author gender. *Literary and Linguistic Computing*, 17(4):401–412, 2002.
10. Shlomo Argamon, Moshe Koppel, Jonathan Fine, and Anat Rachel Shimoni. Gender, genre, and writing style in formal written texts. *TEXT*, 23:321–346, 2003.
11. Shlomo Argamon, Jean-Baptiste Goulain, Russell Horton, and Mark Olsen. Vive la différence! text mining gender difference in french literature. *Digital Humanities Quarterly*, 3(2), 2009.
12. M. L. Newman, C. J. Groom, L. D. Handelman, and J. W. Pennebaker. Gender differences in language use: An analysis of 14,000 text samples. *Discourse Processes*, 45:211–236, 2008.
13. Mary M. Talbott. *Language and Gender*. Polity Press, 1998.
14. Erik Linstead, Lindsey Huges, Cristina Lopes, and Pierre Baldi. Exploring Java software vocabulary: A search and mining perspective. In *Proceedings of the ICSE Workshop on Search-Driven Development–Users, Infrastructure, Tools and Evaluation (SUITE)*, Vancouver, BC, 16-19 May, 2009.
15. Mark C. Marino. Critical code studies (entry from the blog electronic book review), 2006. http://www.electronicbookreview.com/thread/electropoetics/codology.
16. HASTAC (Humanities, Arts, Science and Technology Alliance and Collaboratory) Scholars program. Critical code studies (blog entry), 2011. https://www.hastac.org/forums/hastac-scholars-discussions/critical-code-studies.
17. J. F. Pane and B. A. Myers. Usability issues in the design of novice programming systems. Technical Report CMU-CS-96-132, Carnegie Mellon University, Pittsburgh, PA, aug 1996.
18. Christopher J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

Concept Vocabularies in Programmer Sociolects (Work in Progress)

J. E. Rice[1], B. Ellert[2], I. Genee[3], F. Täiani[4], and P. Rayson[5]

[1] Dept. of Math and Computer Science, University of Lethbridge, Canada j.rice@uleth.ca
[2] School of Computing Science, Simon Fraser University, Canada bellert@sfu.ca
[3] Dept. of Modern Languages, University of Lethbridge, Canada inge.genee@uleth.ca
[4] IRISA, Université de Rennes 1, France francois.taiani@irisa.fr
[5] School of Computing and Communications, UCREL, Lancaster University, UK p.rayson@lancaster.ac.uk

Abstract. The code a programmer writes plays a key role in communicating the intent and purpose of that code. However little is known about how this process is influenced by sociological factors. Does a programmer's background, experience, or even gender affect how they write computer programs? Understanding this may offer valuable information to software developers and educators. In this initial phase of research we focus on experience and writing, while upcoming phases will incorporate reading and comprehension. In this paper we discuss an early experiment looking at how years of programming experience might influence identifier formation.

1 Introduction

Large software engineering projects require many individuals to collaborate on the same code, and ideally everyone involved should have an equally clear understanding of the code. Unfortunately this can be problematic [1]. Both natural language communication, e.g. during the design phase or in accompanying documentation, and artificial language communication, e.g. communication through programming languages, must be considered, particularly as project complexity increases. Increased project complexity also means that up-to-date documentation may not be available, and so code may be the primary communication tool. Clearly it is important to know what factors might impact a programmer's coding choices. The question of how people use artificial (programming) languages is very broad, and so we have begun with a small examination of how people choose identifiers as they are writing programs, particularly in the choice of identifier names when compared to the known vocabulary of words surrounding a particular project. Subsequent phases will broaden this investigation to include broader aspects of language use, and as well examine how varying choices can impact comprehension.

2 Background

2.1 Variation in Programming

Although a programmer is not permitted to break the syntax rules when developing a program, he or she is still allowed significant variability including the choice of several equivalent operators (or combinations) for carrying out computations; the method or approach for breaking down the problem; and the choice of names for identifiers. We are interested in all of these variations; however at this early stage of the research we have restricted ourselves to examining identifier naming conventions.

Unlike categorical approaches to natural language studies, the comparatively new field of sociolinguistics examines linguistic variability. Sociolinguistic studies have shown that sociological variables such as gender, age, socioeconomic status, and register (the relative formality of the situation) systematically correlate with linguistic variables, e.g. [2]. Sociolinguists examine what circumstances affect the production of different and equally viable methods of expression by speakers belonging to different social groups. When such variability becomes ingrained in a specific speech community, this gives rise to various "lects" such as dialects, ethnolects, sociolects, genderlects, and idiolects.

When looking at linguistic variables, computer code provides much less room for variation than natural language. Possibly because of this identifier names act as a method of annotation for the coder. For the parser and compiler, the choice of name is irrelevant as long as it is unique. Where the name choice matters is in the documentary structure to help explain the code to the reader.

2.2 Connections with Quality & Comprehension

As a project matures it generally becomes more complex as varying authors contribute more concepts and possibly more styles of expression [3]. By investigating how styles vary between people, it may be possible to introduce standards to reduce this variation, thus improving comprehensibility. For instance [4] has suggested that using descriptive and consistent identifier names can improve comprehension.

Overall program quality is harder to accurately define than is comprehensibility. Although both have levels of subjectivity, comprehensibility can be entirely defined based on perception. If the majority agrees that a program is comprehensible then it is comprehensible. On the other hand, program quality has a certain amount of objective fact that is difficult to pinpoint. One way to approximate this is by counting the number of software bugs. High "program quality" (low bug count) has been correlated with high "identifier name quality" (use of dictionary-based words, consistent capitalization, concise and consistent naming, appropriate lengths of names, and no type encoding) [5].

These findings can also be used to help improve code as it is being written. E.g. tools have been developed to assist in choosing identifier names based on a set of consistent identifier naming conventions. As reported in [6] programmers were tasked with coming up with names without assistance that were subsequently checked against the recommendations. The participants were rarely able to come up with acceptable names without assistance, where "acceptable" considered important factors such as consistency.

2.3 Defining the Word

Current studies in identifier naming conventions focus on linking them to their sources in natural language, e.g. [7]. These previous studies do not, however, use cues such as camelcase or underscores to break up the identifier names. This should be done to avoid problems that come about when coders write something as "one word" when it should be "two words" (e.g. dataset). Of course a large problem lies in determining an accurate definition of what constitutes a word.

Linguistic theory avoids the notion of a word being just a string of characters, the boundaries of which are defined by whitespace and punctuation [8]. One reason for this avoidance is that this definition breaks down when looking at languages with different conventions, and even within English. In addition trying to force identifier names to fit arbitrary English spacing conventions results in the loss of interesting distinctions. This problem was also addressed in [9]. Instead, providing a comparison between identifiers in programs and those used in the libraries referenced by those programs allows us to determine what is common and standard. In this way domain-specific abbreviations that cannot easily be linked to natural language correlates are not viewed as problem cases or bad practice, but rather an integral part of how programmers choose to name their identifiers. Just as the speech community shapes natural language, the programming community shapes artificial language.

3 Methodology

The goal of this study was to investigate what words individuals chose as identifiers. We hoped to discern patterns in these choices that could be linked to sociological variables. This might tell us something about how different groups used this aspect of programming languages.

Since the scope of this study was narrowed to identifier names, the approach from [10] provided a nice basis for our methodology, which is centered around shifting the paradigm of studying identifier names away from its reliance on the natural languages on which they are based. By treating this as a mode of communication separate from *both* computer code and natural language we hope to obtain a clearer picture of an individual's naming "style" and prevent losing distinctions between extremely similar identifiers.

3.1 Extracting and Parsing Identifiers

Figure 1 (a) presents the pseudocode for extracting identifier names from C++ code. Lines 1–5 reduce the code to identifiers, reserved keywords, and whitespace. Line 6 removes all duplicates and whitespace. Line 7 leaves just the identifiers, and lastly, the identifiers are output in the format required by the parsing script.

(a)

```
1 remove comments
2 remove string literals
3 remove character literals
4 remove preprocessor directives
5 remove non-alphanumeric characters
6 put tokens into set
7 remove reserved keywords
8 output identifiers
```

(b)

```
1 put library concepts into set
2 put sample concepts into set
3 take set difference
4 output set sizes
```

Fig. 1. (a) Extracting identifier names. (b) Comparing concept sets.

This output is then fed into the scripts utilized in [10]. Class names are parsed into "concepts" (or words), where the parsing is performed solely on the assumption of camelcasing and underscoring conventions. That is, camelcasing and/or underscoring is assumed to indicate that there are multiple concepts of interest within the identifier, and so it is broken up accordingly. By letting the programmer's demarcation of identifiers determine the concepts instead of using external sources, a more *descriptive* rather than *prescriptive* analysis can be taken. We believe that how a coder chooses to break up an identifier speaks to how the concepts are viewed. As the concepts are extracted from the list of identifiers, a cumulative sum representing the number of unique identifiers each concept occurs in is recorded. This basic process can be used to evaluate a coder's choices in identifier naming by seeing how many concepts are drawn from source libraries.

3.2 Comparing Concept Vocabularies

To see how each sample compares to each other, a "standard" must be defined as a base point. By using the domain-specific libraries that all the samples from within a particular group reference a common ground can be found. The same procedure for extracting identifiers and parsing them into concepts was run on the library code to form a basis for a standard vocabulary. Figure 1 (b) presents the pseudocode for comparing an individual's concepts to the standard. Lines 1 and 2 import the concept lists to be compared into sets. Lines 3 and 4 output how many concepts are in the sample that are not in the library. A simple set difference removes concepts from the sample that are in the library and the number remaining gives this result.

4 Results

From a fourth year image processing class at the University of Lethbridge, 40 samples of C++ code were drawn from 14 students, listed in Table 1. Each contributed three files, except for participant 13 who only contributed one. All were working on the same set of assignments using

the library Netpbm[1].The Netpbm source files were concatenated to provide one standard source to draw concepts from. While giving consent, each student undertook a short sociological survey asking for name, gender, first spoken language (NL1), first programming language (AL1), years of programming experience, whether the assignment was group work or not, and whether the code was planned to be reused or not.

Table 1. Participants.

PID	Gender	NL1	AL1	Experience	Reuse
1	M	English	BASIC	5 years	Y
2	F	English	C++	2 years	N
3	M	Chinese	C	10 years	Y
4	M	English	TI-BASIC	5 years	N
5	M	Hungarian	Pascal	7 years	N
6	M	English	C++	2 years	N
7	M	English	C++	2.5 years	N
8	M	English	Java	4 years	Y
9	M	English	C++	5 years	N
10	M	English	C	4 years	Y
11	M	English	ACS	5 years	N
12	M	English	C++	1.5 years	N
13	M	English	C++	4 years	Y
14	M	English	Java	7 years	N

For each sample, the percentage of library-external concepts was calculated as shown in Table 2. As expected, the majority of the concepts used were also used in the library. Figure 2

Table 2. Samples.

PID	File	Concepts	External	Percent	Experience
12	1	47	4	8.51%	1.5 years
12	2	62	6	9.68%	1.5 years
12	3	46	6	13.04%	1.5 years
2	1	44	4	9.09%	2 years
2	2	43	5	11.63%	2 years
2	3	60	10	16.67%	2 years
6	1	51	5	9.80%	2 years
6	2	48	7	14.58%	2 years
6	3	63	10	15.87%	2 years
7	1	50	4	8.00%	2.5 years
7	2	48	4	8.33%	2.5 years
7	3	42	4	9.52%	2.5 years
8	1	58	4	6.90%	4 years
8	2	46	4	8.70%	4 years
8	3	52	13	25.00%	4 years
10	1	62	5	8.06%	4 years
10	2	54	5	9.26%	4 years
10	3	71	11	15.49%	4 years
13	1	50	5	10.00%	4 years
1	1	68	8	11.76%	5 years

PID	File	Concepts	External	Percent	Experience
1	2	56	8	14.29%	5 years
1	3	51	9	17.65%	5 years
4	1	53	3	5.66%	5 years
4	2	48	6	12.50%	5 years
4	3	60	9	15.00%	5 years
9	1	45	4	8.89%	5 years
9	2	55	6	10.91%	5 years
9	3	45	5	11.11%	5 years
11	1	49	2	4.08%	5 years
11	2	50	5	10.00%	5 years
11	3	50	6	12.00%	5 years
5	1	70	12	17.14%	7 years
5	2	70	15	21.43%	7 years
5	3	79	17	21.52%	7 years
14	1	62	5	8.06%	7 years
14	2	21	3	14.29%	7 years
14	3	31	5	16.13%	7 years
3	1	40	1	2.50%	10 years
3	2	39	1	2.56%	10 years
3	3	38	2	5.26%	10 years

shows a plot of programming experience against percent of out-of-library concepts. The size of each bubble represents the total number of concepts. Samples from the same participant are grouped by colour, so each circle represents one participant file. A linear regression produces an r value of -0.126256 and an R-squared value of 0.015941. The negative r value means that any potential correlation is negative. This means that as programming experience increases, less out-of-library concepts are used, which is what should be expected. The low R-squared value

[1] available from ftp://ftp.wustl.edu/graphics/graphics/packages/NetPBM

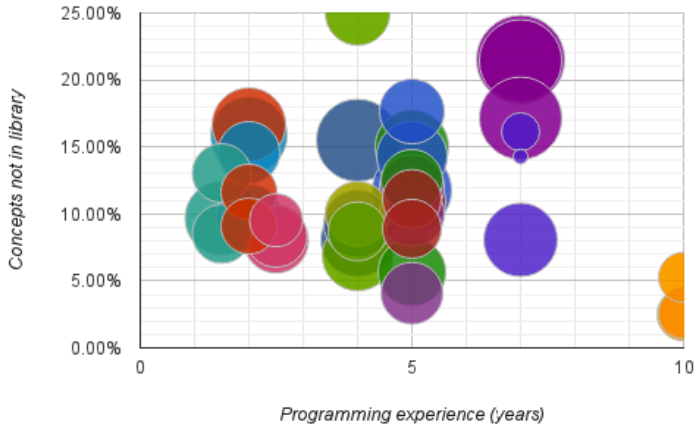


Fig. 2. Programming experience vs concept use.

means that less than 2% of the variation is explained by a linear regression. In other words, the data may as well have been random and there is no reason to say it is correlated.

5 Discussion

Although the current results are quantitatively inconclusive, there are various directions of thought that can be pursued. We are working on creating the groundwork for a basic paradigm of viewing identifier naming conventions. It is important to make a distinction between an individual's concept vocabulary and the natural language vocabulary on which it is based. Making this distinction allows questions such as whether the concept vocabularly is influenced in a similar way as the natural language vocabulary, for instance by one's mother tongue.

Metrics We are currently using multiple samples from each individual; however it may be better to concatenate each individual's code into one large sample beforehand as was done for the library. This would allow grouping of the concepts and prevent potential bias. Additionally, the number of times a concept appears in a sample has not been utilized (we use a cumulative sum representing the number of unique identifiers containing each concept). This statistic could lead to some interesting results by giving weight to each sample's concept use instead of just doing a simple count. This might give insight into a programmer's thought processes, as this could reflect the (perceived) importance of particular concepts. Finally, instead of computing a set difference versus the standard, each individual could be compared to each other or to the entire population to generate another metric for similarity. This, combined with interviews, could determine whether our existing metric could be used to predict how easily another individual (or group) might understand code from a different individual (or group).

Qualitative Analysis It might also be useful to qualitatively examine the samples, given that we have a relatively small sample set at this point. Examination by hand could lead to insights as to how identifiers are being created, and analysis as simple as outputing the concepts in the set created after taking the set difference would allow us to identify what specific out-of-library concepts individuals are using. Following this, checking whether there are any trends between the samples may lead to some interesting results. By grouping individuals together based on their identifier naming style, it may be possible to develop teaching approaches for more directed instruction. Another possible approach could be to look at the specific in-library concepts. For example, we could look at the top ten library concepts reused and see if this correlates to any sociological variables.

Data Collection More data is required before we can draw any conclusions from quantitative analysis approaches. Student data collection has proven to be more difficult than anticipated;

however using online repositories (e.g. open source) leads to the problem of determining the sociological data. We currently have restricted our samples to code from projects that are related, i.e. work based on a class assignment, and this was done in order to ensure that the same libraries were used in all the samples in order that we could use those libraries as a standard. However it might be interesting to lift this restriction, as long as some basis for a standard identifier naming source could be identified.

The initial aim of this study was to investigate the effects of gender on coding, in the hope that this may lead to a better understanding of the underrepresentation of women in the field. Unfortunately the underrepresentation prevented the collection of enough data from female participants to draw any statistically significant conclusions, and this is a problem we are still struggling with.

6 Conclusions

This paper presents our initial work in examining identifier naming styles, with a goal towards linking this to sociological variables. We believe that understanding how groups within the programming society use artificial language to communicate will provide interesting and enlightening information with applications in software engineering, software quality, and computer science education. In other fields researchers are "... committed to examining the way language contributes to social reproduction and social change" [11], and have shown that people "... actively choose ways of framing to accomplish specific ends within particular interaction. These choices are drawn, in part, from sociocultural norms..." [12]. Given the reliance that the world now has on computer programs, we hope that this study may offer a step towards similar research focusing on artificial languages. Our preliminary study was quantitatively inconclusive; however we have offered a methodology to continue with this work, and several avenues for other directions that we hope to pursue.

References

1. R. N. Charette. Why software fails [software failure]. *IEEE Spectr.*, 42(9):42–49, September 2005.
2. W. Labov. *Principles of Linguistic Change, Cognitive and Cultural Factors.* Language in Society. Wiley, 2011.
3. A. Mohan, N. Gold, and P. Layzell. An initial approach to assessing program comprehensibility using spatial complexity, number of concepts and typographical style. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 246–255, 2004.
4. S. Blinman and A. Cockburn. Program comprehension: investigating the effects of naming style and documentation. In *Proceedings of the Sixth Australasian User Interface Conference (Vol. 40)*, AUIC '05, pages 73–78, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
5. S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CMSR)*, pages 156–165, Washington, DC, USA, 2010. IEEE Computer Society.
6. P. A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering*, November 2005.
7. Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, page 79–88, New York, NY, USA, 05/2008 2008. ACM, ACM.
8. Dee Gardner. Validating the construct of word in applied corpus-based vocabulary research: A critical survey. *Applied Linguistics*, 28(2):241–265, 2007.
9. D. P. Delorey, C. D. Knutson, and M. Davies. Mining programming language vocabularies from source code. In *21st Annual Psychology of Programming Interest Group Conference - PPIG*, 2009.
10. François Taiani, Jackie Rice, and Paul Rayson. What is middleware made of?: exploring abstractions, concepts, and class names in modern middleware. In *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware*, ARM '12, pages 6:1–6:6, New York, NY, USA, 2012. ACM.
11. Mary M. Talbott. *Language and Gender.* Polity Press, 1998.
12. S. Kendall and D. Tannen. Gender and language in the workplace. In R. Wodak, editor, *Gender and Discourse.* Sage, London, 1997.

Developing Coding Schemes for Program Comprehension using Eye Movements

Teresa Busjahn Carsten Schulte Edna Kropp

Department of Computer Science
Freie Universität Berlin
{teresa.busjahn, carsten.schulte, edna.kropp}@fu-berlin.de

Keywords: POP-II.B. program comprehension, POP-III.B. java, POP-V.B. eye tracking, POP-VI.F. exploratory

Abstract

This paper introduces an approach to use eye movement data in the context of program comprehension studies. The central aspect is the development of coding schemes, which reflect cognitive processes behind the observable visual behavior of programmers. For this purpose, we discuss to first use a quantitative approach to find those episodes in the eye movements that yield the most potential for analysis. Subsequently, qualitative methods can be used on this subset.

1. Introduction

Tracking a programmer's gaze is probably one of the closest and most direct measurements we have to infer cognitive processes from behavioral and observable data. However, using eye movement data poses several challenges. Even though eye movements and cognitive processes are connected, the relation is complex and there is no easy matching. Moreover, eye tracking produces huge data sets. In this paper we discuss a structured approach to develop an analytical research instrument to study code reading and comprehension. Central for this approach is the development of coding schemes to label and aggregate eye movement data of programmers understanding source code.

As a start we will focus on eliciting program comprehension (PC) strategies from eye movement data. However, the presented approach is suitable for a multitude of questions, e.g. on the difference between novices and experts, the interaction of task type and comprehension process, influence factors like programming language and paradigm, program length, additional visualization, tools/interface issues, plan-like and un-planlike programs, and debugging.

The question we address here is "What strategies do expert programmers use during program comprehension?". We decided to look into experts first, since experts are supposed to have developed successful strategies which can be taught to less experienced programmers. Furthermore, we expect to find some established strategies that are shared by more than one individual.

In November 2013 the first international workshop on "Eye Movements in Programming Education: Analyzing the expert's gaze" was conducted as an attempt to broaden the knowledge about PC strategies. The focus was on cognitive processes behind observable eye movements during source code reading. The workshop was organized in association with the 13th KOLI CALLING Conference in Computing Education. Before the workshop, two sets of eye movement records of expert programmers reading Java were given to the participants.[1]

Participants were asked to analyze and code these records with a provided scheme containing code areas in different levels of detail, eye movement patterns and presumed comprehension strategies. Based on this analysis, the participants wrote position papers describing the eye movement data, commenting on the coding scheme, and possible applications of eye movement research in computer science education. The scheme was revised following suggestions given in the position papers and during the workshop.[2]

1 The data can be downloaded from http://www.mi.fu-berlin.de/en/inf/groups/ag-ddi/Gaze_Workshop/koli_ws_material.

In the following we will introduce a systematic approach to develop coding schemes for eye movement records of programmers in order to study PC processes - without getting lost in the data.

2. Using gaze data

2.1. Eye tracking in Program Comprehension

Eye tracking studies are offering a promising source of data for studying cognitive processes of programmers by 'showing' what is happening, without having to force the subject to think aloud. A problem with methods like think aloud is that they add an extra cognitive load besides the task at hand and therefore affect the comprehension process. Moreover, lots of program elements are never mentioned during a think aloud, even though they are important and taken into account by the programmer. This leads to an incomplete analysis of the mental state. In contrast, eye tracking provides the information which part of the program the subject was perceiving at exactly what point during understanding.

However, on a closer look some obstacles are occurring: The amount of data is bigger and more fine grained compared to e.g. verbal protocols. And also, it is not the thinking process itself that is made visible, but the process of reading something (in our case a program).

In the past, starting empirical studies in a new domain was often done using qualitative methods like Grounded Theory. While this approach seems useful in general, the complexity of eye movement data adds further challenges. In this approach, we suggest to take into account experiences from former eye tracking and PC studies to align the analysis process with the design of a central concept within this process, the coding scheme.

2.2. Eye Movements and Parameters

During reading, the eye stays on one point for a moment and then quickly jumps to the next location. These movements are called *saccades*, the relatively steady state between them *fixations*. When reading English, fixations usually last about 200-250 ms, but the duration can vary from under 50 ms to over 500 ms. The mean saccade length is 7-9 letter spaces, but can cover 1 to over 15 letter spaces. Saccades against the normal reading direction are called *regressions*. Typically, 10 - 15 % of saccades during reading are regressive. Good readers are characterized by short fixations and few regressions. However, these parameters depend on several factors like text difficulty and formatting. More demanding texts induce longer fixations, short saccades and frequent regressions (Rayner 1998).

3. Coding Schemes related to Program Comprehension

In qualitative research a code "is most often a word or short phrase that symbolically assigns a summative, salient, essence-capturing, and\or evocative attribute for a portion of language-based or visual data" (Saldaña 2012), p. 3. This data can have various forms, e.g. think-aloud transcripts, and video. Codes will most likely be used several times and form patterns. While coding, the data is organized and grouped into interrelated categories, which usually undergo refinements into different levels of subcategories. A further step is to compare and consolidate the categories to contribute to theory (Saldaña 2012). A coding scheme is an instrument that contains the possible codes and organizes them into categories.

3.1. A flexible expandable Coding Scheme for Software Comprehension

Von Mayrhauser and Lang (1999) describe a flexible expandable coding scheme (AFECS) to support the systematic analysis of PC. It bases on the integrated comprehension model[3] and is supposed to be consistent with accepted theories of PC. It was developed for protocol analysis on transcribed think aloud protocols reflecting programmer behavior.

2 The position papers and the full version of the coding scheme can be found in the technical report (Bednarik, Busjahn, & Schulte 2014) and at http://www.mi.fu-berlin.de/en/inf/groups/ag-ddi/Gaze_Workshop/.

3 See (Mayrhauser & Vans 1994) for a detailed description

The codes are systematically split into a number of segments, that encode particular aspects of a cognitive process. Starting point is the *mental model* (’program model’, ’situation model’, or ’domain model’). It reflects the level of abstraction at which a programmer is working. Programmers can start building a mental model at any level that appears opportune and switch between any of the three model components during comprehension. The second segment, *element* classifies what the programmer does at that level with the general notions of cognition ’goals’, ’hypotheses’, and ’actions that support the hypothesis driven understanding process’. These actions can be analyzed further and coded in segments with greater detail. Only the first two parts of the coding scheme (mental model and element) are mandatory.

The analysis proceeds from identifying actions of various types to determining action sequences and extracting cognition processes and strategies. The results can be used for statistical analyses, e.g. discovering patterns of cognitive behavior or analyzing frequencies of certain actions.

The coding scheme can be expanded or reduced according to the level of detail desired. Due to this flexibility, the scheme can be adjusted to answer a variety of research questions for various aspects of PC. Hence, researchers can tailor AFECS to their own needs instead of developing a coding scheme from scratch. Often results from different studies are difficult to compare. By using the same scheme, results maintain a degree of standardization and enable comparisons across studies.

This scheme is especially interesting in the context of our approach, as it directly provides a broad range of codes for cognitive processes during program comprehension.

3.2. An open-source Analysis Scheme for Identifying Software Comprehension Processes

O’Brien, Shaft, and Buckley (2001) compose an open analysis scheme for think-aloud protocols to determine the type of comprehension process used by programmers. This scheme extends the AFECS. It distinguishes between bottom-up and two variants of top-down comprehension. The first top-down type, *expectation-based* comprehension, goes back to Brooks. The programmer has a pre-generated hypothesis about the code’s meaning and then scans it for that hypothesis. The second type *inference-based* comprehension is based on Soloway. The programmer scans the code, and derives an hypothesis from the incomplete knowledge about that program. The hypothesis is then validated against the code. Bottom-up processing is described as an initial study of code, line by line, leading to a general understanding of the program.

This scheme fits into the AFECS framework, giving more detail to the segment called *hypothesis action*. If the hypothesis action is generating a hypothesis, then the new segment describes the trigger for this generation process. Being open, the scheme allows for subsequent refinement by other researchers and for replication of experiments. Furthermore, it includes the analysis procedure used to assign verbal data to the elaborated categories.

3.3. A Scheme for Analysing Descriptions of Programs

Good and Brna (2004) present a coding scheme for analyzing free-form program summaries, which allow programmers to express their understanding in their own words at their chosen level of abstraction, including as much detail as they feel is necessary. Pennington’s analysis schemes (Pennington 1987) was the starting point for developing this scheme. There are two kinds of classifications employed, *information types* and *object descriptions*.

The information types classification developed by Good and Brna contains 11 categories, e.g.

- function: the overall aim of the program, described succinctly
- actions: events occurring in the program which are described at a lower level than function, but at a higher level than operations
- control: information having to do with program control structures and with sequencing, e.g. recursion.

The object classification comprises how objects present in the program are described. There are seven object categories, e.g.

- program only: refers to items which occur only in the program domain, and which would not have a meaning in another context, like a counter
- program-domain: object descriptions which contain a mixture of program and problem domain references, e.g. a list of marks
- domain: an object which is described in domain terms, rather than by its representation within the program, e.g. a distance.

Possible analyses with this scheme are the proportion of information types used, and the level of abstraction featured in the summary. Good and Brna assume, that the scheme can also be applied on verbal protocols gathered during comprehension tasks.

3.4. The Base Layer

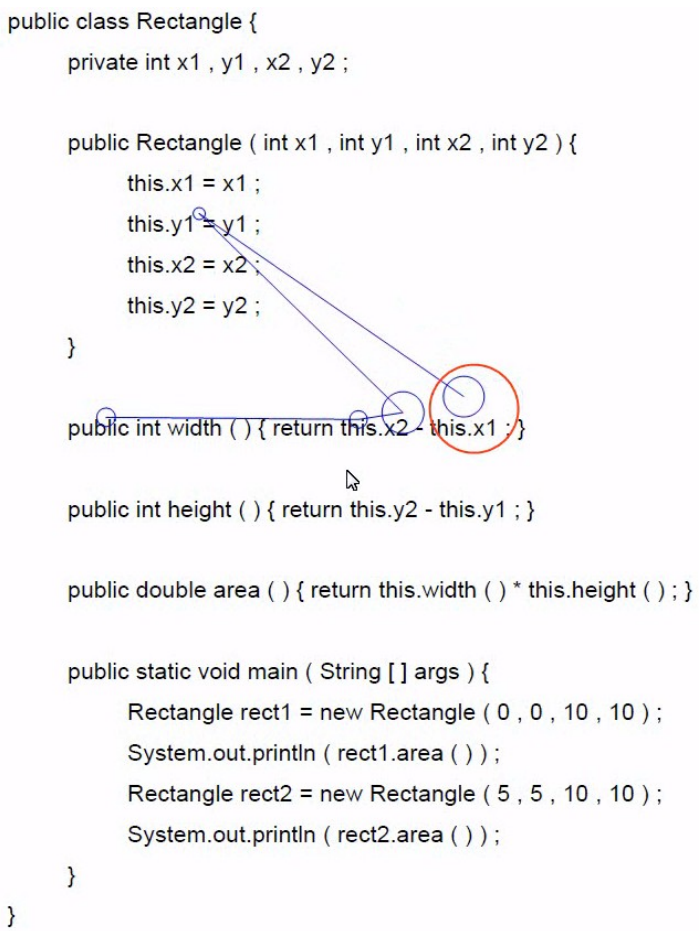
Based on Salinger (2013), Salinger and Prechelt suggest the idea of a base layer in context of studies on pair programming and the grounded theory methodology. The base layer consists of a set of predefined codes (the so-called base concepts), rules for changing these base concepts, including a naming scheme, and a general structure of the concept set. It aims to support a researcher to make faster progress, and to enable studies to be compatible so that results can be related to each other (Salinger & Prechelt 2013). Subsequent studies should be faster because they can use the given set of base concepts to create "higher-level concepts and eventually theory" (Salinger & Prechelt 2013), p. 28.

The idea is to give a kind of head-start, allowing a researcher to begin at a higher conceptual level while still being close to the data. Although this seems to introduce the danger of forcing the researcher to code theory-driven instead of data-driven, the authors claim to have taken some precautions against this danger: the set of base concepts is considerably small; allowing and inviting for additions and changes of the existing base concepts. The base concepts are explicitly not to be misunderstood as a coding scheme, but as a tool "to maximize the reader's capability of thinking flexibly about what it is that appears to be going on in the pair programming session and what might be an appropriate manner of conceptualizing it" (Salinger & Prechelt 2013), p. 35.

In addition, the base concepts aim to be neutral, generic and flexible, and not geared towards a specific research question. However, they are based on some theoretical assumptions, like speech act theory, due to the nature of the data (protocols of pair programmers' utterances during pair programming sessions).

4. Our current Coding Scheme

The previous schemes were mainly created for text data, either of think aloud protocols (von Mayrhauser & Lang and O'Brien, Shaft & Buckley) or of program summaries (Good & Brna). Only Salinger & Prechelt additionally considered audio and video recordings of pair programmers as well as screen castings. We will introduce a coding scheme that operates on eye movements of programmers understanding source code. The first version of the scheme was developed before the Koli Calling workshop. It is based on a short Java program defining rectangles (code 1) and two sets of gaze records by professional programmers. A fundamental decision was to distinguish between observable behavior and its interpretation and to classify codes accordingly.



Code 1 - Source code example used for the workshop (overlaid with eye movements)

Besides primitive categories that denote fixations on a certain point in the program, there are two categories of codes for a series of eye movements called *pattern* and *strategy*. Patterns are observable sequences of fixations, while strategies require the interpretation of a pattern concerning the intention behind this visual behavior. Several researchers involved in computer science education and eye movements defined an initial set of codes, observables as well as potential strategies. Only very few codes, like the scan pattern were adopted from previous research on eye movements in programming. This scheme was given to the workshop participants with the task to code the provided eye movement records using the video annotating software ELAN[4] and to modify the coding scheme as they seem fit.

The workshop participants' suggestions for the scheme were compiled into a revised version which was discussed during the workshop. Further revisions were included accordingly. Finally the observable codes of the scheme which only relate to single fixations were abstracted. Table 1 presents an excerpt from the final workshop coding scheme, table 2 a subblock example.

4 See http://tla.mpi.nl/tools/tla-tools/elan/.

Category	Codes	Description	Classification
(Lexical) Element	Public1 , Methodname1 , }1 . . .	(Lexical) element on which the fixation occurs, e.g. an operator or identifier	Observable
Block	Attributes, Constructor, Main, MethodX . . .	General area in which the fixation occurs, e.g. the main-method	Observable
SubBlock1, SubBlock2 . . .	MethodBody, ReturnLine, Signature, WhileHead, WhileBody . . .	Specific region in which fixation occurs, e.g. a signature or a line containing a return-statement. Can be nested. Granularity depends on structures of interest.	Observable
Pattern	Flicking, JumpControl, LinearHorizontal, LinearVertical, RetraceDeclaration, Scan, Word(Pattern)-Matching	Flicking: The gaze moves back and forth between two related items, such as the formal and actual parameter lists of a method call. JumpControl: Subject jumps to the next line according to execution order. LinearHorizontal: Subject reads a whole line either from from left to right or right to left, all elements in rather equally distributed time. LinearVertical: Subject follows text line by line, for at least three lines, no matter of program flow, no distinction between signature and body. RetraceDeclaration: Often-recurring jumps between places where a variable is used and where it had been declared (Uwano, Nakamura, Monden, & Matsumoto 2006). Form of Flicking. Scan: Subject first reads all lines of the code from top to bottom briefly. A preliminary reading of the whole program, which occurs during the first 30 % of the review time (Uwano, Nakamura, Monden, & Matsumoto 2006). Word(Pattern)Matching: Simple visual pattern matching.	Observable
Strategy	AttentionToDetail, DataFlow, DesignAtOnce, FlowCycle, Interprocedural-ControlFlow, TestHypothesis, Wandering	AttentionToDetail: Readers are trying to comprehend a piece of code that is not believed to contain bugs. In most cases, there is a slowness to AttentionToDetail, but the subject could also be verifying a global property, such as that argument/ parameter types agree or that the semicolons are present in the right places. DataFlow: Following a single object in memory as its value changes through the program. Can also occur backwards through control flow in service of debugging and/or program execution comprehension.	Interpretation

	DesignAtOnce: LinearHorizontal or Scan, hardly any jumps back. The subject's intention is to understand the general or algorithmic idea, without having the need to go into details. Aiming at understanding by linear reading of the complete (needed) code. Can easily be confused with excessive demand/trial and error, might also include TestHypothesis on local levels. Captures high-level algorithmic thinking, thus, features rather large steps as the gaze sweeps over the text typically associated with Linear and Scan patterns. Suggests reading through part or all of the code in a linear manner, intending to acquire an overall understanding of it. FlowCycle: The same program flow sequence is followed several times, the intent might be to gain a first understanding of the flow, strengthening and reinforcing it with repeated examinations of the same code. The Flicking pattern might then suggest the simplest level of the FlowCycle strategy. InterproceduralControlFlow: The subject follows call-chains in real or simulated sequence of control flow. Intention is to understand the execution or to get the outcome of a code section. Focus is on execution between blocks. TestHypothesis: Repetition of a pattern or gaze path. Occurs in connection with DesignAtOnce or ControlFlow. The subject's intention is to check for some details in understanding. Hints at some issue where either the person was distracted, or which is more difficult to comprehend. Involves repetition of a pattern of gaze, and suggests further concentration in order to better understand a particular detail. Wandering: It appears that the subject was backtracking, seemingly searching for a point to resume the reading after a particular path of reasoning had been exhausted, essentially a transition period or a brief rest between bursts of effort.

Table 1 - Workshop coding scheme (excerpt)

Using this scheme on eye movement records provided an excellent basis for the rich discussion during the workshop. The codes were developed partly top-down and partly bottom-up on only two eye movement records on a single program. It is not possible to draw conclusions about the reliability or the completeness of the scheme. Additionally, it is hard to describe the codes unambiguously, some codes are still rather fuzzy. These shortcomings lie somewhat in the nature of the data, the analysis instrument and the kind of research problem. Nevertheless, applying the scheme illustrated the usefulness of this kind of analysis and we can draw from the lessons learned.

Category	Codes	Description	Classification
Signature	FormalParameterList, Name, Type, Visibility	Precise code section on which a fixation occurs: the signature of a Java method	Observable

Table 2 - Example of a subblock

5. A systematic Approach to develop new Schemes

While the current coding scheme shows that using eye movement records have the potential to be used in PC research to study cognitive processes, a more systematic approach for developing such coding schemes seems needed, especially with regard to evaluating the scheme and coding reliability. A purely data-driven approach is not feasible. It takes circa two hours to code a one minute eye movement record and the the coding was experienced as being very tedious by the coders. Moreover, even though circa 10 coders worked on the data sets, it did not seem like some point of saturation of codes was reached.

Therefore we suggest to integrate qualitative and quantitative methods into a combined research design as suggested e.g. by Mayring (2001). Different models for this are possible. We opt for first applying a quantitative approach on the huge amount of eye movement data and use the results to decide, which data is suitable for qualitative analysis. Thereby we first reduce the data to make a qualitative analysis possible. The second analysis step is an interpretation of the eye movements that were identified as relevant, deepening the understanding.

Drawing on the idea of systematically distinguishing between observable visual behavior and inferences of cognitive processes, resulting coding schemes will have again the two different kinds of codes: observables and interpreted cognitive processes. In the following, we will discuss possibilities and alternatives for these two parts of a coding scheme.

5.1. Finding relevant Patterns in Eye Movement Data

Due to the vast amount of data, it is not feasible to analyze complete eye movement records for cognitive processes. Therefore we suggest to apply a quantitative approach to first find those patterns in the eye movements that occur most often to reduce data to be analyzed qualitatively.

A reasonable procedure is to first compute pairs of fixations that appear most often. Subsequently, the same will be done for sequences of three, four and more fixations. This way, jumps from specific individual elements to other elements can be examined. For this, each element in the program needs a unique identifier as indicated in the current scheme in category 'element'. This leads to the most frequent transitions from certain elements to others, which can be qualitatively analyzed to find the reasons for this often occurring behavior or to associate cognitive processes. But in itself, looking at specific elements is not very meaningful, especially when looking at different programs. Hence, more abstract types of elements have to be studied, like jumps from elements of one lexical category to the same category, and to the other categories. Moreover, switches between even broader types of program elements are of interest.

Sharma, Jermann, Nüssli, & Dillenbourg (2012) suggest to organize Java tokens into the three semantic classes identifiers (I), structural elements (S) and expressions (E).

- Identifier: variable declarations
- Structural: control statements
- Expression: main part of the program, like the assignments, equations, etc.

They regard 3-way transitions as one unit of program understanding behavior. It is suggested that a programmer switching between identifiers and expressions tries to understand the data flow and/or the relation among the variables. Transitions among all the semantic classes indicate the intention to understand the data flow according to the conditions in the program. Table 3 shows the categorization of different transitions among the semantic classes.

Type of flow in the program	Types of transitions
Data flow	I → E → I, E → I → E
Control flow	I → S → I, S → I → S
Data flow according to Control flow (Systematic execution of program)	S → E → S, E → S → E, S → I → E, E → I → S, S → E → I, I → S → E, I → E → S, E → S → I

Table 3 - Categorization of different transitions among semantic classes according to (Sharma, Jermann, Nüssli, & Dillenbourg 2012)

On a more general level, it is worth trying to find the most common 'global' patterns, how programmers go about understanding a source code. Finally, we'd like to choose a few control structures that are of special interest, e.g. loops and conditions. For these longer sets of fixations, instruments to compare fixation sequences as proposed e.g. by Cristino, Mathôt, Theeuwes & Gilchrist (2010) and West, Haake, Rozanski & Karn (2006) can be applied in addition to counting frequencies.

Besides computing possible data points to analyze, it is still a good idea to have a human looking for interesting data sequences. There might be patterns which are not frequent but nevertheless yield rich information, like extreme or very unexpected behavior or ideal cases that can be predicted from current theory.

5.2 Eliciting Cognitive Processes

A qualitative approach will be employed to explore cognitive processes. Instead of analyzing the whole gaze record, only those patterns identified in the quantitative step are looked at. Even after this reduction, plenty of data remains. This data is still in form of eye movement records. There are fixations on the program and saccades between them. This can be displayed as an animation, a video or in form of a graph representing the sequence and duration of fixations (see Code 1). As a start, it would be reasonable to find adequate names for the patterns.

As potential methods, we will discuss qualitative content analysis, phenomenography and grounded theory. They share a related initial analytical approach, in which phenomenography and grounded theory go beyond content analysis to develop theory or a distinctive understanding of the experience (Hsieh & Shannon 2005).

Although there are some other schemes we could draw potential codes from (see chapter 3), we will concentrate on a data-driven approach. The advantage is that results are gained directly from the data without imposing categories or theory. While codes in content analysis can be created either data-driven or derived from theory, in phenomenography and grounded theory categories emerge from within the data. In the following, these three approaches are introduced and their potential for the intended procedure is discussed.

5.2.1. Qualitative Content Analysis

Qualitative content analysis is "a research method for the subjective interpretation of the content of text data through the systematic classification process of coding and identifying themes or patterns" (Hsieh & Shannon 2005), p. 1278.

Qualitative content analysis focuses on texts within their context of communication, the data can be all kinds of recorded communication. There are different approaches, of which conventional, directed, and summative are used often.[5] These approaches differ among other things in the source of codes. Directed content analysis uses existing theory or research to derive the initial coding scheme before

5 Mayring (2000) refers to conventional content analysis as inductive and directed content analysis as deductive category development.

analyzing the data. It aims at extending or refining an existing theory. The summative approach counts single words or content and interprets the underlying context. Conventional content analysis is generally used to gain a richer understanding of a phenomenon, when prior theory or research is limited. The coding scheme is derived from data during data analysis. Codes are sorted into categories and relationships among categories are identified (Hsieh & Shannon 2005; Mayring 2000).

The overall intention of qualitative content analysis to interpret meaning from content matches our proposition. Furthermore the level of our intended outcome corresponds to what seems feasible with qualitative content analysis, producing a coding scheme with categories and codes describing PC processes in order to contribute to theory building. Nevertheless, eye movements are not exactly the kind of data, this approach aims to analyze.

5.2.2. Phenomenography

Phenomenography is an empirical, qualitative research approach that describes the variation in the way people understand or experience a certain phenomenon. The analysis consists of an iterative process, in which the researcher goes back to the data again and again. The outcome space of this analysis is a set of categories specifying different levels of understanding. These categories often have a hierarchical structure, going from categories with few features of the phenomenon to depicting richer or deeper understanding. Phenomenography is usually used in educational settings (Eckerdal 2009). The data for phenomenographic research has the form of people's accounts of their own experience and is usually gathered via interviews. Richardson (1999) points out, that other data sources are possible. Though those are in general just other forms of discourse that have the same evidential status as oral accounts.

While the goal to find ways in which programmers understand source code in general agrees with the phenomenographic paradigm, the intended outcome is still different. At the current point, it is of interest to develop a coding scheme to capture different cognitive processes during PC. The outcome space obtained by phenomenography is already a step further than what seems reasonable right now for the coding scheme.

5.2.3. Grounded Theory

While there are different versions of Grounded Theory Methodologies (GTM), they share some common features. Essentially, the idea is to generate theory from data, by repeatedly comparing and analyzing sections of data (open and intermediate coding), adding new data during the research process (theoretical sampling), until the more and more abstract codes and categories can be linked to a theory, explaining or describing the phenomena embedded in the data. Such a theory is conceptualized as emerging from the data.

An important characteristic is the intertwined process of ongoing analysis and generation of new data; connected to the cyclic approach to coding, where the codes and categories are constantly compared to new data and new codes. This nature allows the resulting theory to focus on those important characteristics that are in the data, not in some pre-defined research hypothesis. To allow the research to be open to the data, the role of literature and current state of research is somewhat ambivalent in GTM. When the methodology was first devised by Glaser and Strauss (1967), the "grounding" of theory directly in qualitative data was supposed to replace an uncritical acceptance of existing theory (Richardson 1999). The codes and categories emerging from the data might be very different from what would be expected from previous research. Theory or literature is used prior to research to sensitize the researcher, or during the coding process, when the theory emerges, as another perspective for comparing codes.

So far, GTM seems to be a suitable option for the data-driven analysis of visual behavior while understanding source code. However, it remains to discuss, what representation of the gaze data is needed to use GTM.

6. Conclusion

We presented an approach to use eye movement data for PC studies. For that purpose, we combine quantitative and qualitative methods to develop coding schemes. As an initial example, we worked on a scheme about comprehension strategies by expert programmers. Taking into account previous coding schemes in this context and the procedures of their development, allowed us to reflect potential pitfalls such as the missing comparability of results in advance.

Coding schemes for eye movement data should contain observable behavior as well as interpreted cognitive processes. For the most part, the observable codes can be assigned automatically, which is an advantage over previous coding schemes. Following the proposed procedure facilitates the comparison of data-driven results with other studies, without having to adopt their theoretical premises. Having a consistent, but yet flexible naming scheme as suggested by von Mayrhauser & Lang (1999) and Salinger & Prechelt (2013) will help that.

In order to use the above discussed qualitative methods, the gaze data could be translated into textual form using observable codes as 'label'. The resulting records would have this form: Signature - MethodBody - MethodBody or Scan - JumpControl - LinearHorizontal, enriched with information on the line and an unique name for the element. This might be seen as a representation of the raw data, similar to the transcript of an interview. However, unlike a transcript, any chosen label is already implying a certain interpretation. Hence, this translation process has to be done carefully. It is interesting to now explore the possibility to produce such a representation of the eye movements in a rigorous, and probably automated or semi-automated way.

7. Acknowledgements

We would like to thank Shahram Eivazi, Tersia //Gowases, Andrew Begel and Bonita Sharif as well as the other participants of the Koli workshop for discussing the ideas presented in this paper.

8. References

Bednarik, R., Busjahn, T., & Schulte, C. (2014). Eye Movements in Programming Education: Analyzing the expert's gaze. Joensuu, Finland: University of Eastern Finland.

Cristino, F., Mathôt, S., Theeuwes, J., & Gilchrist, I. D. (2010). ScanMatch: A novel method for comparing fixation sequences. Behavior Research Methods, 42(3), 692–700.

Eckerdal, A. (2009). Novice Programming Students' Learning of Concepts and Practise. Uppsala University, Uppsala.

Good, J., & Brna, P. (2004). Program comprehension and authentic measurement: a scheme for analysing descriptions of programs. Empirical Studies of Software Engineering, 61(2), 169–185.

Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. Qualitative health research, 15(9), 1277–1288.

Mayring, P. (2000). Qualitative Content Analysis. Forum Qualitative Sozialforschung / Forum: Qualitative Social Research, 1(2).

Mayring, P. (2001). Combination and integration of qualitative and quantitative analysis. In Forum Qualitative Sozialforschung/Forum: Qualitative Social Research (2).

O'Brien, M. P., Shaft, T. M., & Buckley, J. (2001). An Open-Source Analysis Schema for Identifying Software Comprehension Processes. In Proceedings of 13th Workshop of the Psychology of Programming Interest Group (p. 129–146). Bournemouth, UK.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology, 19(3), 295–341.

Rayner, K. (1998). Eye Movements in Reading and Information Processing: 20 Years of Research. Psychological Bulletin, 124(3), 372–422.

Richardson, J. T. E. (1999). The Concepts and Methods of Phenomenographic Research. Review of Educational Research, 69(1), 53–82.

Saldaña, J. (2012). The coding manual for qualitative researchers. Sage.

Salinger, S. (2013). Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung. Freie Universität Berlin, Berlin.

Salinger, S., & Prechelt, L. (2013). Understanding Pair Programming: The Base Layer. BoD–Books on Demand.

Sharma, K., Jermann, P., Nüssli, M.-A., & Dillenbourg, P. (2012). Gaze Evidence for Different Activities in Program Understanding. In Proceedings of 24th Workshop of the Psychology of Programming Interest Group (p. 20–31). London, UK.

Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. (2006). Analyzing individual performance of source code review using reviewers' eye movement. In Proceedings of the 2006 symposium on Eye tracking research & applications (p. 133–140). San Diego, California: ACM.

Von Mayrhauser, A., & Lang, S. (1999). A coding scheme to support systematic analysis of software comprehension. Software Engineering, IEEE Transactions on, 25(4), 526–540.

Von Mayrhauser, A., & Vans, A. M. (1994). Program Understanding - A Survey. Colorado State University Computer Science Technical Report CS-94-120.

West, J. M., Haake, A. R., Rozanski, E. P., & Karn, K. S. (2006). eyePatterns: software for identifying patterns and similarities across fixation sequences. In Proceedings of the 2006 symposium on Eye tracking research & applications (p. 149–154). San Diego, California: ACM.

A Case of Computational Thinking: The Subtle Effect of Hidden Dependencies on the User Experience of Version Control

Luke Church[1], Emma Söderberg[2], and Elayabharath Elango[3]

[1] University of Cambridge, Computer Laboratory, luke@church.name
[2] Google Inc., emso@google.com
[3] Autodesk, Elayabharath.Elango@autodesk.com

Abstract. We present some work in progress based on observations of the use of version control systems in two different software development organizations. We consider the emergent user experience, and analyze the structure of the conceptual model and its presentation to see how this experience is formed. We consider its impact on the adoption of such tools outside software engineering and suggest future lines of research.

1 Introduction to Version Control Systems

Version Control Systems (VCS) are an essential part of most development workflows. They are tools that are used for managing code in different states, allowing users to access previous versions and the history of edits. They are also used to co-ordinate the development activity of multiple developers, simultaneously developing on the same code base. Commonly used examples include Subversion [10], Git [5] and Mercurial [**?**]. The use of VCSs is an important, but incidental activity. It does not directly contribute towards the creation of the software at hand. There are a number of core concepts and processes present in these systems that will be important for our discussion:

- **working copy**: the local machine has a working copy, the state of the world, that is currently accessible by software on the machine that is unaware of the version control system.
- **history**: somewhere, locally or on a separate machine, there is a history that represents alternative versions of the world.
- **change lists**: in the general operational flow, users edit their local working copy, bundle a series of edits into a change list, and then integrate it into the history. This latter step may involve resolving of conflicts due to edits by other users on the same content.

The VCS should be an ideal example of where Computer Science can help the world. The tools are undeniably useful and have been extensively tested in the field. The fundamental problem they aim to address is seen in many activities including accountancy, designing buildings and collaborative paper writing. We have observed users inventing their own ad hoc schemes for these tasks and whilst trying to solve a problem that VCSs have already solved, such as allocating unique version numbers, have ended up with data loss. In an ideal computational thinking world, a few core concepts would be widely taught, and tools used by computer scientists would be made broadly available, leveraging the complex solutions encapsulated in VCSs to improve the general population's management of information.

The desirable user experience is probably best articulated as "development without fear" [12]. That is, the ability to fluidly experiment with the confidence that you can always get back to a needed state, and to interact with others without trampling on their work. For some, this is what VCSs achieve, some interviewed users described VCSs as giving them "the freedom to experiment" and "the flexibility to try new ideas". But this is by no means the universal experience; others described the "fear of losing everything", "I dread having to use it", and talked of commands that "you'd be insane to run without copying everything first".

Other work, discussing the usability of VCSs, has concentrated on their complexity and challenges within their conceptual model [11]. We suggest that the difficult experience is not only due to lack of knowledge, but also highlight other contributing design features and consider the implications for design to support broad dissemination. We structure this discussion by first presenting two empirical studies, then consider an analytical evaluation of the user experience characteristics of an VCS, and finally give a brief design proposal and some conclusions.

2 Study Report: Google

The first and second author interviewed 5 engineers at Google, with experience of using the various VCSs at Google for between 4 months and 10 years. The engineers were engaged in various tasks ranging from managing deployment configuration systems to adding features to existing products. All of the engineers had very substantial professional software engineering experience, and used a command line-based VCS on at least a daily basis. In addition, the first author had an informal discussion with a Google engineer functioning as a team lead. During the analysis of the transcripts from the study and the informal discussion, some patterns emerged which we open coded and have separated into trends.

- **Narrow usage, except amongst team leads**: All users in the study, with the exception of the team lead, had established a workflow of a small set of commands. These workflows would typically only be altered when an unusual circumstance occurred. One user had included a command into their workflow after an incident and then stuck with that routine.
- **Risk aversion, little adoption of new features**: Three users provided examples of risk aversion in their workflow. One user had included a command in their workflow to prevent a past problem from reoccurring, with the sole purpose of preventing a problem which may not be current. Two users avoided using alternative commands that would better suit the task they were trying to perform, due to uncertainty of the consequences of running a new command.
- **Social support requirements**: With the exception of the team lead, users heavily depended on social support for learning how to use the VCS. New team members would be guided by the team lead and would learn over time with the help of peers, both locally and via peer-edited support material. Typically, these support mechanisms were used when a new unfamiliar situation occurred, often in preference to the inbuilt help mechanism of the tool. The team lead also viewed part of their responsibility as providing support for their team's use of VCSs.
- **The majority of errors are slips**: While on their commonly used path of command sequences, all users tended to only have slips. Common examples included a malformed argument, or the incorrect use of a flag.
- **Understand the basic conceptual model of used part of the VCS**: Throughout the study, all users were able to explain the basic conceptual model of how the VCS system they were using worked.
- **Startup cost**: New employees receive training on developer workflows, in addition there is self-study training material, and community support pages. These resources appear to be broadly known about and used. In addition, the knowledge transfer from engineers with more experience is important. Both of these elements can be considered startup costs for using VCSs at the company. During informal conversations about this research with another team lead, the recommendation for the team to switch VCS was accompanied with the caveat "but schedule two weeks of downtime to adapt your workflows".

3 Study Report: Autodesk

The first and third author have been engaged with a team with experience developing on Subversion that has moved to using Git. The team included a UX designer, and a number of software

engineers, test engineers, developer-relations engineers and managers. The team ranges from people with several years of experience using Git, to those new to version control. Their professional development experience ranged from 1 to 13 years. Whilst there was not a formal study on Git usage, the authors have observed a number of patterns in the team members' usage during the 9 months.

- **Narrow usage, except amongst team leads**: The usage of most of the team members other than the tech leads was restricted to a specific series of narrow, repeated interactions. When deviations away from this workflow were required to solve a problem, the team members would frequently prefer to manually 'reset' to a known state where their known workflow was functional again.
- **Risk aversion**: Proposed changes to workflows, or the use of techniques to 'repair' local states were not generally adopted. None of the team reported experimenting with features other than their established workflow, whereas during the same period they experimented with a number of different features of the programming language in use.
- **Repair operations are expensive**: Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.
- **Low user confidence**: A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because "it scares the [elided] out of me"
- **Startup cost is perceived as being high**: New members joining the team have reported a considerable cost associated with learning how to use the source code control system. This has frequently been several days of concerted effort. Significant enough that those team members report successful basic interaction as something to mention in the daily scrum stand-up meetings.
- **Social support requirements**: During the period of observation there have been a number of occasions where the team leads have needed to assist other team members with their use of Git. In a number of cases, this occurred when the team member reported being 'blocked' by an issue with their VCS. The same has not occurred over questions about any of the features of the programming language in use.

4 Analysis - Common Themes

There are a number of elements that are common between these studies, despite being from different development organizations with different training and cultures.

Firstly, there is ritualized behaviour amongst the participants. They follow narrow workflows, changes either get integrated into that workflow and repeated each time, or they tend to 'reset' to a known workflow. These users are experienced professional abstraction workers; if anyone is qualified to manipulate the complex chains of abstract entities (multiple versions of a file, alternative worlds etc.), it would be this user population. However, the behaviour we observe is not one typically associated with mastery, it is much closer to the learned ritualistic behaviour that is sometimes found amongst novice computer users.

Furthermore, the participants in both studies *do* understand the VCSs. During the conceptual probing phase, they are able to clearly and precisely articulate the entities and relationships in their version control system. They frequently engage in social discourse around the tooling, discussing the concepts and their manipulation of them with ease. This implies that the oft asserted assumption that people engage in rituals "because they dont know any better", does

not apply here. Their conceptual understanding is strong, they are highly qualified within the domain, but this is not sufficient for them to feel confident in its behaviour, especially when the participants leave their established workflow.

A number of the participants talked about fear. They were "afraid to change their workflow", a task outside their normal workflow "scared them" and they expected very high levels of reward for change (one participant quantified it as needing a 20% productivity improvement within a week to adopt a new tool). This is the response that Attention Investment [2] would predict for a perceived high risk activity.

It is particularly problematic that a tool that would ideally provide 'development without fear' is empirically a tool whose use is perceived as being risky.

5 Analytical Evaluation

We have seen that a number of participants report that VCSs are perceived as being risky systems to use. In order to understand what might give rise to this, let us consider a Cognitive Dimensions [8] analysis of Git. The *activity* being performed is typically Incrementation (e.g. adding one more commit) or Exploratory Design (e.g. restructuring the history of the repository). There are multiple different notational possibilities, the simplest to analyze is probably the command line interface, which follows a typical noun-verb structure. e.g. `git pull` or `git branch --track MyTestBranch`. These are performed from the local working directory. Here, a few dimensions dominate:

- **Hidden Dependencies**$_{CD}$: There are many Hidden Dependencies within Git; there is a dependency that files on the disk are associated with a branch, another dependency between the local branch and the remote repository and a third that remote branched depend on other branches. Files that have been changed on disk have a dependency on their previous state (which can be accessed by performing a 'revert'). New files that have been created, but have not been explicitly '`git add`'ed, will be unknown to the version control system.
  The list of these forms of dependencies is long. They exist with regard to the branches, the commits, the stash etc. There are tools that exist to make some of these dependencies visible, but it is a challenging task to comprehend the graph of dependencies between these hidden entities. Consider the following:
  ```
  $ git status
  On branch master
  Your branch is up-to-date with 'origin/master'.
  nothing to commit, working directory clean
  ```
  This was actually in a working directory that is hundreds of changes (commits) behind the remote repository, but is up-to-date with respect to local version of the remote repository. As a further compounding factor, the Hidden Dependencies are between abstract entities.
- **Abstraction**$_{CD}$: Git has a moderate abstraction barrier. In order to start using it, the user needs to understand a few concepts (at minimum: the file, the commit, the remote repository and usually also the branch). It is not especially abstraction hungry, allowing branches to be added pretty much at the users discretion.
  Whilst most of its entities are abstract (e.g. the branch), this in-itself would be insufficient to explain the emergent properties. As noted earlier, our users were very familiar with manipulating abstract entities.
- **Premature Commitment**$_{CD}$: With a long view, VCSs can be seen as a tool to decrease premature commitment. They allow different possibilities to be explored non-destructively. However, within an episode of interaction, there is extremely high premature commitment. For instance, you must switch branches to manipulate one of the hidden dependencies before performing an operation.
- **Viscosity**$_{CD}$: The viscosity of Git is usually very low. It seldom requires repeated execution of commands to perform even very complex operations.

– **Closeness of Mapping**$_{CD}$: This will always be challenging for a VCS. Whilst the tool maps well to developer workflows, there is no physical analog for what they offer (being able to go back in time, make a different choice and merge it into the present).

None of these dimensions are themselves a direct description of the user experience, they describe the structural usability properties of the system. The experience emerges out of repeated interactions with the tool. This process of experiential emergence is discussed at considerably greater length in [3, 4].

We postulate that it is a combination of the Hidden Dependencies which leaves the user unclear of the state ('where am I? I'll just reset everything'), the within-episode Premature Commitment ('did I switch branches before doing that push? How do I back out of that?') and the issue that the dependencies are between abstract entities ('is it my local or my remote master that my branch is up to date with?') that causes the experience of fear and associated risk aversion. This analytical analysis also explains why the UI tooling for Git (Figure 1), that was used in the second study, did little to mitigate the usability difficulties. Our intent here is not to be over-critical. It is not a bad UI in itself, it provides recognition rather than recall [9], for the typical Git commands, has a clean interface with common commands clearly accessible.

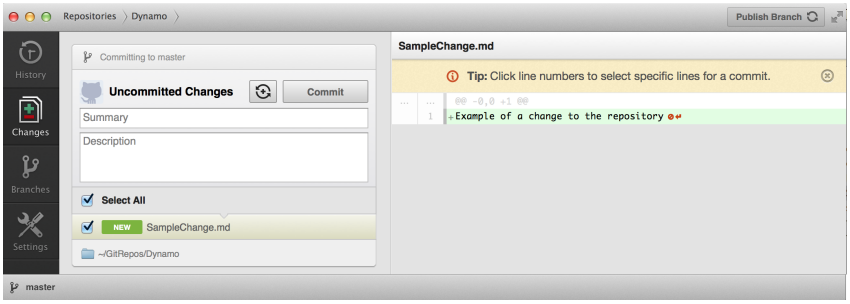


Fig. 1. GitHub's UI tool, Mac OS X, with some whitespace trimmed

However, it does little to help manage the Hidden Dependencies; the user still has to hold the state of the repository in their memory and to mentally simulate the change that is going to occur when they press commit. Or in the absence of doing so, they can just perform a ritual interaction. These dependency issues could be mitigated by making the typical design manoeuvre of making the dependencies explicit, that is, trading Hidden Dependencies for Diffuseness and depicting the states as entities that are available for more direct manipulation. Figure 2 shows examples of the possible outcomes of such a manoeuvre.

Without building it, we can of course only speculate about the usability properties such a system would have. We present the sketches here to encourages others to engage with the experiment of building a version control system that focus on the management of Hidden Dependencies and the emergent user experience of fear.

6 Implications for Design

This work has a number of implications for the design of systems, beyond the immediate example of VCSs.

1. The inclination to assert that conceptual understanding is the only missing piece of programming tooling is not empirically supported. Other aspects are crucial, especially for tools seeking broad adoption
2. Hidden Dependencies may have substantial effects on the experience of using systems, even amongst professional abstraction workers. This is especially relevant to UI libraries that are becoming increasingly dependency dense [1, 7]

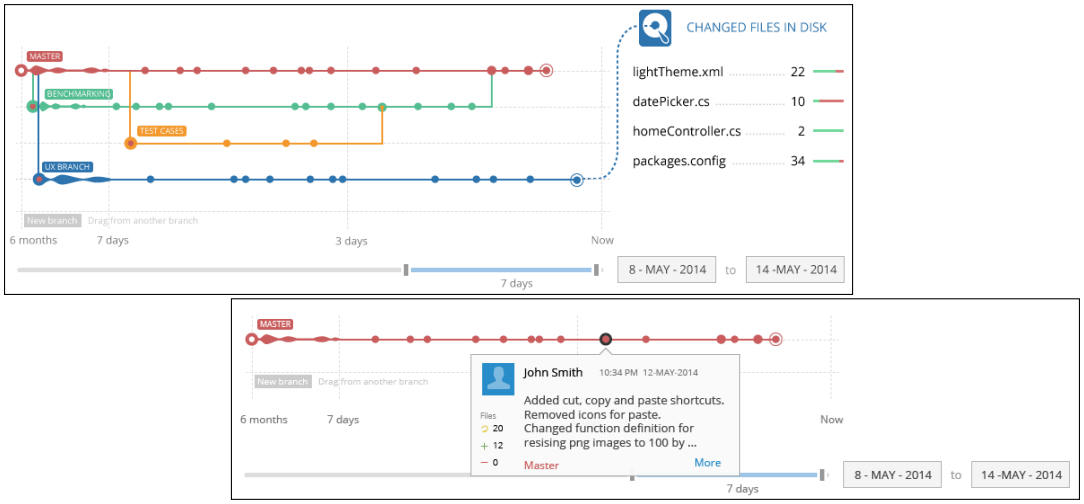


Fig. 2. Upper left: A mock of a tool that makes the changes on the brances, and the relationship between the files on disk and the files in the repository, more directly visible. Lower right: A mock for a tool providing a context plus detail view for understanding the changes that have occured on a specific branch.

3. Risk aversion amongst professionals is a potential "design smell" that indicates that a tool may have serious usability problems when adopted by a wider population

7 Conclusions

In general, VCSs provide an interesting playground for experiments in the psychology of programming. They have many of the same usability properties of programming languages, but without the extremely high implementation costs and challenges to external validity. VCSs play a central role in much software development practice, and they have much to offer to other disciplines. However, we have shown that with their current design, they have very problematic usability, even for professional software engineers. They represent an example where expertise alone is insufficient to give confidence, and have a long way to go before offering 'development without fear'. This is an example of the additional concerns beyond the strictly computational that must be addressed before the transfer of technology from Computer Science to a wider population can be successful.

References

1. Windows presentation foundation. http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx.
2. Alan F. Blackwell. First steps in programming: A rationale for attention investment models. In *HCC*, pages 2–10. IEEE Computer Society, 2002.
3. Alan F. Blackwell and Sally Fincher. Pux: patterns of user experience. *Interactions*, 17(2):27–31, 2010.
4. Luke Church and Thomas R. G. Green. *A Use: Analytical methods for Usability.* 2015. To appear.
5. The Git Community. Git. http://git-scm.com/.
6. The Mercurial Community. Mercurial scm. http://mercurial.selenic.com/.
7. Google. Angularjs, 2014. https://angularjs.org/.
8. Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput.*, 7(2):131–174, 1996.
9. Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In Jane Carrasco Chew and John A. Whiteside, editors, *CHI*, pages 249–256. ACM, 1990.
10. The Apache Organization. Subversion. http://subversion.apache.org/.
11. Santiago Perez De Rosso and Daniel Jackson. What's wrong with git?: A conceptual design analysis. Onward! '13, pages 37–52. ACM, 2013.
12. Dagstuhl Seminar. 13382, 2013. http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=13382.

Work in Progress Report: Nonvisual Visual Programming

Clayton Lewis

Department of Computer Science
University of Colorado, Boulder
Clayton.Lewis@colorado.edu

Keywords: POP-I.A. Learning to program; POP-I.B. Barriers to programming; POP-III.C. Visual languages; POP-IV.B. User interfaces

Abstract

Visual programming systems are widely used to introduce children and other learners to programming, but they cannot be used by blind people. Inspired by the ideas of blind computer scientist T.V.Raman, the Noodle system provides nonvisual access to a dataflow programming system, a popular model for visual programming systems. This paper describes the design and implementation of Noodle and some of its capabilities. The paper suggests that the same approach used to develop Noodle could be applied to increase the accessibility of other visual programming systems for learners with disabilities.

1. Visual programming systems are popular ways of introducing programming to children and other learners.

Despite repeated arguments for their usefulness (e.g. Glinert, 1990; Victor, 2013) visual programming systems have yet to find widespread adoption in a world still dominated by textual languages. But in one important application such systems are now dominant: the introduction of computing to children. Scratch, a visual adaptation of the textual Logo language, has millions of users (Resnick et al., 2009, http://scratch.mit.edu/). Agentsheets, a rule-based language presented in a visual environment rather than textually, is also widely used (Repenning and Ioannidou, 2004, http://www.agentsheets.com/). LabView, a visual programming system originally developed for the control of electronic instruments, is provided as the programming medium for the Lego Mindstorms robotics kits, including WeDO (http://www.ni.com/academic/wedo/), aimed explicitly at younger children (ages 7-11).

These visual programming systems aim to exploit the capabilities of vision to support understanding of program structure and function. Graphical representations and layouts are used to help users recognize the parts of programs, and how they are connected. How well visual systems deliver on these intentions can be questioned; Green, Petre, Blackwell, and others have analysed visual programming systems, with mixed findings (see e.g. Green and Petre, 1996; Blackwell, et al. 2001). Likely the popularity of these languages for children and other learners owes a good deal to the relative simplicity and small scale of the programs typical created by learners, for which the strain on representational systems is reduced.

2. Visual programming systems cannot be used by learners who cannot see.

As noted as early as 1990 (Glinert, 1990, p. 4) visual programming systems present a serious challenge for users who cannot see, for obvious reasons. (In fact, visual programming systems also present barriers to sighted users with motor limitations, because of their typical reliance on mouse interaction. The work to be discussed in this paper addresses these barriers as well, but more modest adaptations of existing systems could likely remove or at least greatly reduce these problems, by permitting navigation of their user interfaces via keyboard commands.)

The limitations of these systems mean that blind children are excluded from learning activities that use them. This is deplorable, since programming is an activity that is in other ways quite accessible to

blind people, because of the flexible access provided by machine-readable representations of textual information, including programs. But, one might feel, if visual programming tools make programming easier to learn for other children, surely it must be right to use them. Equally, one might feel, there just isn't any way to deliver the benefits of "visual" representations to learners who cannot see. But there is a way to attack this painful dilemma.

3. The Raman Principle suggests that no activities are intrinsically visual.

Blind computer scientist T V Raman has suggested (personal communication, 2009) that a way to think about the visual system is as a way to answer queries against a spatial database. If you have an alternate way to ask the queries and get the answers, you don't need the visual system. While Raman has not expressed this principle, in this explicit form, in his writings, related ideas can be found in Raman (1996) and Raman and Gries (1997). See also Lewis (2013).

A clear illustration of the Raman Principle is an adaptation of the Jawbreaker computer game, created by Raman and Chen (n.d.). As discussed in Lewis (2013), playing Jawbreaker involves clicking on coloured balls, making clumps of balls of the same colour go away, and earning points in a way that depends on the size of the clumps. Anyone seeing the game would imagine that playing it is an intrinsically visual activity, and that no one who cannot see the balls and their arrangement could possibly play the game. But Raman, who is totally blind, can play the adapted version of the game.

To create that version, Raman and Chen analysed the questions that sighted players use their eyes to answer, and added keyboard commands, with speech output, that allow players to ask these questions and receive the answers. For example, the scoring system of the game rewards a player who determines which colour of ball is the commonest when the game begins, and avoids eliminating any balls of that colour until the end of the game. That strategy yields the largest possible clump for removal with one click, an effect that dominates the scoring, since slightly larger clumps give much larger scores. A sighted player would judge which colour is commonest by "eyeballing", if playing casually, or by counting, if playing more carefully. A blind player in the adapted game does it by typing "n", and listening to a spoken report, that there are 13 red balls, 15 blue ones, and so on. Other commands and responses allow the blind player to learn what they have to know about the arrangement of the balls in the game.

The Raman Principle does not assert that the mental processes associated with an activity (playing a game, in the example) will be identical for different presentations. Nevertheless it suggests a way to make an activity supported by visual representations possible for non-sighted users.

3.1 The Raman Principle suggests that any visual programming system could have a nonvisual counterpart that is operable without vision.

The logic of the Jawbreaker example can be extended to visual programming systems. For any such system, one can ask, "What information is a sighted user obtaining from the visual display?" One can then provide a means for a non-sighted user to pose a corresponding question, and to receive an answer, using nonvisual media.

4. Noodle is a nonvisual dataflow programming system.

In dataflow systems, one popular form of visual programming system, programs consist of functional units, whose values (outputs) may be connected to the inputs of other functional units, and whose inputs may be connected to the outputs of other functional units. Connections are shown visually as lines or curves, representing paths along which data flow from one functional unit to another. Programs are built by selecting available functional units from a palette, placing them in a construction area, and connecting their inputs and outputs.

Noodle provides a system of keyboard commands with speech output that supports creation, execution, and modification of dataflow programs, with visual presentation being optional. That is, Noodle provides a simple diagrammatic representation of programs, mainly for expository purposes,

but it is intended to be used without any use of these diagrams. Noodle is implemented in JavaScript as a Web application.

Noodle's design is based on a task analysis of dataflow programming, influenced by the Raman and Chen work. Their Jawbreaker user interface supports three kinds of user actions: navigation actions, that do not modify the state of the game, but move a point of focus from one location on the game screen to another; queries, that also do not modify the state of the game, but provided particular information to the user; and operations, actions that do modify the state of the game. Adapting this framework to dataflow programming, the design for Noodle provides navigation actions that move a focus between different parts of the programming environment, for example, between the palette and the construction area (called a *workspace* in Noodle), or between functional units (called *blocks* in Noodle) already placed in the workspace. Noodle also provides operations, including commands that cause blocks to be placed in the workspace, that cause data paths (called *edges* in Noodle) to be placed between blocks, that cause blocks or edges to be deleted, and so forth. The current design includes no queries. Instead, all navigation actions produce a description of the current entity of focus (as discussed later, it may be appropriate to add some queries to the design.) All of these descriptions, and in fact all feedback provided to the user, are presented in audio form, using text-to-speech conversion (using the speak.js package, https://github.com/kripken/speak.js).

Table 1 shows a trace of the creation and execution of a trivial Noodle program.

user types "n" [“new” command]	user types →
system says:type name of new function	system says:palette playSound
user types "foo"	user types →
system says:creating new workspace foo	system says:heading sound operations
user types ←	user types →
system says:palette noteC	system says:workspace foo playSound
user types / [select]	user types →
system says:adding block noteC after externalIn	system says:input connector buffer on block playSound with no edge.
user types →	user types / [select]
system says:palette noteC [see note in caption]	system says:connector selected
user types ←	user types ←
system says:heading sounds	system says:input connector buffer on block playSound with no edge.
user types ↓	user types ←
system says:heading sound operations	system says:workspace foo playSound
user types ←	user types ↑
system says:palette amplify	system says:workspace foo noteC
user types ↓	user types →
system says:palette attenuate	system says:output connector buffer on block noteC with no edge.
user types ↓	user types / [select]
system says:palette mixSounds	system says:adding edge from connector buffer on block noteC to connector buffer on block playSound
user types ↓	user types "r" [“run” command]
system says:palette concatSounds	system plays tone for middle C
user types ↓	
system says:palette playSound	
user types / [select]	
system says:adding block playSound after noteC	

Table 1. Transcript of interaction during creation and execution of a program that plays a tone. Note: When an arrow key is pressed after a command other than a navigation command, the current location is described but not changed, as an aid to orientation.

5. Design and Implementation of Noodle.

5.1 Noodle uses a model-view architecture, with a declarative representation of programs.

The foundation of Noodle is an interpreter for dataflow programs that accepts programs represented as JSON (JavaScript Object Notation) strings. This declarative representation, or model, can be viewed graphically, and in fact could be created and manipulated via mouse interaction, in early versions of Noodle. This same representation can be inspected and manipulated via a completely different user interface, the completely nonvisual interface just described, that uses only keyboard interaction and audio output. Thus a single model can be the subject of multiple views.

This architecture permits multiple user interfaces to be explored easily. Work on Noodle has included four different interfaces: a conventional visual interface, now available for output only, for exposition; the completely nonvisual interface that is the main subject of this paper; an earlier nonvisual interface that used a larger, more complex set of keyboard commands (described in Lewis, 2013); and a button-oriented interface intended for use on phones (currently incomplete.)

5.2 Noodle uses a pseudospatialized navigation scheme to reduce the number of keyboard commands.

The first version of Noodle used a large number of keyboard commands for navigation. For example, separate commands were used to move up and down in the palette, to move up and down in the program construction area, to move among the connectors on blocks, and to follow edges. These operations required a total of eight commands.

To reduce this load (both from learning the commands, and from locating the commands on the keyboard, during use) four generic spatial movement commands (using arrow keys) are defined in the current Noodle user interface. Roughly, the horizontal movement keys move the focus between virtual columns (no spatial layout is actually shown), and the vertical keys move the focus up and down the columns.

The arrangement is "pseudospatial" rather than fully "spatial", not only because no spatial layout is shown, but also because some uses of the keys do not correspond to spatial moves in any simple way. For example, one could think of the palette and workspace as "columns", one to the left of the other, so that one moves “right” or “left” between them, and “up” or “down” within them. But moving to the "right" from a block in the workspace leads to the connectors available on that block for the attachment of edges. Moving further "right" from a connector, if there is a edge attached, leads onto that edge, and moving "right" again leads to the connector on the far end of the edge, which is on some other block in the program construction area. This is plainly not actually "spatial", since moving to the "right" can eventually bring one to a point "above" or "below" the starting point.

5.3 Following an edge shows a contrast between nonvisual and visual dataflow programming.

A key operation in understanding a dataflow program is tracing the data paths that connect the blocks in a program. In visual representations of dataflow programs this is done by tracing lines or curves from one unit to another. While conceptually straightforward, this tracing is often quite difficult to do in practice. Paths are generally not straight, and, worse, frequently cross one another. In complex programs several paths may run in parallel, close together, making it easy to slip over from one path to a different one, while tracing.

Noodle's navigation scheme eliminates this problem. Tracing an edge from one unit to another can be done directly, by moving onto the relevant connector, thence onto the edge, and on to the connector on the other end of the edge. The situation of other edges cannot interfere at all with this process.

6. Capabilities of Noodle

6.1 Noodle supports simple sound processing.

To learn to program one needs to know what programs are doing. Guzdial's (2003) work on media computation suggests that good learning environments allow learners to create media content, such as graphics and sound. For blind learners sound is an attractive choice, and for that reason Noodle includes primitive support for sound processing. Available blocks include ones that produce common musical notes, that mix and concatenate sounds, and that allow the frequency and length of sounds to be under program control.

6.2 Noodle offers the potential of programming on small screen devices.

Current programming systems for phones require large screens. While at least one effort offers application development in Android devices (https://play.google.com/store/apps/details?id=com.aide.ui) the aim seems to be to support tablets with at least modest screens, rather than phones with small screens. The Noodle user interface could be supported with no screen at all, and work has been done on a version intended to be run as a mobile Web app for phones.

7. Nonvisual versions of other visual programming systems should be developed.

Task analysis, as suggested by the Raman Principle, could be applied to the design of alternative user interfaces for visual programming systems that use paradigms other than dataflow. Here are a few illustrative suggestions.

Scratch uses visual cues to distinguish semantic categories and simplify syntactic constraints. These visual cues are the shapes, like the shapes of jigsaw puzzle pieces, that allow a visual judgement to be made that a given piece of program will or will not fit into a program under construction at a given place. If the user tries to place a piece of program in a wrong place, it will not fit. A major reason for the popularity of Scratch, as compared to the older Logo language, is that users do not have to learn complex textual syntax rules, and are not vulnerable to typing errors that produce syntactically invalid code. How could these benefits be secured for users who cannot see?

In a nonvisual version of Scratch, nonvisual navigation commands could traverse gaps in a program, that is, places in a program under construction where new material could be added. From such a gap, other nonvisual navigation commands could directly access palettes of compatible program elements. That is, instead of the user identifying the shape of a gap, and then visually matching the shape of the gap against the shapes of available units in part of the palette, the user could move directly to the relevant section of the palette. Navigation in this nonvisual scheme could be easier than in the current visual scheme, for all users. If this proved to be the case, it could be added to the current Scratch user interface, rather than creating a different, alternative interface.

Another aspect of Scratch that is not workable for blind users is program assembly by visually guided dragging. For example, given a sequence of commands, the user can drag a "repeat" unit up to them, so that it deforms to fit around them, forming a loop with the sequence of commands as the body. Nonvisual placement actions could be provided to support this action. A user would select the beginning and end of the loop body, and then select the repeat unit, and the loop structure could be completed automatically.

Most uses of Scratch today produce graphical, animated output, which would be problematic today for blind learners. In time, support could be contrived for interpreting graphical output nonvisually, as was suggested above for Noodle.

Like Scratch, Agentsheets uses separate palettes for program pieces that play different roles, for example separating conditions and actions for rules (Agentsheets programs are collections of if-then rules.) As for Scratch, nonvisual navigation commands could support this access.

As for Noodle and Scratch, supporting understanding of dynamic graphic behaviour, as well as static arrangements of elements, is an important challenge. Agentsheets has "conversational programming" features that provide (visual) explanations of some aspects of dynamic program behaviour, such as which conditions of rules are currently satisfied. These features might be adapted to provide spoken commentary on program execution, and this enhancement could be of value to sighted users, too. More generally, other design features of nonvisual presentations, such as easier tracing of connections between program elements, or easier access to relevant palette items from a program under construction, could be added to existing visual languages.

8. Conclusion

Nonvisual visual programming is possible, and offers potential benefits to learners who cannot see. Its development may also offer benefit to other users, as is common for work on inclusive design.

9. Acknowledgements

Thanks to Antranig Basman, Colin Clark, Greg Elin, Jamal Mazrui, T V Raman, and Gregg Vanderheiden for useful discussions and encouragement, and to anonymous reviewers for helpful suggestions.

10. References

Blackwell, A.F., Whitley, K.N., Good, J. and Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review* 15(1), 95-113.

Glinert, E. (ed.) (1990) *Visual programming environments: Paradigms and systems.* Los Alamitos, CA: IEEE Computer Society Press.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2), 131-174.

Guzdial, Mark (2003.)A media computation course for non-majors. *SIGCSE Bull.* 35, 3 (June 2003), 104-108.

Lewis, C. (2013) Pushing the Raman principle. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility* (W4A '13). ACM, New York, NY, USA, Article 18, 4 pages.

Raman, T.V. (1996) Emacspeak-- A speech interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '96), Michael J. Tauber (Ed.). ACM, New York, NY, USA, 66-71.

Raman, T.V. and Gries, D. (1997.) Documents mean more than just paper! *Mathematical and Computer Modelling*, Volume 26, Issue 1, July, 45-53.

Raman, T.V. and Chen, C.L. (n.d.) AxsJAX-Enhanced Jawbreaker User Guide. Retrieved from http://google-axsjax.googlecode.com/svn-history/r540/trunk/docs/jawbreaker_userguide.

Repenning, A., & Ioannidou, A. (2004). Agent-based end-user development. *Communications of the ACM*, 47(9), 43-46.

Repenning, A. (2013) Conversational programming: exploring interactive program analysis. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (*Onward!* '13). ACM, New York, NY, USA, 63-74.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009) Scratch: programming for all. *Communications of the ACM* 52, 11 (November 2009), 60-67.

Victor, B. (2013) The future of programming. Retrieved from http://worrydream.com/#!/TheFutureOfProgramming

A Cognitive Dimensions analysis of interaction design for algorithmic composition software

Matt Bellingham
Department of Music and Music Technology, Faculty of Arts, University of Wolverhampton
matt.bellingham@wlv.ac.uk

Simon Holland
Music Computing Lab, Centre for Research In Computing, The Open University
simon.holland@open.ac.uk

Paul Mulholland
Knowledge Media Institute, Centre for Research in Computing, The Open University
p.mulholland@open.ac.uk

Keywords: POP-I.C. end-user applications, POP-II.B. design, POP-III.C. Cognitive Dimensions, POP-IV.B. user interfaces, POP-V.A. theories of design, POP-VI.F. exploratory

Abstract

This paper presents an analysis of the user interfaces of a range of algorithmic music composition software using the Cognitive Dimensions of Notations as the main analysis tool. Findings include the following: much of the reviewed software exhibits a low viscosity and requires significant user knowledge. The use of metaphor (staff notation, music production hardware) introduces multiple levels of abstraction which the user has to understand in order to use effectively: some instances of close mapping reduce abstraction but require the user to do more work. Significant premature commitment is not conducive to music composition, and there are clear opportunities for the greater provisionality that a piece of structurally-aware music software could provide. Visibility and juxtaposability are frequently compromised by complex design. Patching software reduces the hard mental operations required of the user by making the signal flow clear, although graphical complexity can have a negative impact on role-expressiveness. Complexity leads to error-proneness in several instances, although there are some tools (such as error-checking and auto-completion) which seek to ameliorate the main problems.

1. Introduction

This paper presents an analysis, using the Cognitive Dimensions of Notations (Green & Blackwell, 1998), of a representative selection of user interfaces for algorithmic music composition software. Cognitive Dimensions of Notations (CDN) are design principles for notations, user interfaces and programming language design, or from another viewpoint 'discussion tools' for designers (Green & Blackwell, 1998). For the purposes of this report, algorithmic composition software is software which generates music using computer algorithms, where the algorithms may be controlled by end users (who may variously be considered as composers or performers). For example, the algorithms may be created by the end user, or the user may provide data or parameter settings to pre-existing algorithms. Other kinds of end-user manipulation are also possible. The software accepts either precise or imprecise, indicative input from the user which is used to output music via emergent behaviour. A wide variety of algorithmic composition software is considered, including visual programming languages, text-oriented programming languages, and software which requires or allows data entry by the user. The paper considers a representative, rather than comprehensive, selection of software. The analysis also draws, where appropriate, on related discussion tools drawn from Crampton Smith (Moggridge, 2006), Cooper et al. (2007) and Rogers et al. (2011). Finally, the paper reflects on the compositional representation of time as a critical dimension of composition software that is implicit in several of the CDNs.

For detailed images from a wide variety of algorithmic composition software illustrating all issues touched on in this paper, see Bellingham et al. (2014), of which this paper is an abbreviated version.

2. Structure and connections

Sections, and the connections between them, are an important feature of much music. Several genres of music require a sectional structure (ABA, for example). An interface which matches the user's conceptual structure will reduce both repetition viscosity (caused when the software requires several actions to achieve a single goal) and knock-on viscosity (created when a change is made and the software requires further remedial action to restore the desired operation). Cooper et al. (2007) suggest three separate models for the perception of software; the *implementation model* of the software (how it works), the user's *mental model* (how the user imagines the software to work) and the *representational model* of the software (what the software shows the user). Repetition viscosity could be significantly reduced by better matching the mental model to the representational or implementation models. For example, if the implementation model made use of repeating sections the user could apply a change to the section and it would play back correctly both times. If the representational model showed repeating sections (as visual blocks, for example) and then relayed the changed material to all relevant repeats in the implementation layer, the user would input the desired changes once and they would be propagated out to the playback system. This model links to provisionality, which refers to the degree of commitment the user must make to their actions (Green & Blackwell, 1998). It allows users to make imprecise, indicative selections before making definite choices. Provisionality reduces premature commitment as it allows a composer to create sketches before allowing for specific details.

Some of the software under review allows the software to make selections within a given range. *SuperCollider* makes use of `.coin` and `.choose` messages for this reason; `.coin`, for example, represents a virtual toss of a coin. Other software, such as *Max*, can make use of pseudo-random numbers in parameters; this allows the composer to issue a command such as 'use a value between x and y'. Such selections can increase provisionality in the system (the degree of commitment the user must make to their actions), although more complex variations require significant planning which negates the benefits of being able to quickly create a functional piece of code. DAW software such as *Logic Pro* (Apple Inc., 2013) makes use of audio and MIDI loops to facilitate provisionality in composition and arrangement. Users are able to create sketches using loops, replacing them later in the process. Some music composition software allows the user to create music following basic harmonic or rhythmic parameters. *Noatikl* has preset objects which can be used to create sequences, and *Impro-Visor*'s preset algorithms allow for quick musical sketches based on a chord progression (Keller, 2012). One possible design would allow the user to specify the desired structure and then populate the sections. Repeated sections would require two 'pools' of information; those attributes common to both, and those for that specific iteration.

Most software in the domain allows links to be made only between pre-existing entities. In these cases the user is unable to say 'I don't know what is going here', which can be a useful option when composing. One possible solution to this problem would be to decouple the design of the patch/composition from the actualisation. This could take the form of a graphical sketching tool which would allow the user to test the structure and basic design of the patch.

The role-expressiveness of an element relates to how easily the user can infer its purpose (Green & Blackwell, 1998). The use of metaphor (such as mixing desks, synthesisers, piano rolls and staff notation) allows users to quickly understand the potential uses of each editor. Abstractions can be used to make software more effectively match the user's mental model (Cooper et al., 2007). Multiple steps can be combined to make the software conform to the user's expectations. Such abstractions can make use of a metaphor such as the hardware controls of a tape machine. Other hardware metaphors are used in current algorithmic composition software. The *Cylob Music System* by Chris Jeffs (2010) makes use of step-sequencer and drum machine designs, among others.

There are two types of links made in the software under consideration; one-way and symmetric. One-way links send data, whereas symmetric links can both send and receive information. One-way links, such as a send object in *Pure Data* or a variable in *SuperCollider*, do not reflect changes made elsewhere in the system. Visual audio programming systems typically use a patch cable metaphor and, as the majority of physical patching utilises a unidirectional (i.e. audio send or return) rather than

bidirectional (i.e. USB) connection, software such as *Max* and *Pure Data* retains a one-way connectivity metaphor. Visual patching systems allow users to see links at the potential expense of increased premature commitment. The patch-cable metaphor used in visual programming languages makes dependencies explicit and reduces the potential for hidden dependencies. In addition, the use of patch cables is an example of closeness of mapping (Blackwell & Green, 2003).

Both graphical and text-oriented languages can make use of variables and hidden sends and returns. If users are required to check dependencies before they make changes to the software the search cost is increased. This in turn can lead to higher error rates (via knock-on viscosity). Abstractions can impose additional hidden dependencies; users may not be able to see how changes will affect other elements in the patch. Both *Max* and *Pure Data* allow graphical elements (such as colour, fonts and canvas objects) to be added to patches, enhancing the information available via secondary notation (extra information conveyed to the user in means other than the formal syntax). Graphical languages are substantially more diffuse (verbose) than text-oriented languages, and can make hidden dependencies explicit. Text-oriented languages typically have a lower role-expressiveness (how easily the user can infer the software's purpose). *ChucK* (Wang & Cook, 2013) is an interesting hybrid in this respect. Data can be 'chucked' from one object to another using the => symbol, the use of which imitates a patch cable. The other text-oriented languages reviewed do not make direct use of graphical or spatial interconnectivity. In this way *ChucK* makes limited use of Crampton Smith's second dimension of IxD; visual representation (Moggridge, 2006).

3. Time

The passage of time is highly significant when considering the representation of music. Time is not viewed as a separate entity in Cognitive Dimensions, although it is implicit in some dimensions. Payne (1993) reviewed the representations of time in calendars, which primarily focussed on the use of horizontal and vertical spatial information to imply the passage of time: in many cases a similar approach can be taken by music software. Sequencers, such as *Cubase* (Steinberg GmbH, 2013), frequently use horizontal motion from left to right to denote the passage of time. Trackers, such as *Renoise* (Impressum, 2013), frequently show the passage of time as a vertical scroll from top to bottom. Live coding software can present alternative representations of time, as seen in software such as *ixi lang* (Magnusson, 2014), *Overtone* (Aaron et al., 2011) and *Tidal* (McLean, 2014).

Much musical software makes use of cyclic time (loops), as well as linear time. Both of these kinds of time can be sequenced, or mixed, or arranged hierarchically at different scales, or arranged in parallel streams, or all of these at once. *Tidal*, for example, has been developed to allow the representation of linear and cyclic time simultaneously (Blackwell et al., 2014). Software written to perform loop-based music frequently uses a different interface to denote the passage of time. *Live* (Ableton, 2014) makes use of horizontal time in some interface components; other interface elements allow the user to switch between sample and synthetic content in real-time with no time representation. *Mixtikl* (Intermorphic, 2013) is a loop-based system and, in several edit screens, does not show the passage of time at all as the user interacts with the interface.

In a classic paper, Desain and Honing (1993) discuss different implicit time structures in tonal music. They point out that, in order to competently speed up piano performances in certain genres, it is no good simply to increase the tempo. While this may be appropriate for structural notes, decorations such as trills tend to need other manipulations such as truncations without speed-ups or substitutions to work effectively at different tempi. Similarly, elements of rhythm at different levels of periodicity, for example periodicities below 200 ms vs. above 2 seconds, may require very different kinds of compositional manipulation since the human rhythm perception (and composers and performers) deal very differently with periodicities in these different time domains (Angelis et al., 2013; London, 2012). In a related sense, Lerdahl & Jackendoff (1983) uncover four very different sets of time relationships in harmonic structures in tonal music.

Honing (1993) differentiates between tacit (i.e. focussed on 'now'), implicit (a list of notes in order) and explicit time structures. Some of the software under review can be considered in this way; for example, some modes of operation in *Mixtikl* and *Live* utilise tacit time structures, the note lists in

Maestro Genesis and *MusiNum* are implicit time structures, and software such as *Max* or *Csound* can generate material which uses explicit time structures. The flexibility of many of the programming environments under consideration means that the user can determine the timing structures to be used. Honing (1993) also applies the same process to structural relations: he suggests that there are tacit, implicit and explicit structural relations. A system which uses explicit structural relations would allow the musical structure to be both declarative and explicitly represented.

4. Complexity and stability

There are several compositional software interfaces which range from the highly complex and flexible to simple, limited designs. Viscosity - a measurement of the software's resistance to change - is not necessarily a negative attribute. Highly viscous software can present a user with a single, stable, well defined use-case. An example of this is *Wolfram Tones* (Wolfram Research Labs, 2011) which presents the user with a limited control set as a 'black box' (Rosenberg, 1982). Another example is *Improviser for Audiocubes* (Percussa, 2012), in which the complexity of the performance is generated by the physical layout of the Audiocubes (Percussa, 2013). As a result, keeping the sequencing interface simple avoids over-complicating the composition and performance processes. This simplicity, however, increases the viscosity and arguably limits compositional opportunity.

The text-oriented systems under review exhibit poor discriminability due to easily confused syntax, which invites error (Blackwell & Green, 2003). For example, Thomas Schürger's *SoundHelix* (2012) produces code with a large number of XML tags, potentially reducing human readability and increasing the time taken to write the commands. Such a system increases the error-proneness of the system (whether the notation used invites mistakes). Issues of this type can be ameliorated by the syntax checking seen in the Post windows of *SuperCollider* and *Pure Data*, in which errors are outlined in a limited way. A more thorough error-checking system would be a significant improvement in the software's usability. *SuperCollider 3.6* introduced an IDE (Integrated Development Environment) based design, including autocompletion of class and method names. Such a system significantly reduces errors introduced by mistyping.

There are several music metaphors used in the software in this domain which require the user to be conversant in music theory. *Harmony Improvisator* (Synleor, 2013) requires input in the form of scales, chords and inversions. *Noatikl* (Intermorphic, 2012) uses abstractions to create what it refers to as 'Rule Objects' ('Scale Rule', 'Harmony Rule', 'Next Note Rule' and 'Rhythm Rule') to control how the software generates patterns. The *Algorithmic Composition Toolbox* (Berg, 2012) makes reference to note patterns and structures. Roger Dannenberg has explained how staff notation is rich in abstractions (1993); software which uses elements of staff notation is building abstractions on top of abstractions. An example of a consistent design is *Mixikl* (Intermorphic, 2013). The design language refers metaphorically to both hardware synthesisers (the use of photorealistic rotary potentiometers and faders) and patching (patch cables which 'droop' as physical cables do). *Fractal Tune Smithy* (Walker, 2011) makes use of a less consistent design language. The design makes use of notation, piano roll, hardware-style controls, text-based data entry and window and card metaphors. The software is, as a result, highly capable of a wide variety of tasks but potentially at the expense of usability. There can also be consistency issues when software does not use standard operating system dialogue boxes. An example is *SuperCollider*'s save dialogue (McCartney, 2014), in which the 'Save' button is moved from the far right (the OS standard) to the far left. This is a clear example of poor consistency which could lead to unintended user error.

5. Summary

There are opportunities for future work to consider the design of structurally-aware algorithmic composition software. It would be interesting to further employ Cognitive Dimensions in suggesting concrete improvements to the design of the software under review. A full review of time with reference to the CDN using the format suggested by Blackwell et al. (2001) would be a useful process. The CDN is an evolving body of work and there are several new dimensions which could be utilised in future work in the area.

The issue of time raises particular questions. Algorithmic composition tools use varied interaction designs, and may promiscuously mix diverse elements from different musical, algorithmic and interaction approaches. Consequently, such tools can raise challenging design issues in the compositional representations of time. To some degree, these issues parallel similar issues in general programming, for example concerning sequence, looping, hierarchy and parallel streams. However, growing knowledge about how people perceive and process different kinds of musical structure at different time scales suggests that the design of algorithmic composition tools may pose a range of interesting new design issues. We hope that this paper has made a start in identifying opportunities to create or extend design tools to deal better with these challenging issues.

References

Aaron, S., Blackwell, A. F., Hoadley, R. and Regan, T. (2011) 'A principled approach to developing new languages for live coding', *Proceedings of New Interfaces for Musical Expression*, pp. 381–386.

Ableton (2014) 'Ableton live 9', [online] Available from: https://www.ableton.com/en/live/new-in-9/ (Accessed 14 February 2014).

Angelis, Vassilis, Holland, Simon, Upton, Paul J. and Clayton, Martin (2013) 'Testing a computational model of rhythm perception using polyrhythmic stimuli', *Journal of New Music Research*, 42(1), pp. 47–60.

Apple Inc. (2013) 'Logic pro x', [online] Available from: https://www.apple.com/uk/logic-pro/ (Accessed 24 October 2013).

Bellingham, Matt, Holland, Simon and Mulholland, Paul (2014) *An analysis of algorithmic composition interaction design with reference to cognitive dimensions*, Milton Keynes, UK, The Open University, [online] Available from: http://computing-reports.open.ac.uk/2014/TR2014-04.pdf.

Berg, Paul (2012) 'Algorithmic composition toolbox', [online] Available from: http://www.koncon.nl/downloads/ACToolbox/ (Accessed 26 January 2014).

Blackwell, A.F., Britton, C., Cox, A., Green, T.R.G., et al. (2001) 'Cognitive dimensions of notations: cognitive dimensions of notations: design tools for cognitive technology', In Beynon, M., Nehaniv, C., and Dautenhahn, K. (eds.), *Cognitive technology 2001 (lNAI 2117)*, Springer-Verlag, pp. 325–341.

Blackwell, Alan and Green, Thomas (2003) 'HCI models, theories, and frameworks: toward a multidisciplinary science', In Carroll, J. M. (ed.), San Francisco, Morgan Kaufmann, pp. 103–134.

Blackwell, Alan, McLean, Alex, Noble, James and Rohrhuber, Julian (2014) 'Collaboration and learning through live coding (dagstuhl seminar 13382)', *Dagstuhl Reports*, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 3(9), pp. 130–168.

Cooper, Alan, Reimann, Robert and Cronin, David (2007) *About face 3: the essentials of interaction design*, 3rd ed. John Wiley & Sons.

Dannenberg, Roger B. (1993) 'Music representation issues, techniques, and systems', *Computer Music Journal*, 17(3), pp. 20–30.

Desain, Peter and Honing, Henkjan (1993) 'Tempo curves considered harmful', In Kramer, J. D. (ed.), *Time in contemporary musical thought*, Contemporary Music Review, pp. 123–138.

Green, Thomas and Blackwell, Alan (1998) 'Cognitive dimensions of information artefacts: a tutorial', [online] Available from: http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf (Accessed 18 September 2013).

Honing, Henkjan (1993) 'Issues on the representation of time and structure in music', *Contemporary Music Review*, 9(1-2), pp. 221–238.

Impressum (2013) 'Renoise', [online] Available from: http://www.renoise.com (Accessed 24 October 2013).

Intermorphic (2013) 'Mixtikl', [online] Available from: http://www.intermorphic.com/tools/mixtikl/index.html (Accessed 27 March 2013).

Intermorphic (2012) 'Noatikl', [online] Available from: http://www.intermorphic.com/tools/noatikl/index.html (Accessed 27 March 2013).

Jeffs, Chris (2010) 'Cylob music system', [online] Available from: http://durftal.com/cms/cylobmusicsystem.html (Accessed 22 March 2013).

Keller, Robert (2012) 'Impro-visor', Harvey Mudd Computer Science Department, [online] Available from: http://www.cs.hmc.edu/~keller/jazz/improvisor/ (Accessed 27 March 2013).

Lerdahl, Fred and Jackendoff, Ray (1983) *A generative theory of tonal music*, MIT Press.

London, Justin (2012) *Hearing in time*, Oxford University Press.

Magnusson, Thor (2014) 'Herding cats: observing live coding in the wild', *Computer Music Journal*, 38(1), pp. 8–16.

McCartney, James (2014) 'SuperCollider', [online] Available from: http://supercollider.sourceforge.net (Accessed 26 January 2014).

McLean, Alex (2014) 'Tidal - mini language for live coding pattern', [online] Available from: http://toplap.org/tidal/ (Accessed 16 May 2014).

Moggridge, Bill (2006) *Designing interactions*, MIT Press.

Payne, Stephen J. (1993) 'Understanding calendar use', *Human-Computer Interaction*, 8(2), pp. 83–100.

Percussa (2013) 'Audiocubes', http://percussa.us/, [online] Available from: http://percussa.us/ (Accessed 27 March 2013).

Percussa (2012) 'Improvisor', [online] Available from: http://land.percussa.com/audiocubes-improvisor/ (Accessed 22 March 2013).

Rogers, Yvonne, Sharp, Helen and Preece, Jenny (2011) *Interaction design: beyond human-computer interaction*, 3rd ed. John Wiley & Sons.

Rosenberg, Nathan (1982) *Inside the black box: technology and economics*, Cambridge University Press.

Schürger, Thomas (2012) 'SoundHelix', [online] Available from: http://www.soundhelix.com/ (Accessed 26 March 2013).

Steinberg GmbH (2013) 'Cubase', [online] Available from: http://www.steinberg.net/en/products/cubase/start.html (Accessed 24 October 2013).

Synleor (2013) 'Harmony improvisator', [online] Available from: http://www.synleor.com/improvisator.html (Accessed 27 March 2013).

Walker, Robert (2011) 'Tune smithy', [online] Available from: http://www.robertinventor.com/software/tunesmithy/music.htm (Accessed 27 March 2013).

Wang, Ge and Cook, Perry R. (2013) 'ChucK', CCRMA, [online] Available from: http://chuck.cs.princeton.edu/ (Accessed 26 March 2013).

Wolfram Research Labs (2011) 'WolframTones', Wolfram Alpha, [online] Available from: http://tones.wolfram.com/ (Accessed 27 March 2013).

Activation and the Comprehension of Indirect Anaphors in Source Code

Sebastian Lohmeier

M.Sc. Informatik degree course
Technische Universität Berlin
sl@monochromata.de

Keywords: POP-II.B. program comprehension, POP-III.B. new language, POP-V.A. linguistics and mental models, POP-V.B. eye movements and questionnaire

Abstract

A reading experiment is proposed that combines eye tracking and a questionnaire to investigate the effects of the activation of background knowledge on the understanding of indirect anaphors in source code compared to reading Java source code with local variables and qualified expressions.

1. Introduction

Means of referring backwards to what was previously mentioned in a text are called *direct anaphors* in linguistics. Pronouns like *she*, *this*, and definite noun phrases like *the hard disk* can function as direct anaphors. While expressions like `this` exist in programming languages, they do not allow the programmer to choose a previously mentioned entity but work with a fixed entity instead. Lopes, Dourish, Lorenz, and Lieberherr (2003) and Knöll and Mezini (2006) proposed to include anaphors in general-purpose programming languages but did not cover examples like the following, in which *the hard disk* functions as an indirect instead of a direct anaphor:

(1) The computer did not boot. The hard disk had been formatted.

Here, *the hard disk* refers to a part of *the computer*. Indirect anaphors underspecify well-known information (e.g. that computers have hard-disks): the underspecified relation is not encoded in the text, but is taken from world knowledge and added to the reader's mental model of the text's meaning. Lohmeier (2011) describes an implementation of indirect anaphors in an extension of the Java programming language. With the experiment, I seek to identify when a relation learned from source code is well-known to an individual programmer. If that is the case, repeated statements of the relation could be omitted by use of an indirect anaphor when presenting source code to this programmer. Before the experiment is described, relevant prior work from linguistics is summarised, that was used to develop the experiment.

2. Linguistic Background

The proposed experiment is based on experiments on indirect anaphors, activation, and cohesion.

2.1. Indirect Anaphors

There are many ways of characterising indirect anaphors, which are also called bridging anaphors, associative anaphors or accommodations. I use the cognitive model of anaphors from Schwarz-Friesel (2007), which is based on a three-level semantics proposing that a reader constructs a mental representation while reading a text that integrates context-independent conceptual and lexical-semantic information from long-term memory in a text-world model (TWM) that represents context-specific current meanings in nodes and relations between them. In three-level semantics, words refer to nodes in the TWM which are called *referents*. Example (1) from above can be used to exemplify the model of Schwarz-Friesel. The indirect anaphor *the hard disk* refers to a TWM-node that I call HARD-DISK-1. This node is connected to a COMPUTER-1 node that was established when its so-called anchor *the computer* was read.

Both nodes are connected by a PART-OF-1 link from HARD-DISK-1 to COMPUTER-1 that is created through subconscious retrieval from the context-independent concept COMPUTER after reading *the hard disk*. The PART-OF-1 link is also said to function as *anaphora* relation. It increases the *coherence* of the sentence, the mental connection of the referents of a text. Because the *PART-OF-1* link is not expressed in the text (as could be done by using *its hard disk* instead of *the hard disk*), it is said to be *under-specified*. Using *its* instead of *the* would not only yield *coherence* but also *cohesion*: it would be visible in the text that it is coherent. Besides other forms, Schwarz-Friesel distinguishes between indirect anaphors based on part-of (i.e. meronymic) relations like in (1) and indirect anaphors that fill a verb role, as *the chalk* in

(2) The teacher was busy writing an exercise on the blackboard. However, she was disturbed by a loud scream from the back of the class and the chalk dropped on the floor. (Garrod & Terras, 2000, 529)

where the referent of *the chalk* fills the INSTRUMENT role of the referent of its anchor *writing*. Example (2) shows an excerpt of one of the 24 texts that Garrod and Terras (2000) constructed to consolidate earlier research by comparing the understanding of indirect and direct anaphors as well as dominant and non-dominant instruments (besides further conditions). (2) is an example of a non-dominant indirect anaphor. When *with chalk* is appended to the first sentence in (2), the anaphor *the chalk* in the second sentence will pick up that mention of chalk and be a direct anaphor instead of an indirect one. The texts for the conditions with dominant anaphors used a pen instead of chalk because writing is more commonly performed with a pen than with chalk. Besides fixation duration, Garrod and Terras (2000) measured regression-path reading-time for a word, that is the sum of the durations of all fixations from the initial fixation on the word, until the eyes fixate a word to the right of the word. They found that for dominant relations like the one between *writing* and *pen*, subjects' regression-path reading-times separately calculated for the anaphor and for the following verb were not longer for indirect anaphors than for direct anaphors. For non-dominant relations, like the one between *writing* and *chalk*, regression-path reading-times on the verb following the anaphor were 48 ms longer on average for indirect anaphors than for direct anaphors. This means that in terms of regression-path reading-times, the comprehension of indirect anaphors is harder than understanding of direct ones if the relation between the anaphor and anchor is not well-known. This argument requires a task design that avoids problem solving so that delayed effects on eye movements stem from discourse integration only and not from e.g. resolving logical puzzles. It can then also be assumed that the *midas touch* problem does not occur where subjects gaze at a certain location while thinking about a problem instead of perceiving their environment.

2.2. Activation

How well a relation is known may be modelled via its activation in mind, a theoretical construct that describes how probable and fast recall of a referent may be. Rayner, Raney, and Pollatsek (1995) report that words are read faster at later re-occurrences in a text: gaze durations for rare words were about 50 ms shorter from the third encounter on, compared to the initial occurrence; gaze durations for frequent words were about 10 ms shorter from the second encounter on. The dominance of a relation between a verb and an instrument could therefore be manipulated by repeatedly exposing subjects to expressions of the relation in texts.

2.3. Cohesion

The use of indirect anaphors reduces the cohesion of a text, i.e. does not show in the text the coherence relations a reader is to establish during reading. In a study to replicate the so-called *reverse cohesion effect*, O'Reilly and McNamara (2007) increased the cohesion in a biology text on cell mitosis presented to college students by replacing pronouns by noun phrases as well as by adding elaborations and sentence connectives. Subjects answered comprehension questions that asked for information from a single sentence (text-based questions) or required integration of information from two sentences (inference-based questions). O'Reilly and McNamara (2007) found that readers with high prior knowledge answered text-based questions better for low cohesion texts than for high-cohesion texts if their comprehension skill

was low. They explain this *reverse cohesion effect* by hypothesising that low-skill readers skim more and therefore likely miss more detail in high-cohesion texts compared to low-cohesion texts. While the effect was not present for readers with high comprehension skill, the study confirmed the possibility that reduced cohesion can, depending on prior knowledge and comprehension skill, potentially improve comprehension as measured by comprehension questions. The same effect may occur when cohesion is reduced in source code through use of indirect anaphors subject to prior knowledge and (program) comprehension skill.

3. Indirect Anaphors in Object-Oriented Programming

As has been done in Lohmeier (2011), class declarations in object-oriented source code, e.g. in Java, can be interpreted as conceptual structures: the field and accessor method declarations of a class can describe part-of relations. Likewise, the arguments and the return type from a method declaration can be interpreted as verb roles that are used to anchor an indirect anaphor in a method invocation expression which is based on the declaration of the method. An indirect anaphor is anchored in a method invocation expression when the returned value is not assigned to a local variable – otherwise the anaphor would be a direct one and be related to the local variable. To anchor indirect anaphors in arguments of a method invocation expression, it would be necessary to omit method arguments, and use some kind of default value, which is not possible in Java. As in Lohmeier (2011), I will therefore use the return value only to anchor indirect anaphors in method invocations. In an extension of the Java programming language an indirect anaphor has a type and can be anchored in expressions with a type that declares a field or accessor method of the type of the anaphor. An indirect anaphor can also be anchored in a method invocation expression if the return type of the method is compatible to the type of the anaphor. Indirect anaphors are marked with a dot preceding the type name used to find an anchor. E.g. `.ServiceID` can anchor in a method invocation that returns a `.ServiceID` or in an expression of a type that declares a field of type `ServiceID` or defines an accessor that returns a `ServiceID`. The activation of the relation between indirect anaphor and anchor can be manipulated by varying how often or how recently programmers were shown the declaration or use of such fields or (accessor) methods. Direct anaphors can also be used in source code. Their syntax is identical to the syntax of indirect anaphors. The type name that is part of the direct anaphor is used to find an expression or parameter of that type within the method body or constructor body that contains the anaphor. The value of the expression will at runtime be used as the value of the direct anaphor. Direct anaphors in source code enable repeated reference to values made accessible via indirect anaphors. Direct anaphors are introduced together with indirect anaphors in this experiment because omitting them removes a potential source of confusion lying in the fact that direct and indirect anaphors share the same syntax but are understood differently.

4. Experiment

Participants are assigned to one of four groups that balance potential position effects of item sequence (items 1-20 followed by items 21-40 vs. items 21-40 followed by items 1-20) and sequence of exposition to target expression type (using indirect anaphors in the first and not the second block of items vs. not using them in the first and using them in the last block). The experiment comprises the following factors: a between-subject factor of program comprehension skill (high vs. low), as well as three within-subject factors of target expression type (indirect anaphors vs. local variables or qualified access), the activation (high vs. low) of the target relation that is under-specified in indirect anaphors and given in the source code otherwise, and the type of comprehension questions (text-based vs. inference question). I do not treat the different kinds of indirect anaphors (anchored in a field or accessor vs. anchored in the return value of a method) as separate conditions because both cases involve the under-specification of a direct relation between indirect anaphor and antecedent. The following dependent variables will be analysed: regression-path reading-time on the target expression, regression-path reading-time on the word following the target expression, task duration (the time a subject takes to complete a task), and error rate in comprehension questions. Based on the review of prior work in linguistics, I expect the following effects.

A. Regression-path reading-time on an indirect anaphor or the following word will be shorter, the more active – i.e. more recently and frequently presented – the underspecified relation, analogous to the dominance effect found by Garrod and Terras (2000).
B. If a target relation is highly activated, regression-path reading-times will be equivalent for both types of target expressions, like the results of Garrod and Terras (2000) for dominant verb arguments.
C. For programmers with low program comprehension skill and for highly activated relations, the error rate for text-based comprehension questions is expected to be lower for the test tasks with indirect anaphors than for the control tasks with local variables and qualified expressions, in line with the findings of O'Reilly and McNamara (2007) for readers with high background knowledge and low comprehension skill. High background knowledge is here equated with high activation of relations.
D. For highly active relations, duration could be lower for test tasks than for corresponding control tasks because under-specification reduces the amount of text to be read and Garrod and Terras (2000) found that indirect anaphors for dominant relations are gazed at as long as the control targets.
E. Alternatively, task duration could be higher for test tasks with anaphors than for corresponding control tasks, if indirect anaphors are generally harder to understand than normal Java code.

4.1. Participants

I intend to test 30 subjects with practical experience in larger software projects using Java or Scala – both students enrolled in a Master of computer science degree course and professional programmers.

4.2. Apparatus and Procedure

Participants are seated in a dedicated laboratory room in front of an SMI iView X Hi-Speed 1250 tower-mounted eye tracker using a scan rate of 500 Hz. The stimuli are displayed in a customised version of the Eclipse IDE that uses a plug-in (http://monochromata.de/eyeTracking/) to control the eye tracker and correlate eye movement data with words displayed in the IDE's editor with syntax highlighting. The Eclipse IDE is customised to hide the application toolbar and disable markers and annotations in the editors (e.g. for errors shown because the editor is not able to parse indirect anaphors). The editors are put into a disabled state that removes the context menu and prevents text selection.

After being welcomed, subjects complete a program comprehension skill questionnaire. The results of the questionnaire are used to balance the average program comprehension skill questionnaire score between the four groups of the experiment. An introduction to anaphors in source code and a test on anaphors in source code are performed before the eye tracker is calibrated and the participants perform the main test that comprises 40 items. To limit problem solving during the main test in favour of reading comprehension, subjects are instructed to work quickly, but not quicker than is required to work accurately. Subjects are informed that they are going to read source code that does not contain any errors. After the main test, subjects complete comprehension questions and have 5 minutes to write a short summary of important relations in the source code. The procedure takes about 90 minutes per subject.

4.3. Materials

The items were derived from source code of the Apache River project (http://river.apache.org/) that is sufficiently specialised not to be known by subjects and to enable an effective manipulation of activation levels. Subjects of all groups receive 40 items each; 32 of the items contain anaphors. The average activation of the underspecified relations of indirect anaphors is balanced between different kinds of indirect anaphors, as are the number of occurences of the different kinds of indirect anaphors. Four fixed lists of items are used. The items with direct and indirect anaphors were constructed by replacing local variables and qualified expressions by indirect anaphors and by replacing local variable declarations and uses of local variables by direct anaphors. The items with indirect anaphors were arranged such that each low-activation relation was explicated only once, three items before the item with the indirect anaphor under-specifying the relation. Each high-activation relation was explicated more than once in

```
private void addOne(ServiceRegistrar registrar) {
  LookupLocator loc = registrar.getLocator();
  String host = loc.getHost();
  if (loc.getPort() != Constants.discoveryPort)
    host += ":" + loc.getPort();
  JRadioButtonMenuItem reg =
        new RegistrarMenuItem(host, registrar.getServiceID());
  reg.addActionListener(wrap(new Lookup(registrar)));
}
```

```
private void addOne(ServiceRegistrar registrar) {
  LookupLocator loc = registrar.getLocator();
  String host = loc.getHost();
  if (loc.getPort() != Constants.discoveryPort)
    host += ":" + loc.getPort();
  JRadioButtonMenuItem reg =
        new RegistrarMenuItem(host, .ServiceID);
  reg.addActionListener(wrap(new Lookup(registrar)));
}
```

Fig. 1. A method that is part of the source code of an item with a control target expression (a qualified expression, top) and test target expression (bottom). The bottom shows the test target expression `.ServiceID` that functions as an indirect anaphor. The indirect anaphor is anchored in the expression `registrar` that has a type that declares an accessor method `getServiceID()` that returns a `ServiceID` instance.

items preceding the item containing the indirect anaphor. Each item is followed by a yes-no question with feedback on the correctness of the response to keep subjects focussed on the task. The question is displayed when the subject clicks a button to replace the source code with the yes-no question. Subjects are able to freely view the source code of the current item, of all previous items and the yes-no question of the current item repeatedly by clicking buttons to show them. Presentation of an item and its yes-no question is finished when the user clicks either the *yes* or the *no* button to register her answer. Task duration of an item is defined as the time a subject takes to answer the yes-no question since the first time the source code of the item was displayed.

The source code of each item fits into the editor without requiring scrolling. It contains a package declaration and class or interface declaration with a single method or constructor declaration. Import statement are collapsed and cannot be expanded. All comments have been removed. Statements were inserted or modified in order to control confounding variables. See Figure 1 for an example method declaration from an item. Code from a class or interface declaration can occur in more than one item, with or without partial overlap between the items. The distance between anchor and indirect anaphor was balanced so that the anchor could not be identified solely based on its distance from the indirect anaphor. Because the experiment will be conducted in Germany and I do not assume that there are valid sources of word frequencies for German programmers reading English source code, I did instead balance the frequencies of words used as indirect anaphors and anchors within the text comprised of all items. I also balanced the length of words used as indirect anaphors and anchors.

The comprehension questionnaire contains 20 open-ended questions, 10 text-based and 10 inference-based ones. 5 text-based and 5 inference-based questions target or include knowledge of the relations under-specified in indirect anaphors in the test tasks resp. the relations explicated in the control tasks.

4.4. Data Analysis

The planned analysis comprises the effect of target expression type, program comprehension skill, question type and activation ($2 \times 2 \times 2 \times 2$) on error rate, the effect of target expression type and activation (2×2) on regression-path reading-time of the target expression or the subsequent word and the effect of target expression type and activation (2×2) on task duration.

5. Discussion

Because I want to examine basic effects during reading, I do not aim at high external validity. Consequently, I do not claim that the results are directly transferrable to programming tasks that involve both reading and writing; I do not consider how often indirect anaphors identical to the ones I tested would occur in source code written by programmers themselves; I do not speculate about the frequency with which the activation levels I created through my manipulation occur in practice. Such aspects may be studied in the future.

A number of relevant outcomes are expected from the experiment. Data collected for hypothesis A will show whether there is a measurable and a significant difference on-line between indirect anaphors with under-specified relations at different activation levels. Data for hypothesis B will permit an initial statement on whether there are forms of indirect anaphors that are not harder to understand for programmers than are local variables and qualified expression used instead of the indirect anaphors. Data for hypothesis C will supplement this on-line result with off-line data from comprehension questions that will show whether certain forms of indirect anaphors can improve comprehension for certain kinds of programmers. Data for the two alternative hypotheses D and E will show whether these comprehension-related effects are paralleled by an overall decrease or increase of time duration. If data for hypotheses B and C shows that indirect anaphors are beneficial in certain situations, it will be necessary to investigate how the presentation of source code in integrated development environments like Eclipse can be personalised. Display could be personalised by continuous gathering of eye tracking data to determine whether the relation under-specified in a given indirect anaphor can be well-known enough to make a specific programmer benefit from the presentation of the indirect anaphor at that point in time. In such cases, indirect anaphors will be displayed by the IDE instead of local variables or qualified expressions. In other situations, when it is assumed that there will be comprehension difficulties arising from the use of the indirect anaphor that do not support the work of the programmer, a local variable or a qualified expression will be displayed instead of the indirect anaphor.

6. Acknowledgements

I thank Nele Russwinkel for providing the eye tracker used to prepare the experiment and Arthur Jacobs who provides the eye tracker for the experiment. I am grateful to Lutz Prechelt who took a lot of time to discuss the experiment and to the three anonymous reviewers for their helpful comments.

7. References

Garrod, S., & Terras, M. (2000). The contribution of lexical and situational knowledge to resolving discourse roles: Bonding and resolution. *Journal of Memory and Language*, *42*, 526–544.

Knöll, R., & Mezini, M. (2006). Pegasus: first steps toward a naturalistic programming language. In *Companion to the 21st ACM SIGPLAN symposium on object-oriented programming, systems, languages & applications (OOPSLA '06)* (pp. 542–559). New York: ACM.

Lohmeier, S. (2011). *Continuing to shape statically-resolved indirect anaphora for naturalistic programming.* Retrieved 2014/06/01, from http://monochromata.de/shapingIA/

Lopes, C. V., Dourish, P., Lorenz, D. H., & Lieberherr, K. (2003). Beyond AOP: toward naturalistic programming. *SIGPLAN Notices*, *38*(12), 34–43.

O'Reilly, T., & McNamara, D. S. (2007). Reversing the reverse cohesion effect: Good texts can be better for strategic, high-knowledge readers. *Discourse Processes*, *43*(2), 121–152.

Rayner, K., Raney, G. E., & Pollatsek, A. (1995). Eye movements and discourse processing. In R. F. Lorch & E. J. O'Brien (Eds.), *Sources of coherence in reading* (pp. 9–35). Hillsdale: Erlbaum.

Schwarz-Friesel, M. (2007). Indirect anaphora in text: A cognitive account. In M. Schwarz-Friesel, M. Consten, & M. Knees (Eds.), *Anaphors in text : cognitive, formal and applied approaches to anaphoric reference* (pp. 3–20). Amsterdam: Benjamins.

Exploring Creative Learning for the Internet of Things era

Alan F. Blackwell, Samuel Aaron

Computer Laboratory
Cambridge University
[Alan.Blackwell,Sam.Aaron]@cl.cam.ac.uk

Rachel Drury

Cambridge
rachel.drury@gmail.com

Keywords: POP-I.A. arts collaboration, POP-II.A. end-user

Abstract

We describe a study of a group of artists commissioned to create a new artwork involving programming with the Raspberry Pi computer. From an initial sample of 10 professional artists who mainly work with conventional media, 5 were selected for an intensive series of workshops leading to public presentations of the project and their work. The artists learned to program using the Sonic Pi environment that had recently been created for use in schools. During the project, the Sonic Pi language was also enhanced with new features in response to the artists' creative objectives. Throughout the project, data was collected to record the experiences of the artists, including initial self-efficacy questionnaires, reflective diaries, workshop evaluations, and focus group discussion.

1. Introduction

As low-cost computing devices become increasingly ubiquitous, the opportunity to customise, configure and assemble them into systems increasingly resembles a craft pursuit. The growing popularity of the hacker and maker movements as cultural movements means that programming, as a democratically accessible skill, is acquiring the status that might in previous times have been associated with carpentry, gardening, cookery, home decorating or scale model construction. Popular craft movements are pleasurable and creative pursuits, while also being associated with some degree of utility or public display. In most cases, popular crafts stand in relation to a body of professional craft skills - whether small-scale artisanal or large-scale engineering occupations. The social, practical and creative potential of such skills, together with their professional applications, makes them a natural focus for school curricula. Creative and practical craft skills are an important aspect of citizenship and participation in material culture.

2. Learning to Code as Creative Craft

The fact that low-cost computing is becoming more ubiquitous means that programming is becoming a craft skill of this kind - a creative material competence rather than simply an intellectual or scientific pursuit. We believe that the role of computing in the schools curriculum is becoming modified in recognition of this change. Ten years ago, educational usage of computers was almost wholly oriented toward consuming packaged services (word processors, presentation software etc), with the implicit assumption that the creation of new applications would be relevant only to the technocratic elite rather than the general population. Use of educational programming languages such as Logo and Basic had largely disappeared from the schools curriculum. However, recent popular movements such as Code Clubs, together with the Computing At Schools consortium, have reemphasised programming as a universal practical skill. The rising recognition of hack days, make spaces, and other creative technology buzzwords means that amateur and hobbyist engagement with programming is once again a rising trend.

This new popular interest in programming recapitulates to some extent the UK fashions of the early 1980s, when BBC series The Mighty Micro drew attention to newly available low cost technology, and the BBC Microcomputer offered a standard educational platform for home and school use. In the

Internet of Things era, a group of the early users and developers of the BBC Micro have reconvened[1], in a project that led to the launch and popular success of the low-cost Raspberry Pi computer. The Raspberry Pi Foundation, the educational charity responsible for that initiative, has been a partner in the research described here, in association with the development of Sonic Pi – an environment for creating live-coded music at a level of complexity that is suited to first-language teaching (Aaron & Blackwell 2013).

The implications of these trends for psychology of programming are firstly that "craft-like" creative practices already recognised among highly skilled professional artist-engineers (Woolford et al 2010, Blackwell 2013) are becoming more widely distributed among end-user programmers, and secondly that end-user programming is taking place in the context of a broader range of technical and material practices. Both of these tendencies involve technologies that bridge professional artists and hobby/enthusiasts, for example under the banner of the "maker movement". The preferred computing platforms among these communities include the Arduino, Microsoft Gadgeteer, ARM mBed, Beagleboard and of course the Raspberry Pi.

As noted by one reviewer of an earlier version of this paper, these research concerns are closely linked with those of the "Critical Making" community. We are aware of this relationship, and are continuing to explore it, especially in the context of an experimental graduate course being taught in Cambridge this summer, under the title "Critical coding: An introduction to digital design for researchers and graduate students in the humanities and social sciences"[2].

3. Research Context

In this paper we describe an observational study that was designed to explore individual experiences during creative learning of a technological "craft" by a defined end-user population. We had recently been developing school curriculum materials, and carrying out classroom lesson observations, using the Raspberry Pi computer running Sonic Pi (Aaron & Blackwell 2013, Burnard et al in press). In this new study, we were interested in observing adult users, also using the Raspberry Pi, but in a creative professional context rather than an educational setting. This builds on previous research into programming tools for professional artists that has been reported at PPIG (Church et al 2012, Blackwell & Collins 2005), and also on an earlier investigation of the craft practices of professional software developers who work in professional arts contexts (Woolford et al 2010).

The Defining Pi project was hosted by Wysing Arts Centre near Cambridge, and supported by a grant from Arts Council England. Ten established artists were invited by Wysing Arts Centre to participate in an experimental programme exploring ways in which the Raspberry Pi and Sonic Pi software might adapted or extended for use in creative arts contexts. The ten artists attended a half-day workshop in Cambridge, at which they were given a hands-on introduction to the Raspberry Pi and Sonic Pi. The ten were then invited to write brief proposals for a short research and development project, describing an artistic question or artwork that would be developed using these tools. Five of these proposals were selected by a judging panel, and were funded as artistic commissions. The funded artists worked for two months, during which they attended three further workshops to share their work in progress and receive technical assistance. The final outcomes of Defining Pi, including the work generated and the participants' experiences, were presented to the public at a range of talks and workshops representing different perspectives on creative technology: from popular academic audiences (the Cambridge Festival of Ideas) to visual arts (Wysing Arts Centre) technology enthusiasts (Cambridge Makespace) and performing arts (Cambridge Junction). Selected press coverage of these events, and of the project as a whole, is recorded in the URLs of Appendix D.

The four pieces that were presented at the final workshop of the project, and in various combinations to public audiences, were as follows:

[1] http://www.moviesandquotes.com/the-blues-brothers-1980/were-putting-the-band-back-together/

[2] http://www.digitalhumanities.cam.ac.uk/Methods/Criticalcoding

- Slow Scan Raspberry Pi: Rob Smith's project uses the Raspberry Pi, Sonic Pi and Raspberry Pi camera module to capture an image, encode it into audio and transmit it to a second computer that decodes the sound into an image, which builds up line by line as it is received. He wanted to make the transmission of data tangible to the viewer – the conversion of image data to sound that can be heard travelling through space seemed like a good way to do this.
- Kate Owen's starting point was looking at the movement of fingers typing on a computer keyboard and the parallel to a pianist's fingers on a piano. The final outcome is a live performance which is both visual and aural. The performer types the code of a Sonic Pi program, which is also projected for viewing by the audience. The executing code triggers the playing of sound patterns and recorded samples, including recordings of Kate's fingers hitting the keys and 'tap dancing', and of her voice reading out the file and folder names, alongside more staccato tapping sounds. The code can also be read as a visual poem based on the names of the directories and sound files embedded in it.
- Shapes & Things: Richard Healey's code picks an image at random from a collection of cut-out images of tropical plants that are hosted on the Raspberry Pi's SD card. Sonic Pi then places it at random on the screen, repeating the process infinitely, filling the screen with foliage, in which layers of information appear as dense and exotic as the primeval jungles of Henri Rousseau.
- A transcription of worldwide and cosmic events: Chooc Ly Tan saw the potential for data to become a medium, through which to create accidental associations of text, sound and imageries – as part of a score. She used Sonic Pi to collect: data sourced from the Internet (specifically selected forums, etc); information that surrounds the environment and, the physical phenomena were keys to investigate notions of synchrony (of events) and complexity (of systems), present in the universe. This data was presented via a series of real and computer generated images and sounds.

4. Data collection

Research data was collected throughout the project, in the following forms:

Introductory workshop (10 artists + 4 facilitators)

- At the start of the introductory workshop the artists completed a short written questionnaire, describing their expectations of the project at the point they had been recruited. (Appendix A)
- At the end of this first workshop the artists answered the same questions again, so that we could assess the way that their expectations had changed in response to the workshop presentations and activities.
- All artists completed a self-efficacy questionnaire, designed to explore individual confidence and frequency of computer use along with a range of practical technical skills. (Appendix B)

Commission proposals (10 artists)

- Following the introductory workshop, eight out of the ten of the invited artists submitted short proposals, describing artistic objectives and potential outcomes for the commissioned work. Consideration of these proposals was carried out independently of the questionnaire data from the initial workshop - artists had been told that selection for commission would not be based on their responses to the questionnaire, and the research team did not look at the questionnaires until after commissioning was complete.

Reflective diaries (4 artists)

- Each artist was asked to keep a diary, reporting their experiences and thoughts about work in progress on every day that they worked on the project (Appendix C). Although potentially valuable, only two of the five kept detailed daily diaries. Two others completed partial diaries, and one did not return the diary.

Workshop assessment (5 artists)

- At each of the joint workshops held during the commissioning period, artists were asked to complete the same short questionnaire used at the start and end of the introductory workshop (Appendix A).

Focus group discussion (4 artists, 3 commissioning team, 1 facilitator)

- The final workshop closed with an hour-long facilitated discussion of artists' experiences during the project, which was recorded and transcribed for analysis.

5. Analysis

5.1. Self-efficacy questionnaire

All ten artists completed the initial questionnaire (Appendix B). This allows us to make some broad observations about the previous experiences found across our sample. However, we recognise that this is a very small sample, and that the analysis below can only be offered as tentative observations of trends that might be explored in future research, rather than statistically reliable.

Based on our previous experience as administrators and collaborators in the professional arts, we believe that this sample of ten artists is reasonably representative of demographics in the intended group. They are aged between 30-45, 50% male/female, and almost all educated to the level of a Masters in Fine Art. They are established as professional artists, but not primarily digital arts practitioners (this last was a recruitment criterion in the goals of the project). None had studied maths or science beyond secondary school level, none had professional programming experience (questions 33-40) and the majority had never programmed a computer, although two had some experience of Basic, and two had used AppleScript.

Most scored relatively highly in the computing self-efficacy questionnaire (questions 1-10), with an average of 4.06 on the 5-point scale. Two of the sample were widely separated from the main group on this measure, with average scores of 3.1 and 3.3 over the 10 questions. There is some potential for bias in the responses to this self-rating scale, given that all respondents were aware of the forthcoming selection process for award of commissions. Although the preamble to the questionnaire did emphasise that the research data would not be used for selection, it is possible that the artists may have been unsure of this, especially as this was the first time that they had met the research team. As it turned out, one of the two low-scoring individuals was selected for the commissioned group, as was one of the highest-scoring.

The questions exploring tinkering (questions 26-29) showed a clear distinction between individuals who dismantled devices and made small repairs, and others who did not. With regard to our research question exploring “craft” creativity in the software domain, this distinction was predictive of the degree of ambition in the resulting artworks. We therefore intend to use this kind of question again in future, as we continue to investigate this technical culture trend.

The questions regarding home maintenance (questions 19-23) showed quite widely distributed responses, with no clear correlation to measures of computer self-efficacy or programming. This suggests that people carry out home maintenance tasks according to their particular domestic arrangements, rather than because of natural inclination or aptitude. We did see some stereotypical distribution in responses according to conventional male/female gender roles. As observed by Blackwell (2006), this stereotypical pattern occurs more strongly in the tinkering questions 26-29 than in the utilitarian home maintenance questions.

A notable feature of the questionnaire results was that some behaviours that might be unusual, or evidence of enthusiasm for creative hobbies, in the general population are ubiquitous among professional artists. All respondents built layered Photoshop images on a daily basis, wrote HTML for their own websites, and regularly used hand and power tools. These behaviours were consistent across the sample, independent of the other tinkering and craft-related measures.

Finally, it seems that three specific computer-related configuration activities are fairly well correlated with the computing self-efficacy questionnaire. These are: regular adjustment of Facebook privacy settings (Q18), creation of Excel formulae (Q17), and use of a command line (Q13). The last of these may be particularly useful for future research, in that it is consistently associated with the highest total across all our craft and software measures, and was the central feature in an exploratory cluster analysis of the questionnaire scores. However, we note once more that the very small sample size means that we are unable to treat these suggestions as statistically reliable.

5.2. Workshop questionnaires

The responses to the workshop questionnaires were analysed from the perspective of narrative development through the course of the project. Each artist completed the same questionnaire five times. They did not have direct access to their earlier answers, but individual concerns clearly emerged and were developed across the sequence of responses. Our analysis therefore responded to this data by treating it longitudinally, considering the developing experience of each artist in terms of the "stories" formed by the sequences of five answers to each question.

The camera module – this was an initial point of interest for all, but only two of the artists' projects made use of the camera. This is partly because use of the camera was optional – the artists' projects were driven by creative goals rather than the specific equipment available. However, the technical challenges of providing code support for the new module may have been an additional consideration when project ideas were being developed, meaning it was only used those who tended to be more confident.

The Raspberry Pi platform – all the artists were enthusiastic about the idea of coding in such a "simple" context, often with rather poetic ideas of what this might entail. However lack of fluency by comparison to familiar computing tools, through system setup, speed of response and technical obstacles, caused significant reframing of objectives and adaptation to the Raspberry Pi. This was not what they expected, and made it harder to be spontaneous and creative.

Educational resources – the online resources provided for a technically literate hobbyist community seem to be inappropriate to this group, who found there were significant barriers to entry in terms of the level of technical knowledge required. Several tried Google in the hope of finding solutions to problems, rather than exploring specialist forums. Advice from Sam was essential, but at times may have reduced independence and self-efficacy. Initially, the main exchange of information was during the in-person workshops – both interaction with Sam, and exchanging advice and experiences among themselves. However, as the project progressed and hardware problems were overcome, the creation of a shared space on Github provided a wonderful forum for individual problem-specific threads of conversations with artists starting to help each other out during the final phase of the project. The project mailing list was also relatively successful serving to offer a means for more broader questions and discussions.

Development tools – a key element of the project was the intention to adapt the Sonic Pi language with new media capabilities to support the creative projects proposed by the artists. However as the project progressed, the artists only gradually became familiar with our intention that (by working with Sam) they could extend the Sonic Pi language, and that those extensions would become their tools. We discuss this further below.

Creative outcome – all artists struggled to identify and maintain a concept of what they were trying to achieve artistically and technically, having to adapt their working methods while feeling that they were "near the bottom of quite a steep hill". The relatively short period of the project and the steep learning curve meant that all of the artists had to reduce the ambition of their original proposals. In retrospect, it might have been better for us to have offered a more extended period at the start of the project to learn the technical aspects, with greater variety of phased experiments and alternative outcomes for each artist.

Audience – with uncertain outcomes, the artists found it difficult to answer our question about audience. Because the work was commissioned as research and development projects without a

specific audience in mind, there was significant ambiguity with regard to who the audience for the commissioned work might be – researchers, programmers, Raspberry Pi users, schools, arts centre visitors or festival audiences. As a result, the artists felt a tension between the action of programming and the notion of audience. One artist eventually concluded that using the Raspberry Pi "feels like an isolated experience" rather than something for an audience, while others emphasised visceral experiences of otherwise invisible Internet technologies, or even opened up the experience of programming itself, in a way that communicated some degree of the challenge and obscurity that they had experienced themselves.

We had asked artists to comment on ways in which they felt the Raspberry Pi designers might better support work of the kind they do. All of them found that the practical challenges of assembly and configuration were an obstacle that should be addressed – we report on this further below.

5.3. Reflective diaries

Although all of the five artists set out to make daily reflective diary entries as requested, only one of them continued through the whole of the development period. It was unfortunate that more of this data was not available, because the result did provide real insight into the experience of working with the Raspberry Pi to develop an artistic concept. However, the one complete diary does provide a valuable opportunity for triangulation with respect to other sources of data collected during the project. As a result, we are able to be reasonably confident that the analysis of this diary is to some extent representative. Importantly, the partially completed diaries from the other artists do confirm this analysis.

As with the workshop questionnaires, the approach that we take to the diary data is to explore narrative themes as they develop over the period of the project.

Technical theme: The first few weeks of the project were mainly concerned with start-up – getting the right combination of peripherals and connectors assembled in a suitable working location. For most of the project, descriptions of the technology were concerned with obstacles, and reported failures and frustration.

Artistic theme: Technical frustrations generally presented an obstacle to satisfying artistic achievements. They did occasionally result in creative sidesteps – for example, when the bare Raspberry Pi was unable to produce audio or video output of any kind, the heat of the CPU was used to melt a bar of chocolate. In order to explore the apparent obscureness of interaction with code, the keyboard was painted brown (incidentally resembling a large chocolate bar) to see whether code could be entered by touch-typing – an experiment that was soon abandoned! Satisfaction did start to appear after three weeks or so, especially as aspects of the original concept started to emerge (in altered form), and the somewhat encouraging closing entry after nearly two months “feel like I have found a creative way to engage with RP … finally”.

Concept theme: The development of the artistic concept did proceed in direct response to experiences of using the Raspberry Pi and Sonic Pi. Serious attention was given to its strengths and weaknesses, although many of these might have seemed less of a barrier to an artist with more experience of programming, electronic hardware development or Linux tools. The result succeeds in expressing a personal creative vision, although it is accompanied by concerns over whether this personal experience will be appreciated by an audience who are accustomed to more polished outcomes.

5.4. Focus group discussion

We recorded and transcribed a facilitated discussion involving four artists at the final workshop, together with the commissioning and technical team. Thematic analysis of the transcript identified the following issues:

Collective community: The joint workshops were a key element of the project - these were points at which artists experienced each other's projects, were able to meet with Sam in person, and most importantly, developed confidence by seeing what their peers were achieving, and "went away renewed".

Platform development: The intention of the project to evolve the Sonic Pi language with support for a wider range of technical capabilities and genres was extremely ambitious, requiring artists to look ahead to what they might be able to achieve, but before they had been able to experiment with the facilities they would use. By the last workshop, the potential of defining a new language was starting to be understood.

Hardware configuration: The Raspberry Pi appeals to tinkerers in part because of its openness – a bare circuit board that can be packaged or applied by hobbyists in many different ways. However, despite the fact that this aspect initially appealed to the artists in this project, the need to assemble components and connections from materials at hand posed immediate obstacles to their work. Furthermore, although the Raspberry Pi is cheaper and less encapsulated than a laptop or tablet, that lack of encapsulation also makes it less convenient in a studio environment that benefits from portability - being constrained by network sockets or TV screen locations.

Experimental momentum: Both development and hardware issues meant that the artists found it difficult to establish and maintain momentum. When unable to make progress with Raspberry Pi, they would move on to another project. This experience reflects anecdotal reports that Raspberry Pi's bought by non-technical purchasers can often get put away in a drawer unused, if users experience initial difficulty getting to grips with setup and programming. If we had been able to establish a residency model (perhaps like immersion courses in language teaching), with local technical support, this would almost certainly have been less of a problem.

Educational motivation: The experience of typing and seeing something happen was perceived as motivational. Sam's teaching methods are effective, but the artists were impatient to achieve more at an early stage. Their memory was that at the introductory workshop they "just made a beep," although an audio recording of that workshop reveals considerable sonic experimentation, which, as one would expect, demonstrates a difference between the creative ambitions of professional artists and the school students that Sam had previously worked with. The teaching materials for using Sonic Pi in schools include a simple "cheat sheet" of functionality supported by the language, but for use with artists, an ideal way forward would support avenues for exploration, combined with opportunities to be playful or transgressive.

Creative outcome: The artists perceived their final results as scaled down and less ambitious than the original proposals. Nevertheless, they did create distinctive work that spoke directly to their experiences of the device. At the time of this focus group discussion, none of the work had yet been seen by the public. Subsequent public shows and performance, generally combining short talks about the project by Rachel and demonstrations by the artists with Sam's live coding performance, and sometimes hands-on experimentation with Sonic Pi, have been very well received.

6. Discussion

In this final section, we discuss some of the most salient findings that emerged across the analysis of the different data sources collected during this research. These offer reflection on the project as a whole, and also some indication of opportunities that we consider important for further research.

6.1. Engagement and Attrition

A key element of this project was the objective to work with established professional artists, who, in order to maintain a steady flow of paid work, often have a broad portfolio of activities, much of which is driven by schedules determined months or years in advance – including teaching responsibilities, exhibition openings, construction commissions, performances and so on. As a relatively small commission, taking place over a short timescale, their commitments to the Defining Pi project had to be integrated into these timetables.

The Defining Pi workshops, although a central aspect of our research process, also had to be scheduled to account for bookings at Wysing Arts Centre, Sam Aaron's international bookings for live coding performance, and the obligations of several other collaborators. Finding dates when facilities would be available and the five artists would also be able to attend was difficult, with all of

the originally planned dates having to be rescheduled to accommodate the best possible attendance. Allowing for more time between the selecting the artists and the start of the programme would most likely have eased this issue. Subsequent analysis of interviews, workshop feedback and diary reports confirmed that these workshops were indeed an essential part of the development process, in the eyes of the artists themselves. However, some of the five were able to attend only a small proportion of these. Those who did missed workshops were less engaged, and were not able to complete work that they found satisfying, with one artist unable to complete his project at all.

This can be taken as advice for those planning similar research in future – but it also interacted with the elements of frustration with basic hardware and software capabilities discussed earlier.

6.2. Out-of-box experience of empowerment

The Raspberry Pi is intended to be an open platform, that encourages experimentation not only by providing a supportive community and an open-source operating system, but by revealing the circuit board, its components and its interfaces. A low-cost platform is also perceived as low-risk, in the sense that users can experiment with it without being overly concerned that they might damage an expensive piece of equipment. In our informal observation of computing education for users with low self-efficacy, they find it empowering to assemble their own computer – for example, by adding the USB keyboard and mouse, the display, power supply and memory card to make their Raspberry Pi operational.

In promoting the Raspberry Pi concept, we have sometimes drawn comparisons to the relatively closed consumer experience offered to users of devices such as the Apple iPad. The software, hardware and media retail "ecosystem" of that product is strictly controlled by Apple, in contrast to Free/Libre Open Source Software such as GNU/Linux. Customers are not encouraged to program their iPad (for a long time, the AppStore terms specifically prohibited the distribution of Apps that provided programming functionality), and they are certainly not encouraged to open it up and look at its circuit board. The iPad is sealed shut, providing as closely as possible a completely seamless external surface, with no visible screws or fasteners.

The result is of course a trade-off – Apple products provide a well-controlled "out-of-box" experience, in which the user's experience provides a journey of induction into a refined and predictable commercial world.

In contrast to the Apple experience, open-source platforms such as Raspberry Pi suggest a narrative of technical empowerment, in which all the components are visible, anything can be changed, and the device can be configured in limitless ways. In the Defining Pi project, we hoped that professional artists would experience this empowerment as a liberating starting point for creative experience.

Unfortunately, evidence from workshop reports and diaries suggests that this ambition was frustrated by the difficulties experienced when setting up the Raspberry Pi at home. We provided each of the artists with a Raspberry Pi in the most basic state – the stand-alone board – and demonstrated the process of how to connect it up in the first workshop, at which each of the artists successfully set up their own Pi. The unanticipated obstacles related to the range of behaviours that can be found even in the "standard" interfaces of the Raspberry Pi. Raspberry Pi uses HDMI video output for compatibility with domestic televisions. However, several of the artists did not own HD televisions – and even where they did, these were not necessarily located in their studio workspace. Without a television, it was necessary to obtain an HDMI to VGA converter for use with older computer monitors. However, details such as the difference between a "converter" and a "cable" were unnecessarily worrying. Several of the artists had Apple keyboards that had apparently compatible USB connectors, but included internal USB hubs that contended with the Raspberry Pi system. Others had older keyboards and mice with PS/2 rather than USB connectors.

As a result, the early experiences of configuring and starting the machine were rather "frustrating" – a word that was used consistently and repeatedly in the artists' reports. A consequence was that much of the early enthusiasm for the project dissipated, as cables, keyboards and monitors were bought, tested, swapped, reordered and so on. In retrospect, this was a poor decision on our part. In the same way as

some Raspberry Pi users have given up shortly after getting started due to similar issues, busy artists are equally likely to respond to obstacles by shifting their effort to other projects in which they are making rapid progress, especially where they are must wait for external parties before an obstacle will be resolved. If we had provided a complete kit of parts (several are available for Raspberry Pi), and more detailed instructions on what to do, everything would have run more smoothly – though perhaps with different ultimate outcomes.

6.3. Domain specific language as an artistic tool

Sam's live coding research has been heavily influenced by his previous experience as a developer of domain-specific languages for business and commercial applications. Sonic Pi, in particular, has been developed as a set of domain-specific extensions to the Ruby language, supporting basic music synthesis functions (Aaron & Blackwell 2013).

As a result, it was natural that he should approach this project with the objective of supporting the technical needs of the artists by providing new domain-specific language functionality in Sonic Pi. This is consistent with previous recommendations for art-technology collaboration, suggesting that technologists should provide new tools that empower artists to use technology in new ways (Turner et al 2005). However, it is rather different to our previous observations of digital artist-engineers as craftspeople for whom the creation and maintenance of their own tools is a key element of their creative practice (Woolford et al 2010). Indeed, Sam himself (as with many other live coders) developed and continually refines his own tools – both Sonic Pi, and the more powerful Overtone language oriented toward expert users.

In the case of the artists commissioned to work on this project, we deliberately chose to work with professional artist "end-users" (Ko et al 2011) rather than professional software artist-engineers. We wished to work with this group in order to understand their technical ambitions through early experiments, and to extend the Sonic Pi language into a tool that would help realise those ambitions. However, this meta-level of collaboration was not initially understood by the artists, due to their inexperience with programming concepts and their set-up frustrations. Although it was made explicit from the outset, several of the artists commented that they did not really understand the implications until much later in the project.

In the early stages of the work, the delays in hardware configuration of the Raspberry Pi morphed into perceived delays as Sam waited to receive feedback from artists, and artists waited for technical assistance from Sam. However, as these delays were overcome and confidence grew, the relationship between Sam and the artists as a group developed with an increasingly level of shared communication via the Github forum and discussion group. It was one of the more technically experienced artists (who had some previous programming experience, and also rated high on computer self-efficacy and "tinkering" scales) who fully absorbed the meta-level nature of domain-specific language development in the following diary entry: "realised it's not just about Sam helping us - he wants a conversation about how to structure functions and things".

As this was discussed with other artists in Github exchanges, subsequent workshop meetings, and the final focus group, the implications of creating a new language for computing became more apparent to all members of the group. As an objective for our own future work, we look forward to finding more effective ways to communicate this concept of the conceptual "tool" in a digital era, and reconciling it with the creative and motivational experiences that people have through craft.

7. Conclusion

Overall, there is much we can learn from this short experiment, to inform future projects with artists who would like to work with technology but who don't have programming experience. Despite the "self-help" ethos of communities such as Raspberry Pi, we would advise longer project timeframes and increased support with setting up equipment at home, or use of one of the standard Raspberry Pi kits that are now available. Nevertheless, the work produced by these artists demonstrated interesting and experimental uses of embedded technology. The public talks and workshops were highly popular, attracting total bookings of nearly 300 (limited by room capacity, with almost all events sold out).

Collaboration with the artists has provided valuable input to the Sonic Pi project, including a significant number of new features. The artists themselves gained new insight and skills that they intend to apply in future, building on these projects. Finally, we have benefited greatly from the perspective of Raspberry Pi users who share the creative craft ambitions of the Raspberry Pi Foundation itself, but bring very different life experiences and working practices to their work.

8. Acknowledgements

A large number of people and organisations have contributed to and assisted with this project. We are grateful to Arts Council England, who supported the commissioning and management expenses of the Defining Pi project through the Grants for the Arts scheme. We are grateful to Donna Lynas and Gareth Bell-Jones of Wysing Arts Centre for hosting workshops and working with us to commission and facilitate the work by artists. We are grateful to the Raspberry Pi Foundation for donation of equipment used in this project, and for a donation that has funded Sam Aaron's appointment. We are grateful to the venues and individuals who have helped to present this work to the public: Nicola Buckley at the Cambridge Festival of Ideas, Daniel Pitt at Cambridge Junction, Saar Drimer at Cambridge Makespace, Jason Fitzpatrick at Centre for Computing History, and Tahira FitzWilliam-Hall of the Circuit project. Finally, we are very grateful indeed to the artists who have collaborated with us in this experiment, and most especially for their patience and good will during an unusually challenging process.

9. References

Aaron, S. and Blackwell, A.F. (2013). From Sonic Pi to Overtone: Creative musical experiences with domain-specific and functional languages. Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design, pp. 35-46.

Blackwell, A.F. and Collins, N. (2005). The programming language as a musical instrument. In Proceedings of PPIG 2005, pp. 120-130.

Blackwell, A.F. (2006). Gender in domestic programming: From bricolage to séances d'essayage. Presentation at CHI Workshop on End User Software Engineering

Blackwell, A.F. (2013). The craft of design conversation. In A. Van Der Hoek and M. Petre, (Eds), Software Designers in Action: A Human-Centric Look at Design Work. Abingdon: Chapman and Hall/CRC, pp. 313-318.

Burnard, P., Aaron, S. & Blackwell, A.F. (in press). Using coding to connect new digital innovative learning communities: Developing Sonic Pi, a new open source software tool. Presentation at SEMPRE 2014 - Researching Music, Education, Technology: Critical Insights. Society for Education and Music Psychology Research.

Church, L., Rothwell, N., Downie, M., deLahunta, S. and Blackwell, A.F. (2012). Sketching by programming in the Choreographic Language Agent. In Proceedings of the Psychology of Programming Interest Group Annual Conference. (PPIG 2012), pp. 163-174.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M., Erwig, M., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.-B., Rothermel, G., Scaffidi, C., Shaw, M., and Wiedenbeck, S. (2011). The State of the Art in End-User Software Engineering. ACM Computing Surveys 43(3), Article 21.

Thrift, N. (2006). Re-inventing invention: new tendencies in capitalist commodification. Economy and Society 35(2), 279-306.

Turner, G., Weakley, A. Zhang, Y. and Edmonds, E. (2005) Attuning: A Social and Technical Study of Artist-Programmer Collaborations, In Proceedings of the 17th Annual Workshop on the Psychology of Programming Interest Group (PPIG 2005), 106-119.

Woolford, K., Blackwell, A.F., Norman, S.J. & Chevalier, C. (2010). Crafting a critical technical practice. Leonardo 43(2), 202-203.

Appendix A: Workshop Feedback

Technical	In what ways does the camera module seem to meet your needs or not?
Platform	In what ways does the Raspberry Pi seem a good or bad platform for your work?
Resources	In what ways do the educational resources for Raspberry Pi suit your needs or not?
Tools	Which tools for Raspberry Pi seem to meet your needs or otherwise?
Outcome	In what ways has your work with Raspberry Pi satisfied your ambitions or not?
Audience	How do you think work with Raspberry Pi will be received by an audience?
Aspiration	How could your process or outcomes raise aspirations for Raspberry Pi users?
Strategy	Does the work and/or process suggest any modification or critique of policy for the Raspberry Pi foundation?
Other	Anything else we should be aware of or thinking about?

Appendix B: Self-efficacy Questionnaire

Part 1: Computing Confidence

Imagine you were given a new software package for some aspect of your work. The following questions ask you to indicate whether you could use this unfamiliar software package under a variety of conditions. For each condition, please indicate whether you think you would be able to complete the job using the software package.

1. if there was no one around to tell me what to do as I go.
2. if I had never used a similar tool like it before.
3. if I only had the software manuals for reference.
4. If I had seen someone else using it before trying it myself.
5. if I could call someone for help if I got stuck.
6. if someone else had helped me get started.
7. if I had a lot of time to complete the job for which the software was provided.
8. if I had just the built-in facility for assistance.
9. if someone showed me how to do it first.
10. If I had used similar packages before this one to do the same job.

Part 2: Typical computer usage

11. What operating systems are you familiar with (MacOS, Windows, Linux …)?
12. What software packages do you use regularly (Word, Excel …)?
13. How often do you use a command line?
14. If you use Microsoft Word, how often do you use paragraph styles / create new paragraph styles?
15. If you use Photoshop (or equivalent, like Gimp), how often do you organise an image as layers?
16. If you maintain your own web site, how often do you write HTML code?
17. If you use Excel (or other spreadsheet), how often do you create a formula?
18. If you use Facebook, how often do you modify your privacy settings?

Part 3: Tinkering

19. How often do you personally change parts on a bicycle or car?
20. How often do you re-organise your household files?
21. How often do you change time settings on central heating controls?
22. How often do you service plumbing or electrical fittings?
23. How often do you use a sewing machine?
24. How often do you use carpentry or hand tools?
25. How often do you use power tools?
26. How often do you take something apart to see how it works?
27. How often do you keep old parts in case they are useful?
28. How often do you make an electrical circuit?
29. How often do you make mechanical repairs?

Part 4: Technical education and experience

30. What was your highest school/university qualification in mathematics?

31. What was your highest school/university qualification in a science subject?
32. Have you ever written computer program? (if not, skip to the final question in this part)
33. What programming language are you most familiar with?
34. Where did you learn this programming language?
35. How many years have you been using this language?
36. What is the largest program you have written in this language?
37. What other programming languages have you used?
38. Do you write software as part of your work? If so, how many weeks, months or years have you spent writing software for your work?
39. Have you been paid to write software for other people? If so, how many weeks, months or years have you spent writing software professionally?
40. Is there any other experience or training you have had that you think may be relevant to this research?

Part 5: Demographic data

41. Age
42. Gender
43. First language
44. Highest educational qualification

Appendix C: Reflective Diary

Technical	What technical breakthroughs did you make, or major obstacles encounter, today?
Artistic	What aspects of today's work have been satisfying or otherwise?
Concept	How did your thinking about the project develop today?

Appendix D: Selected Press Coverage of Defining Pi

http://www.cam.ac.uk/sites/www.cam.ac.uk/files/uni-newsletter-summer-2013.pdf

http://www.cambridge-news.co.uk/Business/Business-News/When-Raspberry-Pi-met-the-artists-20130813170131.htm

http://www.cambridge-news.co.uk/Business/Business-News/Raspberry-Pi-finds-new-application-in-the-arts-20130820093523.htm

http://www.cambridge-news.co.uk/Whats-on-leisure/Choice/Our-top-10-picks-for-the-Festival-of-Ideas-20131018060030.htm

http://www.wysingartscentre.org/archive/wysing_on_tour/raspberry_pi_cambridge_festival_of_ideas/2013

http://www.cam.ac.uk/festival-of-ideas/events-and-booking/defining-pi-artist-led-experiments-with-the-raspberry-pi

http://www.cam.ac.uk/festival-of-ideas/events-and-booking/junction-university-sonic-pi-with-dr-sam-aaron

Evaluation of a Live Visual Constraint Language with Professional Artists

Meredydd Williams

Computer Laboratory
Cambridge University
Meredydd.Williams@cl.cam.ac.uk

Keywords: POP-I.B. Choice of methodology; POP-II.A. Learning styles; POP-III.C. Visual languages; POP-IV.B. User interfaces; POP-V.B. Observation; POP-VI.F. Exploratory.

Abstract

A qualitative study of the Palimpsest visual language was undertaken to understand how it would be best used by professional visual artists. Previous work found that individuals with high levels of self-efficacy in both the visual arts and computer use are able to use Palimpsest most effectively. An extension to champagne prototyping was used to study whether visual artists find the language more engaging when their own artworks are being used, and when within their own professional environment. It was found that individuals indeed find most utility in the system when manipulating works similar to their artistic style, and approach tasks differently within their own studios. This offers future opportunities for undertaking studies of professional environments.

1. Introduction

Palimpsest is a visual programming language which is "inspired by early data structure-oriented languages including FORTH and LISP" (Blackwell 2013). The technology is intended for use upon Android touchscreen devices, but is currently best accessed through a Java JAR application. The central abstraction that Palimpsest possesses is that everything is a layer, from the introductory textual tutorial, to the menu screens themselves. Artefacts can be rearranged and stacked in a similar manner to a collage, gradually building an artwork from smaller components. Although many of the effects that can be applied would most likely appeal to graphics editors, such as cropping and colour adjustment (Figure 1), Palimpsest does not assume a particular application domain, and possesses functionality for performing calculations, building animations or recording peripheral input. Whilst traditional textual programming languages might present an intimidating barrier to some visual artists, Palimpsest is designed to facilitate exploration and "tinkering" (Beckwith 2006), allowing artefacts to be adjusted and positioned in a similar manner to within the physical world.

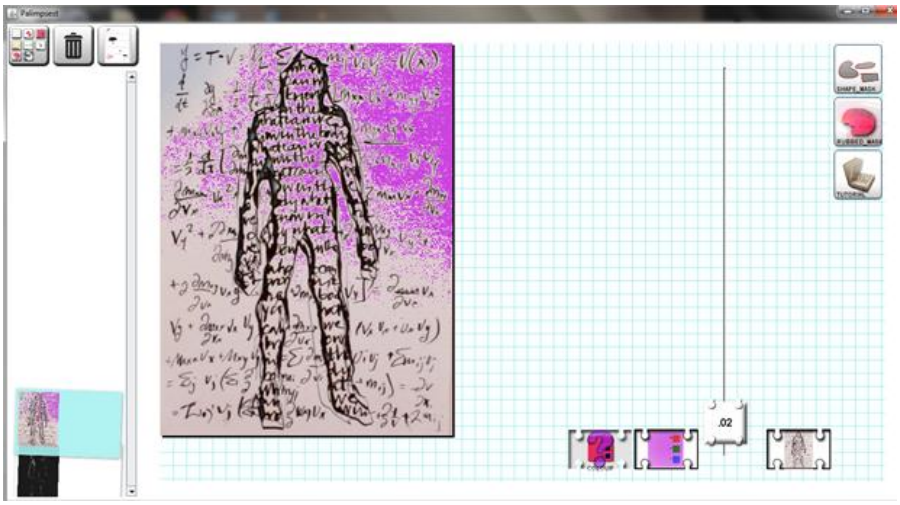


Figure 1 - The Palimpsest system displaying a modified artwork

Previous work by Blackwell and Charalampidis (2013) found that individuals with high levels of self-efficacy in both the visual arts and computer use are able to use Palimpsest most effectively. Within that study, five computer science participants (assumed to possess confidence with computers) and five more artistic participants (assumed to possess confidence in their artistic ability) interacted with Palimpsest as a means of studying which category of user would find the tool most usable. The study was undertaken within the Computer Laboratory in Cambridge, and had participants' first completing self-efficacy questionnaires to assess their confidence in both computer use and art, before proceeding gradually through the 53-stage textual tutorial. It was found that some individuals experienced difficulty with the linear instructions, and so the observer provided a demonstration of the system, which subsequently greatly improved their performance. A free play session followed, where participants could experiment with the tool for up to twenty minutes, before the study was completed with a Cognitive Dimensions of Notations questionnaire (Blackwell 2003).

This study differs in a number of important aspects. Since this previous work presented that those possessing high computer use and visual arts self-efficacy can most effectively use the system, it shall be of interest to observe the utility that professional artists, clearly confident in visual media, can find in Palimpsest. "A central aspect of Champagne Prototyping is that highly credible participants can be used to evaluate the new feature" (Blackwell 2004), which is a key reason for these artistic individuals being selected. Whereas the previous study was conducted within laboratory conditions, this work concerns interactions within a participant's own professional environment, such as an arts studio or workshop. Furthermore, whilst the previous work was rigidly structured and required participants to progress though a linear introductory tutorial, within this study each demonstration is customised to best suit the artistic style of the particular individual. Similarly, during our "free play" stage, participants are encouraged to interact with artworks meaningful to them, whether these embody their typical artistic practice or are artefacts which they have previously created. This experimentation stage makes use of screen capture technology to record the interactions between the participant and the system, which are then played back to the artist during the final interview stage to facilitate contextual inquiry (Beyer 1997). This interview is semi-structured in nature and aims to capture an individual's experience of the system, their professional environment, the meaningful artworks, and their computer use self-efficacy. Whereas the previous study assessed which category of users are most adept with the system, the methodological approach within this paper differs greatly. Interactions with meaningful artefacts within a professional artistic environment are observed, to best assess whether artists can find utility in the Palimpsest system, with the demonstration, location, artworks and questions customised to suit each individual, who is assisted in their interactions by the participant observer (Atkinson 1994). Question customisation results from prior consideration of each individual's artistic field, in addition to the semi-structured nature of the interview, leading to expanded and open-ended responses.

The key research questions to answer comprise of:

- Do those professional artists possessing high levels of self-efficacy find the most utility in the system?
- What features of the system do professional artists find the most engaging, and which require further refinement?
- Did the modification of artworks which represented the participant's artistic style have an effect on their experience with the system?
- Did the location, with the participants using the system within their own professional environments, have an effect on their experience?

2. Related Work

There exists substantial literature relevant to the study. Self-efficacy is an important construct from social psychology and has been shown to influence the decisions which individuals undertake. Bandura (1977) described how levels of confidence affect "whether coping behavior will be initiated", and how long individuals will try to succeed with a task before giving up. Later work concerned the self-efficacy mechanism (SEM), how high levels of confidence might lead to improved performance, and how "collective efficacy" can be developed with certain social conditions (Bandura 1982). Beckwith (2006) also studied the self-efficacy concept, and how "tinkering may yield positive benefits by making the user feel in control of the system". It was shown that individuals possessing low confidence in their own abilities, in this case girls undertaking debugging, can find benefits from playing with their environment. Earlier work, conducted in 1995, also aimed to study the effects of confidence on computer use. Compeau (1995) undertook a seminal study which showed that organisational influences, such as encouragement and support, were important factors in deciding whether users made effective use of computers.

Attention Investment, the balance between the costs of undertaking an action against the potential benefit, has also been studied in great depth. Frequently participants will only find utility within a system if it offers clear benefits over continuing a traditional approach. The artists analysed within this study use physical processes to undertake parts of their professional work; Palimpsest must offer some advantage. Blackwell (2002) considered end-user programmers, those who code for their own use rather than a profession, and the risks they might perceive when deciding whether to manually program a solution. Within the work, participants are presented with the task of correcting spelling mistakes within a document, either through "direct manipulation", likely to be a repetitive process, or the "programming" strategy which is likely to involve a higher attentional cost. It is explained that the length of text and general feel of the document, among other factors, shall influence whether the decision is made to automate or proceed manually. Other work directly focuses upon the Attention Investment model (Blackwell & Burnett 2002) and explores how this might influence the design of functionality within systems such as Forms/3, a novel spreadsheet-based visual programming language by Burnett (2001). The main advantage of this model is described as its flexibility: "it focuses squarely on the user's problem-solving choices wherever they might arise", reflecting that the structure can adapt for the situation the individual finds themselves in. This model shall be useful within my proposed study as a means of understanding the rationale of why participants prefer certain facets of the system. (Eckert 2012) discusses the parallels of design sketching across a wide number of fields, finding that sketches can represent different "degrees of formality". The paper also describes how architects would draw "renditions for different audiences", which can be seen in the case of the Palimpsest visual language. This tool presents information in an informal manner, contrasting with the often viscous text-based formal languages, and tries to allow artistic individuals to design a program on their own terms, rather than forcing them to adapt to methodologies that may be alien or intimidating.

Champagne Prototyping was developed as a means of providing "evaluation techniques suitable for real-world programming environments" (Blackwell 2004), rather than traditional measures that might rely upon laboratory conditions. The paper describes the technique as "an early-evaluation technique that is inexpensive to do, yet features the credibility that comes from being based on the real commercial environment of interest, and from working with real users of the environment", and relies upon the use of credible participants and transcript evaluation. Cheapness and quick feedback are the key advantages of this approach, seen when inexpensive mock-ups are utilised as opposed to full implementations.

Within this paper an extension to this approach is suggested, where both meaningful artefacts and professional locations are used to effectively simulate a working environment, which can then be analysed and studied through participant observation.

3. Participants

The four artists observed within the study were all personally introduced by Alan Blackwell. These individuals were selected due to their proximity, all possessing studios within the immediate Cambridge area (one such example within Figure 2), and for their self-efficacy in visual art, all being professional artists. Although none of the artists were raised in Cambridge, there is a possibility that their views might not be wholly indicative of individuals within other locations within the UK. Their computer use self-efficacy was to be judged from an initial questionnaire, and the levels of confidence would be expected to vary from participant to participant. However, since these artists were known to Blackwell, a lecturer at the University of Cambridge Computer Laboratory, there is a chance that they would be more likely to exhibit some confidence with computing devices. Of the four participants, only Diana had briefly previously seen Palimpsest, whilst the tool was in an intermediate development stage. Although this might suggest that she possessed different initial expectations of the system, this past experience was several years in the past and hence unlikely to have a large effect on her interaction. The introductory process was brief, with myself solely interacting with the artist following their confirmation to view the Palimpsest tool. For this reason it would be expected that the artists' responses would not be significantly affected by the introductory stage.

Figure 2 - An artist's professional environment

The four artists that participated with the study were as follows:

Issam is a Syrian-born artist in residence at Christ's College, Cambridge and works predominantly in fine art, collage, physical installations and photography. The latter passion has led to the existence of a dark room within his studio, which also houses architectural drafting boards, computing devices and extensive room for physical experimentation.

Diana is a professional artist and teacher at Anglia Ruskin University, who also provides bespoke classes across a wide range of artistic fields. Her work is largely multimedia, making use of animation, visual editing and photography, drawing, and live interaction. A frequent theme is the juxtaposition of obsolete computing machinery against alternative art forms, such as interpretive dance. In this way, Diana aims to modernise discontinued devices and equipment most frequently encountered within museums of technology.

Melissa is a professional artist practising in Cambridge, who works predominantly in sculpture, whether stone, metal or wire in nature. Rather than focusing upon structures in a representative manner, Melissa aims to present her material artefacts through suggestive and emotional approaches. She possesses a degree-level qualification in Physics, and has been seen to juxtapose mathematical calligraphy with the human form within some of her works.

Bettina practices at the Wysing Arts Centre, Bourn and has produced numerous site-specific pieces for local government, architects, and the UK Arts Council. Her works concern experimentation with language, whether rendered through textual representations in metal and stone, or purely the spoken word. Although she classes her pieces as contemporary, her training derives from a traditional background.

4. Experimental Procedure

The study comprised of several tasks undertaken within the participants' own context, in order to facilitate the most natural and accurate artistic environment. The four locations used differed greatly, two artists possessed studios within their homes, whilst the other two made use of external facilities. A brief, informal script was prepared in advance in order to most effectively control the system information that was presented to the participants. Approximately an hour and a half was taken to record all the data required for each study. The tasks followed the structure of Champagne Prototyping and included:

A. A brief ten-minute initial stage to better understand a participant's artistic style and what it is they wish to receive from using the Palimpsest system. This included the completion of a short questionnaire to judge the level of self-efficacy each participant showed towards computer use. This artefact, found as Appendix A, was the 2006 adaption (Beckwith 2006) of the Compeau and Higgins questionnaire (Compeau 1995), with only the ethical preamble modified for this study, in order to maintain the validity of the document.

B. A demonstration was presented to the participant, being chosen specifically to suit the individual interests of the artist. This ten-minute stage acted as a brief introduction and tutorial of the language for features of the system that might be of most interest to this participant. The artwork modified within this stage was submitted by each artist several days prior to the study, in order to ensure that the materials were personal to each participant. The artefacts submitted were works previously created by the individual artist, which were sent via email so that the images could be imported into the system when necessary.

C. The participant was asked several questions about the transformation and how they understand it to behave (included as Appendix B), as a means of gauging their level of comprehension. This stage only took approximately ten minutes but helped in both understanding the self-efficacy levels of this participant and recording their first impressions of the system. Audio recording was undertaken, with the permission of the participant, to assist eventual data analysis and to allow the observer to concentrate wholly on the responses received.

D. The participant was allowed up to twenty minutes to interact freely with the system, using artwork individually meaningful to them, whilst asking questions and receiving encouragement from the participant observer. Features were encountered predominantly by discovery rather than encouragement, though on occasions the participant would ask for advice in how to begin a task. Although it is arguable that twenty minutes is a short amount of time for an individual to fully explore the system, in all cases the entire period was not used and the participant felt comfortable to proceed to the final semi-structured interview. This task intended to allow the artist to interact with features of the language they found the most interesting, whilst giving the observer the opportunity to make recommendations. Visual screen recording and audio capture was used to record the actions undertaken during this stage, both for the purpose of analysis and for replaying the footage within the final interview to facilitate contextual inquiry. Once again, the artists were able to modify their own artefacts which they previously submitted. The selection was constrained to the images that they previously submitted, though further files could be downloaded from the Internet and imported to the system if they presented their artworks online. The key task of the process was this exploration period, rather than the final semi-structured interview, giving the artist’s an opportunity to directly interact with a novel system.

E. A fifteen-minute final interview was conducted (included as Appendix C) to assess the participant's understanding of the system, the components they found the most useful, and any alterations they might recommend. Upon all occasions this stage exceeded the fifteen-minute allotted time, though this was consistently due to the participant offering detailed, extended responses which were of use within data analysis. This process was semi-structured, customised to suit the individual artist, and aimed to capture qualitative data. Sections of the visual records were replayed to the participant, similar in style to contextual inquiry, so they could explain their motivation and whether modifying their own artworks led to any changes in their emotions. Once again, audio recording was undertaken to assist data analysis.

Although this task did not directly study interaction with the Palimpsest visual language, it was also required:

F. A contextual inquiry of the participant's professional environment was undertaken, with photographic evidence. Whilst this stage took only approximately five minutes, it was valuable for later analysis of the qualitative data and connecting facets of an individual's environment to their experience with the system. Generally it was found to be of more interest when the participant described and explained their environment themselves; anecdotes were provided for the positioning of certain artistic tools and rationale was given for the placement of technological equipment.

As previously mentioned, the introductory demonstration presented to each participant varied based upon their artistic style and the artwork which they personally submitted. The structure of these demonstrations is described below.

When presenting the system to Issam, several graphics were modified which he had provided. Based upon his interests in collage and layering, a masking effect was applied upon a silhouette image of several household keys, one of his previous artworks. The cropped pieces were then resized with their transparency adjusted so that underlying artefacts could be viewed. A large image of Christ's College was then imported, the location the artist's studio is situated, with strong colour adjustment and quality reduction effects applied which appeared to intrigue the participant. The original key artefact was then placed atop this background as a means of placing the product of the environment within the environment.

Since Diana frequently used graphics editing software, it was important both to display functionality that was familiar, and actions that Palimpsest could perform which existing tools might not. For the demonstration an image of an old communications receiver was used, in an attempt to modernise archaic technology in a similar manner to her work. Firstly, simple visual editing effects were applied, such as changing the hue of the image and applying a shadow outline. More advanced functionality was then displayed as an attempt to differentiate Palimpsest from proprietary editing packages. Examples of this included spin effects, where an image could be set to rotate automatically, or recording mouse movements across the screen.

Melissa frequently experiments in wire sculpture, and for this reason attempts were made to produce similar effects. An image of a humanoid figure was selected, where its size was increased so it filled the entire screen. As a next stage, a strong effect was applied which removed several pieces of the artefact, leaving a thin frame remaining. The colour of this construct was then adjusted to a golden hue, producing a metallic appearance. Within further interaction with the system, Melissa experimented with other visual editing effects and basic animation.

Bettina commented within the study that she was interested in exploring animation effects, and that these could be of use in her professional work. For this reason, a supplied image of the moon was imported and cropped the artefact so that only the central shape remained. As the next step, a large circle of similar size was created, then coloured entirely black. This was placed a layer above the moon image and experimentally increased and decreased in size. A recording was then taken of this shape slowly moving across the moon graphic, as an attempt to mimic the waxing and waning of the celestial body. Bettina particularly enjoyed experimenting with this effect and saw utility in using the tool as a means of creating quick, simple animation effects.

Ethical considerations were important when conducting a study which both is performed within another's environment, and modifies their artworks. The participants were all informed that they were free to withdraw from the study at any point, and that they did not have to provide demographic data if they did not wish. Permission to edit the artworks was acquired from the copyright-holders, as was permission to capture ethnographic photographs within their professional environment. Similarly, permission was granted so that audio recordings could be taken of the interviews, to assist data analysis, and so that screen recording could be used to replay the actions undertaken within the free play session.

5. Data Analysis

Data gathered from the four individual studies comprised of:

A. Four initial questionnaires, including demographic data, which aimed to assist analysis of the computer self-efficacy level each artist possesses.

B. Four audio files containing:

   a. The participants' responses to the post-demonstration interview. These were valuable in assessing initial interpretations of Palimpsest and how challenging the tool is to understand.

   b. The participants' descriptions of their professional environment and why it is structured in a particular manner. These assisted understanding of whether their location affected their experience interacting with the Palimpsest system.

   c. The participants' responses to the final semi-structured interview. These were valuable in gaining a comprehensive understanding of each participant's opinion of the system, its utility for their professional work, and whether they believed the artworks chosen for the study were appropriate. The general interview template was prepared in advance of the studies and modified dynamically depending upon the responses given by each participant.

C. Four screen-capture video files illustrating their interactions with the system during the free experimentation stage. These assisted understanding in what facets of the tool certain artists found the most and least interesting, and were presented to the participants during the final interview stage.

D. Four sets of photographs taken within the respective participant's professional environment. These also aided understanding of whether the studio might affect a participant's interaction.

Analysis of this data was undertaken through the process of Grounded Theory, where concepts are identified within qualitative data, then grouped into categories and collected to form a theory (Glaser 1978). Firstly it was required for the audio transcripts to written in a textual form so that the opinions of the participants are easier to study. In a similar manner, screen capture recordings and ethnographic descriptions were also converted into text. Connected concepts were then highlighted and collected so that correlations could be studied between computer use self-efficacy, the artworks modified, the environment the software was used within, and a participant's experience with Palimpsest. Transcripts were made whilst the data is being analysed to locate at which points ideas are developed, with “memoing” a key concept (Glaser 1998). Once initial findings were drawn from that, the generated hypotheses were refined, with effort taken to critically analyse the theories which are being developed. The study followed Glaser’s approach of induction and that “all is data”, rather than Strauss and Corbin’s systematic approach (Strauss 1994).

Qualitative data was extracted from many sources, including audio transcripts of the post-demonstration interview and final stage, footage captured by the screen recording tool, and the participants' ethnographical descriptions of their professional environment. In performing a Grounded Theory analysis of this data, the audio transcripts were transcribed and key concepts for were collected for each participant, as presented in Figure 3. In the case of Artist A, it repeatedly seemed apparent that they considered Palimpsest to have a “clean interface” and so this was noted. As a

following stage, instances where the clarity of Palimpsest was mentioned within the textual transcripts were highlighted, as a means of understanding whether this view was consistent. Once a fully formed opinion had been understood, it could then be connected to their self-efficacy levels and experience with the tool.

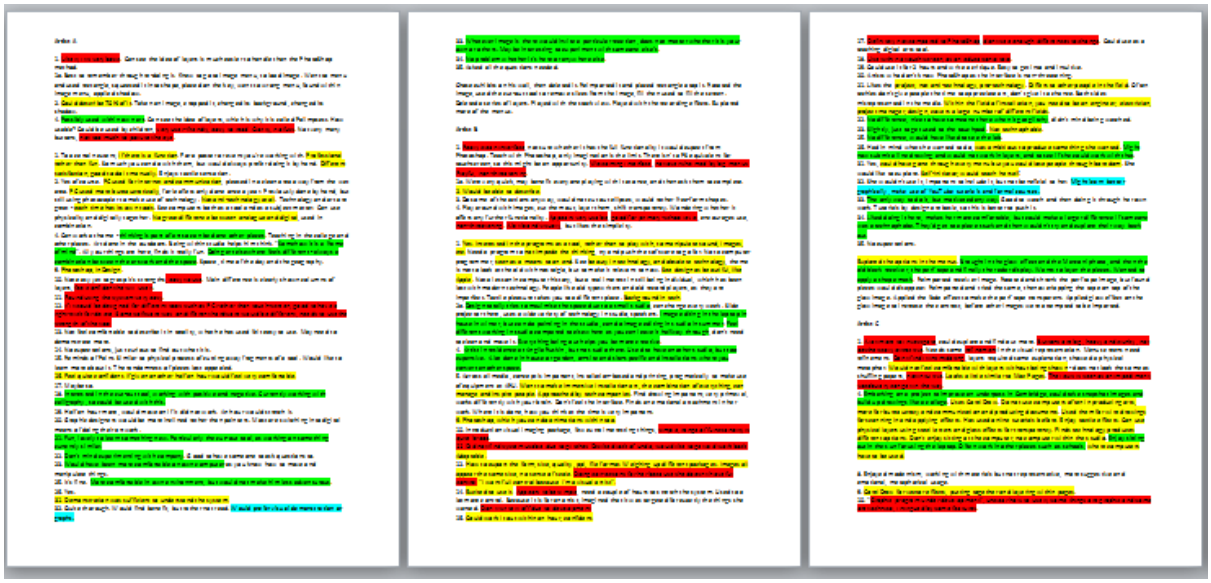


Figure 3 – Concept highlighting as part of Grounded Theory

6. Conclusions

The conclusions for this study focus upon numerous topics, both methodological and practical. These findings shall be divided in the following manner.

6.1. Self-Efficacy

The four self-efficacy questionnaires displayed highly differing levels of confidence concerning computer use. Whereas Artist A did not feel confident that they could use new software without assistance, and primarily used computers for more bureaucratic purposes, Artist B strongly believed that they could understand a novel package, having undertaken programming before and made use of editing tools. Neither Artist C nor Artist D confessed to enjoy experimenting with computer tools, but whilst the former only used devices for bureaucratic tasks, the latter made use of image editing software and saw the utility in artistic programs, but believed there were too many different packages to learn. All participants felt far more confident in completing a task if they were able to ask another for assistance, though the removal of time-constraints generally did not seem to affect their self-efficacy. Those who enjoyed tinkering and displayed confidence in computer use were able to interact with Palimpsest with greater ease during the free experimentation stage, requiring less assistance and being more assertive with their choices.

In comparing our findings to the previous study, there exists many instances of contrast. Whereas the 2013 work presented that those possessing the highest levels of self-efficacy in both visual arts and computer use found the most utility in the system, in this case a different result is seen. Artist C experienced several usability challenges with the system and therefore did not make meaningful progress during the free play period; the same participant explained they did not enjoy experimenting with software and possessed a relatively low self-efficacy score. However, although Artist B saw utility in the tool, they also experienced difficulty through attempting advanced functionality which was not present within the system. This individual was thoroughly acquainted with graphics editing tools, possessing high self-efficacy, but became frustrated that Palimpsest often could not undertake the same actions. In hindsight it would have been preferable if entirely novel functionality was presented to the participant, rather than some graphical editing effects in attempt to capture their interest. Both Artist A and D enjoyed making use of the system, believing there were possibilities for

it to be used within their own work. Although neither participants possessed very high computer use self-efficacy, they were experienced with technology, and therefore not intimidated by the tool. It appears that in these cases, where individuals both are not confused by the software and also understand its limitations, that participants see most utility in the system.

6.2. Professional Environment

All four participants believed that their professional environment is an important part of their art. Artist A believed that the studio is "somehow a frame of mind"; Artist B liked that artworks could be left out which assists the creative process; Artist C possessed numerous studios and believed the environment had a great effect; and Artist D confessed to feeling different within their studio, despite the portability of a laptop. Interestingly though, opinions differed whether the location of the study affected their experience with the system. Artist A felt more comfortable within their own environment, but did not believe it made them more adventurous, whilst Artist C confessed to wanting to view the interesting location of the Computer Laboratory. In contrast, Artist B enjoyed performing the study within a more personal environment, believing that "technophobic" individuals might be daunted in an alien location and less likely to experiment. Artist D described themselves as not a "typical studio artist", and did not think this task would differ greatly in another location, but that larger efforts might be affected. They also mentioned that for creative ideas it is preferable to not be observed, representing that a typical studio environment might be beneficial. It should be noted that this question was posed after the Palimpsest free experimentation session was completed and, although the artists might openly state that they would not have objected to an alternative venue, it is challenging to predict whether they actually would have acted in a different manner.

6.3. Meaningful Artwork

The studies also concerned interactions with artwork meaningful to each individual artist, ensured by requesting participants submitted a selection of images that reflect their artistic style. It was seen that most of the participants appreciated interacting with specific artworks, as a means of making the experience more engaging. Whereas Artist A stated they might have enjoyed using another artwork as a means of viewing an interesting new artefact, Artist B believed that their image allowed them to begin the task with a clear objective. Artist C differed slightly in that they thought that being challenged to replicate a completed project might be more instructive, but Artist D found that interacting with their chosen art was inspiring and revealed more possibilities than using simple shapes. Unfortunately Artist C encountered numerous difficulties regarding the Palimpsest user interface, expressing that they felt constrained, being unable to interact with their artwork in the way that they wished. If a higher computer use self-efficacy was possessed, it might have been possible for this participant to perform the actions they desired, and hence possibly regard the task as more similar to their professional work. It should also be noted that neither Artist A nor Artist C undertake graphics editing within their own work; therefore it is likely they would not experience a connection so readily as those who perform these actions on a daily basis. It is arguable that artists shall possess strong emotions regarding their artworks and therefore this might have had an effect on their interaction with Palimpsest. Whilst this is true, individuals could react in highly differing ways: embarrassed to use their personal artefacts, unwilling to modify a precious piece, intrigued to see how their work could be developed, or proud to present their favourite artwork. For this reason, it is not obvious to conclude that interacting with personal artworks shall assist a professional artist to engage with Palimpsest to a greater degree.

6.4. Palimpsest Utility

Participant opinion on the Palimpsest system differed across many topics, but also displayed consistent similarities. Artist A found the interface "clear" and "basic", finding the arrangement of layers easier to understand than within commercial graphics editing packages. They also believed that the "cut-out" function of the system offered benefit and possessed similarities to actions undertaken in their non-technological professional work. Artist B also praised the "clean interface" and believed it "non-threatening", but found the lack of advanced image editing functionality an issue. They accepted

that the tool could perform complex actions which other packages could not, and believed the system could be useful as a niche product for touchscreen devices. Artist C stated they did not find the interface intimidating, but disliked the aesthetic presentation of both the layers and menu buttons. Artist D similarly regarded these buttons as "clunky", but appreciated the clear feedback, "direct" interface, and believed the product to be "very accessible". Both Artists B and C commented upon the lack of "undo" functionality as negatively affecting their experience, but all participants stated they could see some application within their own professional work. Artists A and B both believed the system could be used to teach graphics editing to children, whilst D saw the tool as valuable for mock animation. Opinions appeared to differ based upon computer use self-efficacy levels and background: Artist B was confident and hence found the system constraining, whilst Artist C disliked experimenting with technology and therefore regarded the unfamiliar interface to be confusing. Artists A and D did not possess particularly high self-efficacy levels but were not intimidated by the tool, and hence were able to both make progress during the free play session and see real opportunities for the system within their own work.

It was also found that three of the four artists admitted that they often learned the most effectively through graphical means or demonstration. Artist B and D both admitted to using YouTube tutorials for learning complex concepts, whilst Artist A disliked text-based instructions and preferred graphs. Currently the Palimpsest tutorial is provided through a linear, 53-step, textual process, and several participants commented that they might prefer an alternative approach. Artist C, who experienced difficulties interacting with the system, found the help menu lexis to be "confusing", and expressed that they would have responded more favourably to examples. These findings suggest that artistic users might learn more effectively if graphical or video-based instructions were included within the Palimpsest system. Though the exploratory period was only twenty minutes, therefore making a complete understanding of the system challenging, this study reflects upon artist's initial reactions to Palimpsest and the utility they believe it can bring to them. A far longer demonstration and free-play period would be required to establish total understanding; less feasible when studying professional artists working within their own professional environments and looking to emulate the practice of working rather than tuition.

6.5. Champagne Prototyping Extension

The key methodological concept within this paper was the development of a novel Champagne Prototyping extension. To best analyse the utility which real artists found within the Palimpsest tool, participant observation was undertaken within their own professional environment, and using artefacts which they regarded as typical of their artistic style. All participants stated that they felt comfortable within their studios, with Artists B and D also believing that modifying their own artworks assisted them in seeing a clear objective for the task. With all artists also seeing some utility for the tool within their own work, it is possible that this methodological extension assists participants in viewing the actions within the study as not greatly dissimilar to their own professional tasks.

Further Work

Further work could be undertaken to explore interesting concepts from this study. One user commented that they would have been able to evaluate the system more comprehensively if the software was distributed in advance and then participants were asked to create an artwork which they found interesting. Palimpsest could be emailed to participants with the tutorial included, and two weeks later asked to present their product and explain their experience of the process. In this situation an artist would be less constrained by temporal pressures and have a greater opportunity to fully explore the system; it would be of interest to study the utility that was found in this situation. To test the learnability of the software, a participant could be presented with a demonstration of the system, and their recall on how to perform a task could be assessed at varying time intervals; perhaps immediately, one day, one week and one month. As a means of further exploring whether the environment in which an action is undertaken, or the customisation of an artwork, has an effect on participant experience, it would be possible to assess ten artists within laboratory conditions with

general tasks, and ten artists within their own studios and using their own art, before comparing their responses.

Acknowledgements

I would wish to sincerely thank Issam, Diana, Melissa and Bettina for dedicating their time to participate in the study. I would also wish to thank Kath Powlesland for taking part in the pilot observation, which taught valuable lessons in performing semi-structured interviews.

References

Atkinson, P., & Hammersley, M. (1994). Ethnography and participant observation. Handbook of qualitative research. Thousand Oaks, CA: Sage.

Bandura, A. (1977). Self-efficacy: toward a unifying theory of behavioral change. Psychological review, 84(2), 191.

Bandura, A. (1982). Self-efficacy mechanism in human agency. American psychologist, 37(2), 122.

Beckwith, L., Kissinger, C., Burnett, B., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. (2006). Tinkering and gender in end-user programmers' debugging. In Proceedings of CHI 2006, 231-240.

Beyer, H., & Holtzblatt, K. (1997). Contextual design: defining customer-centered systems. Elsevier.

Blackwell, Alan F. (2002). First steps in programming: A rationale for attention investment models. IEEE Proceedings of Human Centric Computing Languages and Environments, 2-10.

Blackwell, A., & Burnett, M. (2002). Applying attention investment to end-user programming. In Proceedings of Human Centric Computing Languages and Environments, 28-30.

Blackwell, A. F. & Green, T. G. (2003). A Cognitive Dimensions questionnaire optimised for users. In A.F. Blackwell & E. Bilotta (Eds.) Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group, 137-152.

Blackwell, A. F., Burnett, M. M., & Jones, S. P. (2004, September). Champagne prototyping: A Research technique for early evaluation of complex end-user programming systems. In 2004 IEEE Symposium on Visual Languages and Human Centric Computing, 47-54.

Blackwell, A. F. & Charalampidis, I. (2013). Practice-led design and evaluation of a live visual constraint language. University of Cambridge Computer Laboratory Technical Report 883.

Burnett, M. M., Atwood, J. W., Djang, R. W., Reichwein, J., Gottfried, H. J., & Yang, S. (2001). Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. Journal of functional programming, 11(2), 155-206.

Compeau, D. R., & Higgins, C. A. (1995). Computer self-efficacy: Development of a measure and initial test. MIS quarterly, 19(2).

Eckert, C., Blackwell, A., Stacey, M., Earl, C., & Church, L. (2012). Sketching across design domains: Roles and formalities. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 26(3), 245-266.

Glaser, B. G. (1978). Theoretical sensitivity: Advances in the methodology of grounded theory (Vol. 2). Mill Valley, CA: Sociology Press.

Glaser, B. G. (1998). Doing grounded theory: Issues and discussions. Sociology Press.

Strauss, A., & Corbin, J. (1994). Grounded theory methodology. Handbook of qualitative research, 273-285.

Ghosts of Programming Past, Present and Yet to Come

Russell Boyatt Meurig Beynon
Computer Science
University of Warwick
{rboyatt, wmb}@dcs.warwick.ac.uk

Megan Beynon
C3RI
Sheffield Hallam University
megbey@gmail.com

Keywords: programming; computational thinking; making construals; Empirical Modelling; music

Abstract

Twenty-five years ago, when the Psychology of Programming Interest Group (PPIG) was established, the concept of programming seemed rather robust and clear. With Turing's characterisation of algorithms at its core, its meaning could be safely broadened to include the many peripheral activities surrounding the production of software that – as can be inferred from the Call for Papers – have become the natural habitat for PPIG. Today, the nature and status of programming is much murkier. For instance, in his state-of-the-art reflections on programming [13], Chris Granger poses the questions *What is programming?* and *What is wrong with programming?*. Despite the great promise – and perhaps even greater expectations – for applications of computational thinking, programming issues that were perceived as of the essence are no longer so prominently represented in computing-in-the-wild. For instance, the programming languages SCRATCH and Go – with radically different motivations – shift the focus from programming paradigms, formal programming language semantics and sophisticated abstract programming constructs to highly pragmatic issues relating to the experience of the developer. This paper makes the case that in understanding this intellectual development in approaches to program-related activity it is helpful to stop stretching the notion of 'programming' ever wider and to appeal instead to a broader complementary concept. To this end, in this paper, we propose to confine the term 'programming' to the authentic meaning it acquires through Turing's characterisation, and address the wider agenda to which it has been enlisted – and through which its meaning has been adulterated – by introducing the notion of 'making construals'. This proposal reflects twenty-five years of parallel development of Empirical Modelling, an approach to computing that has taken much inspiration and encouragement from the PPIG agenda [7, 9, 11].

Introduction

PPIG stands at the intersection between computing and psychology. Its primary concerns are the *psychology of programming* and *computational aspects of psychology*. When PPIG was conceived in the late 1980s, it was particularly fashionable to regard *computation* as central to the connection between computing ("computational thinking" [23]) and psychology ("the computational theory of mind"). On that basis, it might have seemed that research on programming paradigms and methods of formalising software development would prove to be the driving forces behind subsequent new developments that have brought computing to the forefront of so many people's lives.

In practice, the vibrant culture of computing-in-the-wild that has been so well-represented in PPIG conferences over the last twenty-five years seems to have thrived on a raw pragmatism that admits only a vicarious relationship with its abstract computational roots. For instance:

What is the role for standards and formal specifications of software systems? – witness the pragmatic way in which the communications software behind the internet has been developed.

What is the contemporary status of Codd's elegant mathematical relational model for databases? – witness the logically flawed but pervasive implementation of SQL relational databases, and the adoption of alternative database models [5].

What prospect is there that new programming paradigms can resolve the problems of software development? – witness the sterility of the forty year old controversies that have surrounded different paradigms and the rise and demise of OOPSLA.

The disconnect between theory and practice reflected in these questions and responses has great relevance for PPIG – it is a symptom of topical concerns about the nature and status of programming itself. In a very recent post [13], the software developer Chris Granger identifies problematic aspects of programming and poses the question *What is wrong with programming?*. After talking to about four hundred people with a wide variety of perspectives on programming, he concludes that 'programming is our way of encoding thought such that the computer can help us with it' and what is wrong with it is that programming is *unobservable, indirect* and *incidentally complex*.

Granger's conclusions are revealing. They point to technical issues to be addressed in developing programs of just the sort that have been the focus of interest for the PPIG community. They also highlight a more fundamental conceptual problem. Programming has unambiguous historical roots in the work of Turing; in no way is programming intended for 'encoding thought' – it is what is appropriate for specifying algorithmic behaviours conceived by 'a mind following rules' [20,6]. And whereas Granger is here observing Bret Victor's injunction: "we must change programming" [22], it is more appropriate to seek an alternative concept – *complementary* to programming – to account for computing practice.

Our title and the character of our message take their inspiration from Charles Dickens's *A Christmas Carol*. Though the celebrated ghosts who haunted Scrooge had a common mission, their perspectives were individual: they surveyed the past, the present and the future and found in each a different provocation. In a similar spirit, the three perspectives in this paper – past, present and yet-to-come – reflect a common aim but are informed by different kinds of experience. These are broadly correlated with the experience of the three authors of, respectively: finding ways to teach programming to novices that can provide an appropriate foundation for a mature understanding of computing; establishing a conceptual framework for studying programming that does fuller justice to computing-in-the-wild; making the transition from specialist work on software integration to teaching music.

The Ghost of Programming Past

In which the Ghost of Programming Past visits schools to find old initiatives for teaching programming renewed – and facing familiar challenges, and is perplexed by new ventures that sidestep the wisdom of the ancients to promote programming as a lived experience.

Teaching computer programming – yet again!

Understanding and teaching computer programming has been a central concern for PPIG. The challenges that this presents have been highlighted by topical developments in UK schools.

The status of Computer Science (CS) in UK schools is changing, with the challenging nature of the discipline becoming more widely recognised. In January 2012, the UK ICT national curriculum was disapplied, signalling the shift away from ICT skills, e.g., teaching students how to use proprietary software packages, towards a broader range of CS skills and knowledge. These changes have been endorsed by initiatives such as *Computing at School* that are committed to building resources and support networks. However, understanding the requirements of both teachers and students is complex and time consuming.

Adopting an enhanced computing curriculum taxes teachers across all levels of school education. The first author's experience of engaging with school teachers demonstrates that a significant number of teachers are struggling to develop the skills and resources required to deliver academic CS content. These difficulties arise for a variety of reasons: many teachers have educational backgrounds unrelated to CS, have to cope with real and perceived time pressures, and lack experience with appropriate pedagogy. Further, few schools have the flexibility or resources to allow teachers the time or funding to attend courses. While myriad websites, resources and forums are available for sharing and discussing, many teachers currently find it difficult to acquire the skills they need and to put this content in the context of a classroom delivery.

Students must be taught to develop the cognitive and computational thinking skills, and not simply skills and knowledge specific to applications or programming languages. Though programming is a

core skill for Computer Science, it is only a preliminary step towards appreciating programming in its relation to the subject. And though concrete experience of what is involved in programming is essential, teaching someone to program in a specific language does not, in itself, convey an understanding of relevant general Computer Science concepts. For example, the process of wrestling with the syntax and semantics of C or Java can *detract* from the idea that programming is based on a set of abstract, language-independent, principles of computational thinking.

Appreciating the place of programming within the broader framework of practical computing is even more challenging. In this context, computer programming can be viewed from two perspectives:

- as an abstract activity concerned with devising recipes to establish a specified functional relationship between input and output;
- as a core activity for the implementation of software applications, where the concrete real-world interpretation of abstract program constructs, such as variables, must inform the practical construction of programs.

A further distinction must be made where software development is concerned. If the design of a software application is routine then the machine-like agency that is exploited in programming is easily identified and stable in character. Such applications are amenable to the principles and techniques of computational thinking, where the programmer's task is to figure out the machine instructions that specify a required behaviour. If a software application demands radical design, the processes of prescribing computations and of identifying the supporting agency are entangled in such a way that a less constrained way of conceiving programming activity is required. In effect, the construction of software must proceed in parallel with learning about the application domain and the empirical identification of the machine-like agency to support its implementation [8].

Towards programming as a lived experience

Grasping the syntax necessary to write a program, and being able to figure out how this prescribes a behaviour, is the first step towards 'understanding programming'. What distinguishes the experienced programmer from the novice is the degree of fluency with which they can connect program code with a behaviour. The novice needs much practice to acquire the ability to interpret a program without having to analyse it laboriously symbol-by-symbol and statement-by-statement and perform a conscious process of interpretation.

In the first instance, understanding a program involves knowing how it prescribes the behaviour of the computer. For this purpose, analysing a program step-by-step is a mechanical process of translation that in principle requires knowledge of how the computer itself works. In practice, the programmer is typically insulated from the details of the machine-level execution through a high-level programming language. The behaviour of a procedural program, for instance, is to be conceived in terms of variables and data structures whose values change. In making this translation to a behaviour, there are specific rules to follow, and these apply in all contexts.

One of the conceptual difficulties faced by the novice programmer is that understanding how program constructs manipulate variable values is not enough. These manipulations of state have to be related to the function of the program, and further interpreted in terms of the application domain. For the experienced programmer, the process of interpreting program constructs as abstract state manipulations becomes routine and potentially subconscious. Making the further connection between the program and its function is more challenging, and this skill is much more difficult to learn. It is important from the very beginning, however, since the learner is soon obliged to consider how to develop a program to carry out a particular task.

It is in this broader sense of 'program understanding' that the psychological aspects of programming are most topical. In interpreting a program in functional and contextual terms, the programmer typically gains clues from the informal characteristics of the program. The layout of the program can be important in recognising the flow of control and the significant phases in the program execution. The names of variables may connect them with observables in the application domain. Comments may be used to document the behaviour and guide the human interpreter.

The traditional emphasis in teaching computing has been on teaching the skills and concepts behind 'computational thinking'. The presumption that underpins academic computer science, and dictates the core characteristics of the ‘new’ computing curriculum for schools, is that computer science education is first and foremost concerned with the fundamental notion of ‘algorithm’ and the profound mathematical insights about the scope and limitations of this concept. On that basis, the most important consideration is that students learn how to express computational recipes for machines in an abstract programming language. The above discussion highlights a complementary perspective on programming activity – one in which the PPIG community has been prominently engaged – where the emphasis is on the moment-by-moment experience of the programmer and the mental processes involved. This shifts the focus from the program as an abstract product to programming as an activity that, from an experiential perspective, can be helpfully likened to a musical performance.

When viewing programming as a moment-by-moment experience, it is helpful to adopt an alternative broader perspective than computational thinking alone affords. This reflects the wide variety and scope of questions that a novice programmer typically has to address, such as: *'what is the syntax error in my code?', 'what if I change the value assigned to this variable at this point?', 'what if I reorder the statements in this loop?', 'how can I modify this statement to achieve the intended effect?', 'what if the input to the program has the following character?'* To answer such questions, the learner needs an appropriately rich mental model of a program that embraces more than abstract logical analysis alone affords. The computer itself can give direct and immediate support for such mental models. This is illustrated in many contemporary environments to support learning programming:

The Scratch environment [19]: In Scratch, a program is assembled using a visual interface, thus eliminating syntactic errors and complex syntax. The appeal of the environment owes much to the fact that pre-programmed input and output methods for accessing devices such as sensors and actuators and resources such as video can be readily integrated into programs without specialist knowledge. Scratch programs are relatively unsophisticated but can be readily published to social networks.

The W3Schools environment for web development [24]: One of several similar environments to enable non-specialists to develop and program webpages, W3Schools supports editing of code fragments in conjunction with the webpages they describe, conveniently set up so that a programmer can experiment by modifying code and observing the impact. This exploits a distinctive characteristic of browser environments: their robustness when programs are syntactically or logically incorrect.

Bret Victor's Learnable Programming (LP) environment [22]: Victor has developed a remarkable prototype environment to convey his vision for 'learnable programming'. This environment focuses squarely upon supporting the novice in making the direct association between code and meaningful behaviour. Victor's vision is close in spirit to the idea of programming as performance, as is illustrated by the way in which his virtuosic real-time demonstrations incite his audience to applaud [21].

All three environments illustrate a trend towards giving greater prominence to the actual experience of the programmer and the way in which this informs program development. Characteristic of this experience is its holistic nature. The programmer interacts in one and the same environment to modify the programming code, to test the effect of executing a code fragment, to see whether the program gives plausible output on different inputs, to inspect the behaviour of the program and correlate this with the code state-by-state. Throughout the development, the programmer shifts between using and developing roles in the spirit of Papert's constructionism [17]. The aspiration for the environment is to support the development of a program through self-directed learning conducted by experimenting with the code and observing the impact of changes immediately. Shifting the focus from thinking about behaviour in purely mechanistic computer-oriented terms to attending to how meaningful changes are effected is a critical step towards meeting this aspiration. This shift is particularly well-represented in Victor's work, where there is strong support for freely modifying variable values and observing the effect as in spreadsheet-style dependencies. For instance, as is illustrated in Victor's serendipitous discovery of a visual animation effect that moves the blossom on a tree as if in a wind [21], changes that are first explored manually with the idea of comprehending the code can be re-purposed through automation. But the constructionist ideal cannot be realised without setting to one side an essential characteristic of *programming*: prescribing a recipe to achieve a functional goal [2].

The Ghost of Programming Present

In which the Ghost of Programming Present points out the direct relevance of established principles for making construals to the challenge of realising programming in lived experience, and the merits of adopting an empirical and pragmatic semantic stance that is broader than computational thinking.

A conceptual framework for 'programming as a lived experience'

The conceptual framework of 'computational thinking' is not in itself sufficient to understand the character of the support for the novice programmer's mental models represented in the emerging programming environments mentioned above. The focus in such environments, unlike Turing's focus, is on the states of mind of a person *devising* rather than following rules. By way of a simple illustration, in Victor's account of creating the effect of wind on blossom, he is evidently concerned with the experience and support that the computer offers in the process of developing a program whose precise functionality is emergent. What works depends on pragmatic considerations such as the speed of the processing, the colour and resolution of the display and the cultural background and imagination of the human observer. As Victor himself emphasises, the response to the feedback from the experimental construction is more than prosaic 'modelling of a requirement' – open-ended exploration of what is feasible and effective by way of dynamic visualisation is involved.

To express the creative nature of the role being attributed to the 'programmer' in this setting, the term 'maker' is adopted. The maker engages with activities that reflect an uncertainty about the scope and potential for program development – and indeed for computer application – that can derive from many sources. For instance, it may stem from the status of a task as (e.g.) as yet ill-specified, or essentially concerned with experiential issues, or presenting a challenge in radical software development, or acknowledging a novice's limited understanding and prior experience of the capability of a machine. The key components representing in 'making' are *constructing* and *learning*, and this choice of term reflects the essential creative aspect of constructionist learning.

To do fuller justice to the experiential aspects of programming activity, it is helpful to invoke the modelling framework that has been developed for Empirical Modelling (EM) [25] depicted in Figure 1 below. The true significance and nature of the concepts represented in Figure 1 will be clarified later, when they are applied to programming as approached from an EM perspective.

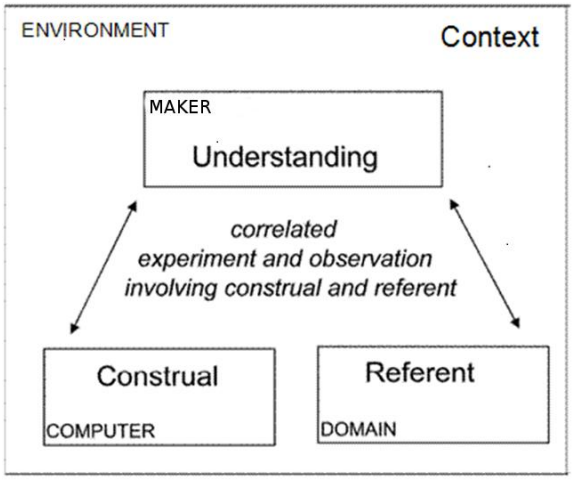


Figure 1 Making an EM construal

It is helpful to give a brief characterisation of the four principal ingredients depicted in the diagram: the *construal*, its *referent*, the maker's *understanding* and the *context*. Figure 1 should be interpreted as depicting the way in which these ingredients are at all times related, though all four evolve throughout the process of learning and development. With reference to Figure 1:

The **construal** refers to the computer-supported artefact within the programming environment that helps the maker to realise their mental model. The scope of the construal reflects the many ways in which the maker can observe and interpret the program and modify it interactively. It will typically

include an environment for editing the program code, assembling the program from visual components (cf. Scratch), allowing the values of variables to be reassigned using sliders as in Victor's LP environment, observing the impact of changes to program code upon behaviour etc.

The **referent** is the behaviour associated with interaction with the program as apprehended by a user. Depending on the current context, this may be behaviour in the application domain (e.g. the rippling movement of blossom on a tree), or a more prosaic machine-oriented interpretation of behaviour.

The maker's **understanding** takes the form of tacit personal knowledge of skilful interaction, familiar interpretations, and expectations that the maker acquires through experience of interaction with the construal and with its referent. This typically evolves moment-by-moment as the program development and use proceeds, and goes beyond knowledge of abstract logical relationships.

The **context** for interaction with the construal and its referent is dynamic and potentially volatile. It reflects the motivations of the maker from moment-to-moment, and the roles they adopt. The variety of ways in which the maker can interact is well-illustrated in Victor's LP environment [22], where sometimes interaction is for the purpose of demonstrating what has been established, is sometimes open-ended experiment, sometimes concerns the way in which program code is framed, is sometimes carried out with the interpretation of the execution of a programming construct in mind, etc.

The correlation between interaction with the construal and its referent can take many forms. Expert functional programmers may be most adept at looking at a functional program and 'instantly' inferring what abstract input-output relationship it prescribes. Their expertise is of the same kind that applies to skilful programmers using any paradigm: they can configure code in such a way that its behaviour is intelligible and amenable to adaptation in meaningful ways, they can readily conceive program code that can realise a given behaviour. Adopting an EM perspective involves taking proper account of the experiential character of programming. Whatever programming paradigm is involved, the expert programmer has to be able to recognise the connection between 'looking at the code' and 'imagining a behaviour', and this involves correlating two independent experiences. As Victor remarks in [21], this process of correlation is easier to interpret in experiential terms if it can be represented using explicit visualisation as a relation between states rather than behaviours (cf. the way in which the movement of a character in a game is animated using his environment by first creating a time-independent representation [21]). The shift in emphasis towards learning through interaction with artefacts that this entails naturally leads Victor to invoke Papert's constructionist stance (cf. [17]).

More about construals

The character and effectiveness of environments that use visualisation and interaction to help the programmer appreciate the relationship between program code and the behaviour it prescribes depends critically on the programming paradigm, programming language and specific details of the program code. A program devised by a novice may prescribe the correct behaviour, but do so in a way that is hard to understand and difficult to adapt. Were we to experiment by changing the values of the variables within such a program in the spirit of Victor's LP environment, the effects might be hard even for the programming expert to predict. Declarative programming techniques were first introduced because of the perceived difficulty of imagining the behaviour of procedural programs and relating this to meaningful behaviour in the application domain. This difficulty cannot be attributed solely to bad programming practice. For instance, some of the most ingenious and efficient algorithms achieve their results in ways that render them brittle and hard to understand. This characteristic of procedural approaches is not easy to reconcile with the fact that Papert advocated the procedural language Logo as a vehicle of constructionism [17,10,2].

More insight into this problematic aspect of traditional programming can be gained by looking more closely at what studying the experiential aspects of programming activity from an EM perspective entails. The key concepts in EM are *observables*, *dependencies* and *agency*. An **observable** is something to which the maker can ascribe an identity and current status. A **dependency** is a predictable synchronisation of changes to observable values that is perceived by the maker. The current state of an EM construal is determined by the current status of its observables and changes to this state are effected by redefining the status or values of observables or reconfiguring the

dependencies to which they are subject. All such changes of state are attributed to **agents** – these include the maker and agents whose interventions are automated.

These concepts can be interpreted with reference to Figure 1, as it applies to the development of conventional programs within environments of the kind mentioned above. Within such an environment, the observables would include syntactic characteristics of the program code, such as the variables and their names, the code layout and the components of the program structure in so far as these could be stably identified, and the attributes – size, colour, style – of the font in which the code was presented. They would also include the status of any of the interface widgets for manipulating the code, and features of any diagrammatic representations that assist in the visualisation of behaviour. An example of a dependency might be the relationship between the position of a slider and the value of variable as in Victor's LP environment [22].

The correlation between the construal and its referent depicted in Figure 1 is between the observables, dependencies and agency in the construal and the referent. In effect, there should be metaphorical counterparts for the patterns of observables, dependencies and agency encountered in the referent within the construal. It is typically quite difficult to account for the relationship between a procedural program and the behaviour it prescribes in these terms. Features exemplified in Victor's LP environment can be helpful in this respect, but a more radical shift in perspective is ideally required.

Empirical Modelling

The interpretation of Figure 1 takes on an entirely different character if we substitute a spreadsheet for a program (cf. Granger's observations in [13]). In a spreadsheet, cells and their values are 'live' observables with which the maker can interact 'as-of-now', and the definitions of values in formulae express dependencies latent in 'as-of-now' interaction. In these respects, the cells of a spreadsheet typically have direct counterparts in the external world to which they relate in a live fashion amenable to experimental validation. By contrast, the correspondence between variables in program code and meaningful observables in the application domain typically has a nominal abstract quality and may be sketchy or tenuous. As the use of spreadsheets as an instrument for *what-if* analysis confirms, there is a very direct correlation between interaction with a spreadsheet and interaction in the world. The very formulation of the spreadsheet reflects what we understand as sensible interaction in the application domain. And whereas it is hard to predict whether changing a variable in a procedural program has a familiar, plausible or absurd real-world interpretation – or indeed admits any interpretation at all, possible ways of redefining the cells of a spreadsheet afford just such nuances of meaning.

EM is a practice in which the focus is on making construals using specially-adapted principles and tools [25]. The main tool developed for EM has been the EDEN interpreter. This allows the values of observables of diverse kinds to be freely specified by definitions similar in character to the definitions of cell values in a spreadsheet. A family of definitions framed in this way serves as an interactive environment that corresponds to an external referent in the manner that is depicted in Figure 1. Many varieties of agent interaction can then be realised by redefining observables. Such interactions are typically first implemented manually through the direct intervention of the maker. Program-like behaviours can be identified by refining the family of definitions and exploring patterns of agent interaction that can admit appropriate meaningful interpretations. In this respect, making a construal is a more general activity than writing a program. It has the virtue of enabling the user to frame the manual and automated agency involved in implementing the program with explicit attention to which observables are changed and what dependencies these observables are subject to. This is consistent with the need in general in programming reactive systems to engineer the reliable interaction between agents that is taken for granted when programming a conventional computer [8].

For a full discussion of EM, the interested reader may refer to [4] and related publications on the EM website [1-11]. A noughts-and-crosses construal from which a suite of programs for playing games with a family resemblance to noughts-and-crosses, developed using EM principles and implemented using EDEN, was presented as an illustrative example of EM in the 6th PPIG conference in 1994 [9]. A version of this construal that has been re-implemented using JS-EDEN, an online version of EDEN currently under development, can be accessed via the JS-EDEN interpreter [26].

The Ghost of Programming Yet to Come

In which the Ghost of Programming Yet to Come champions music as the inspiration for blends of human and computer agency beyond the user model, connects large-scale software integration with a paradigm for personalised music-teaching, and notes the virtues of 'making construals' as a better instrument than 'programming' for making music.

Computing and music-making

When viewing programming activity as resembling performance, it is natural to make analogies between writing programs and composing music in an improvisatory style. Bret Victor makes this analogy when discussing learnable programming [22]: *"An essential aspect of a painter's canvas and a musical instrument is the immediacy with which the artist gets something there to react to. A canvas or sketchbook serves as an 'external imagination', where an artist can grow an idea from birth to maturity by continuously reacting to what's in front of him."*

Such an analogy invites further analysis and critical evaluation. Nothing in the abstract theory of computation refers explicitly to the idea that a machine offers experiences to the programmer in the way that a musical instrument *of its essence* does. As commentators on Victor's LP essay have observed, the concept of responding to the visual feedback that the program affords makes a great deal of sense for a JavaScript program that specifies an animated image on a webpage, but does not apply so naturally to typical systems programming tasks. The element of performance in Victor's manipulation of the JavaScript code that is so much in evidence in his *Inventing on Principle* talk [21] relies crucially both on the character of the programming environment and the precise nature of the interactive transformations he performs. As Victor himself makes clear when criticising the environments for learning programming developed by the Khan Academy [15], the principle of changing parameters in program code and observing the consequences makes sense only when used in a context that is suitably engineered and annotated. Without imposing such a context, the practice of editing program code and 'seeing what happens' typically leads to confusion and incoherence. The notion that certain principles are being respected in this empirical approach to developing programs is implicit in Victor's claim that LP is an archetype for constructionist learning in Papert's sense.

Central to Victor's LP is the idea that a programmer should be able to understand the correlation between program code and the behaviour it effects in a manner that can be traced directly to their immediate experience. As has been argued in many EM publications [2,4,5,10], processes of construction that rely on establishing relationships that are directly experienced are more appropriately conceived as 'making construals' rather than 'writing programs'. Substituting "making a construal" (as described in a previous section) for "writing a program' resolves those discrepancies that make the analogy between creating artefacts by computer and playing a musical instrument problematic (cf. [3]). Construals are essentially concerned with exploiting the computer as a source of interactive experience; they establish a playground for agency that subsumes the roles of learners, developers, users etc and allow these to be blended and distinguished at the maker's discretion; their construction reflects the way in which what can be directly experienced evolves through the refinement of representations and acquisition of skills on the part of human agents. In these respects, substituting 'making construals' for 'writing programs' liberates far richer visions for human agency within computing environments than is enfranchised by the notion of 'computational thinking' alone.

Software architecture and principles for music teaching

Further insight into the parallels between computing and music can be gained from the third author's experience first as a software engineer specialising in software integration and subsequently as a music teacher.

In the development of software solutions, a major challenge is to introduce software into an existing environment in such a way that it will integrate with many different applications on many platforms with diverse configuration. For large software systems, this presents problems that cannot be resolved simply at the specification level; the complexity of the interactions between software components and the diversity of the environments is such that a systematic iterative process of implementation, testing

and refinement has to be followed. The Google developer Rob Pike's keynote talk about the programming language Go at SPLASH 2012 [18] highlights the pragmatic nature of the issues that are topical in this context (cf. slide 11, which lists concerns that are not addressed by language *features*). For instance, a key goal in designing Go and its development environment was to reduce the compilation time for systems so as to make the iterative process of deployment more efficient.

At this high level, software engineering can be identified as the deployment of a 'system' in an 'environment'. By analogy, teaching an instrument can be seen as imparting a skill set to the learner. The skill set can be seen as the counterpart of the system, and each learner as a specific environment. Within the skill set, each skill has similar status to a component of the system. The dependencies between skills guide the way in which content should be delivered to the learner in much the same way that dependencies between components of a system guide its architecture and design.

Just as a system is more than its set of components, and serves some overall functional aims, the skill set to be imparted to the learner is directed at achieving specific goals, such as being able to play a particular piece, or selection of pieces within a particular idiom. In implementing a system, designing the set of components that will be deployed for minimum viable product is the starting point, and the entire architecture of the system is also understood, developed, and tested for each environment as time goes by. In a similar spirit, it is appropriate to identify a basic skill set that is needed to master a particular repertoire, and to consider what is involved in teaching this to each individual learner.

This establishes a parallel between the role of a software engineer in system architecture and design and the role of the instrumental teacher in planning the delivery of content.

Once the system architecture has been designed, the implementation and installation of the software can begin. A particular coder then takes responsibility for the implementation of a component. The instrumental teacher takes on a role resembling that of a coder when devising ways in which to impart a specific skill – for example, framing a plan to teach reading the notes using a mnemonic such as 'FACE corresponds to the spaces in the treble clef' to a student for the first time. In the same way that a coder develops a build process that sets up a component for a specific environment, the instrumental teacher has to devise a plan that takes account of the particular characteristics of the learner.

Once the skills have been deployed and debugged, a specific objective such as a performance of a piece can be addressed. Preparing for performance is itself an iterative development process. The coder attempts to execute the component on the environment, debugs the experience and then refines the implementation. In a similar spirit, the learner performs music in accordance with the guidance of the teacher, then refines the performance with the help of the teacher in response to the experience.

The bugs that arise in the software engineering process take several forms, each of which has a counterpart in the instrumental teaching context:

- the design of the component as a concept might itself be problematic – using mnemonics to learn to read music frequently introduces new bugs to the environment, and it is also often something that the student has covered before. There will be an existing installation of 'using mnemonics to read music' with which this installation might conflict. A decision needs to be taken about whether to try to integrate the two versions or to teach the same thing using an entirely different design e.g. reading music using no mnemonics;
- the implementation of the component is problematic e.g. once the software has been deployed it appears to function correctly at first glance, but intermittent problems emerge, revealing underlying conditions in the student environment that were not addressed in the design and implementation of the component that need to be debugged; the component must then be revised and redeployed, or replaced with a different approach;
- the delivery of the component is problematic e.g. the component is well designed for the student environment, but it was installed at a time when other crucial software was not running properly, or not running at all, or there was not enough processing power available. For example, the student was busy running another program about something else when the topic was introduced, or visual observation was required for the topic to be absorbed but their eyes were too tired;

- the environment requires pre-requisites to be installed before the component can be deployed e.g. the student needs to be familiar with the patterns and shapes of a stave and a clef before a technique for labelling parts of it can be deployed.

Perspectives on human and computer agency

A pivotal question for PPIG is the extent to which human agency can be conceived as machine-like. Some schools of thought take the extreme view that all human activity admits a rule-based account, including activities such as music-making and composition that are deemed to exhibit creativity [12].

Likening a learner to a computer environment may appear to be endorsing the idea that teaching music is like programming a machine. Certainly, a performer can ostensibly behave like a machine. Learners can be taught to "play Mozart" etc by being trained in the appropriate repertoire of skills (playing scales and arpeggios, learning classical harmonic progressions etc) required for this purpose. There are no doubt schools of instrumental teaching that could with some justification be seen as *programming* musicians to perform in a particular way.

The analogy between 'learner' and 'computer environment' here has a different significance. The qualities of computer environments are too difficult to predict and impossible to preconceive for the challenge of effective cross-platform software integration to be met by aligning all environments to a universal norm. The individual characteristics of environments have first to be discovered through empirical testing, and adapted only where necessary or desirable. A good integration strategy then exploits the distinctive pre-existing features of the environment itself. In this same spirit, music teaching can enable students to integrate new knowledge and skills with those they already have.

The peculiar blend of human and computer agency that making a construal affords is in fact well-suited to supporting teaching of this nature. Making construals gives computer support for sense-making that can in principle be applied to explore the particular individual characteristics of learners, and to identify their pre-existing skills, limitations, knowledge and misconceptions. It can also be used to train students in basic musical skills. For instance, a construal devised by the third author in connection with teaching students to read different clefs at the keyboard comprises a stave in which the bottom, middle and top lines are highlighted in bold . For each clef there is a direct correspondence (a "dependency") between pairs of adjacent bold lines on the stave and clusters of consecutive notes that lie under the five fingers on the keyboard. Such a construal hints at the subtlety of the dependency relationships that become second nature to the accomplished pianist, which may (for instance) connect the shape of the hand with patterns of notes on the page, or the lateral movement of the hand over the keyboard with up-and-down trajectories of the melodic line.

As the open-ended nature of the noughts-and-crosses construal discussed in [9] illustrates, there can be great subtlety in the human agency that underpins what appear to be simple tasks. In contexts where this agency is inconceivably rich and multifarious, making construals is more appropriate than programming as a means to enhance human agency precisely because of its essentially provisional and pragmatic character. For instance, the quality of the agency involved in playing a musical instrument is much more than acquiring a repertoire of reliable machine-like skills and learning to apprehend specific dependencies. A fine instrumentalist is far more versatile than a machine, has characteristics that are highly individual that reflect distinctive physical attributes (e.g. size of hand), strategies for performing (e.g. sight-reading vs memorising vs improvising), and capacities to interpret different modes of representation (cf. reading from solfa notation, guitar tablature or a figured bass). It is also self-evident here that the representation of music, the characteristics of the instrument itself, and the cultural context in which it is presented is critically important: cf. the practical impossibility of playing from a MIDI file or emulating quarter tones on a piano.

Teaching that promotes such an exalted view of musicianship has *giving the learner the capacity to make their own construals* as its goal. The end products to which such teaching aspires are motivation and enjoyment rather than accomplishment alone. These qualities, so difficult to sustain in a rule-based goal-oriented culture, play an invaluable role in giving meaning to our musical and human life.

Concluding remarks

The second author first ventured to pitch the ideas that developed into EM at the first PPIG. These ideas have since matured out of all recognition, especially through the contributions made by many generations of research and undergraduate project students. Particularly relevant to PPIG is the fact that the philosophical stance that best suits EM is 'radical empiricism', as conceived by William James, the 'father' of American psychology, at the beginning of the last century [14,1]. It is particularly significant that the direct associations between 'experience of the program code' and 'experience of the behaviour to which it refers' to which Victor aspires in [22] can be construed in Jamesian terms as empirically given "conjunctive relations". This decisively shifts the focus in viewing an interactive computer artefact from the abstract formal semantics derived from the program text to the meaning as it is established and evolves through direct experience of skilful interaction. This perspective, so crucial for the musician [3], is also most congenial to the culture of PPIG.

Looking to the past, it is apparent that what we understand by and expect of 'programming' has evolved to a point where serious reconsideration is required. The role of programming languages in particular needs to be reappraised, as the practices that surround constructing programs as texts are ill-suited to the purposes to which 'programming' is being recruited.

Looking to the present, there is – in Empirical Modelling – a better conceptual framework in which to view the exceptionally rich ways in which computing is being applied. This goes beyond targeting different varieties of use, or modes of development, of software that are predicated on delivering specific functionalities by exploiting established machine-like agencies. The computing activity that is associated with programming in the narrow sense now embraces support for crafting human roles beyond those of users, programmers and developers in conjunction with the agency that formerly was assumed to be given by computing platforms.

Looking to the future, computing activity must increasingly acknowledge the personal agenda of individuals in the same spirit that musical instruments and skills have to be adapted to the personal characteristics of learners, performers and composers. Turing's model of 'a mind following rules' has its place in this setting, but cannot be invoked as a way of developing and validating such rules.

This is not to deny the power and potential of a computationalist stance. Beyond question, activity that is in the spirit of programming in its authentic meaning has a secure role and glorious future prospects in contemporary culture. There is no knowing what can be achieved through the application of computational thinking backed by technological advances and algorithmic ingenuity. No doubt we shall be surprised by future applications for rule-based goal-oriented activities and new contexts in which human agents can act in the role of users.

Nevertheless: our humanity is more than can be expressed in such terms. As has been argued by Jaron Lanier [16], the pervasive status of computing and our limited conceptual grasp of its nature are in danger of imposing a computational thinking paradigm upon areas of life where it is out of place, with damaging social and psychological effects. We need a framework for thinking about computing that acknowledges the scope for more exalted forms of human agency than 'using a computer program'. No one is better placed to respond to this challenge than the PPIG community.

Acknowledgment

We are indebted to Steve Russ for helpful comments and to Tim Monks, Nick Pope and Joe Butler for their work on the design and implementation of the JS-EDEN interpreter.

References

[1] Beynon, W. M. (2005).. Radical Empiricism, Empirical Modelling and the nature of knowing. In (ed. Itiel E Dror) Cognitive Technologies and the Pragmatics of Cognition: Special Issue of Pragmatics and Cognition, 13:3, December 2005, 615-646.

[2] Beynon, W. M. (2007). "Computing technology for learning – in need of a radical new conception." Educational Technology and Society 10.1: 94-106.

[3] Beynon, Meurig. (2011). From formalism to experience: a Jamesian perspective on music, computing and consciousness. Chapter 9 in Music and Consciousness: Philosophical, Psychological, and Cultural Perspectives (ed. David and Eric Clarke), OUP, 157-178

[4] Beynon, Meurig. (2012). Modelling with experience: construal and construction for software. Chapter 9 in Ways of Thinking, Ways of Seeing (ed. Chris Bissell and Chris Dillon), Automation, Collaboration, & E-Services Series 1. Springer-Verlag, January 2012, 197-228

[5] Beynon, Meurig (2012). "Realising software development as a lived experience." Proceedings of the ACM international symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, October 21-25, Tucson, Arizona, USA, 229-244

[6] Beynon, Meurig. (2013). Turing's approach to modelling states of mind. In Alan Turing – His Work and Impact (eds. S Barry Cooper and Jan van Leeuwen), Elsevier, 85-91.

[7] Beynon, W.M., Boyatt, R. and Chan, Z. E. (2008) Intuition in Software Development Revisited. In Proc. 20th Annual Psychology of Programming Interest Group Conference, Lancaster University.

[8] Beynon,W.M., Boyatt, R.C. and Russ, S.B. (2006). Rethinking Programming. In Proceedings IEEE Third International Conference on Information Technology: New Generations (ITNG 2006), April 10-12, 2006, Las Vegas, Nevada, USA 2006, 149-154.

[9] Beynon, W.M. and Joy, M.S. (1994). Computer Programming for Noughts-and-Crosses: New Frontiers. Proc. PPIG'94, Open University, 27-37.

[10].Beynon, W.M. and Roe, C.P. (2004). Computer support for constructionism in context. In Proc. of ICALT'04, Joensuu, Finland, August 2004, 216-220.

[11] Beynon, W.M. and Russ, S.B (1992). The Interpretation of States: a New Foundation for Computation?. Proc. PPIG'92, Loughborough.

[12] Cope, D. (2005). Computer Models of Musical Creativity. Cambridge, MA: MIT Press.

[13] Granger, C. (2014). Towards A Better Programming. Available from http://www.chris-granger.com/2014/03/27/toward-a-better-programming/

[14] James, William. (1996/1912). Essays in Radical Empiricism, London: Bison Books.

[15] Computer Programming, Khan Academy. Available at http://www.khanacademy.org/compiuting/cs

[16] Lanier, J. (2010) You are not a gadget. Penguin Books.

[17] Papert, S. (1980). Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc.

[18] Pike, Rob (2012). Go At Google, Available from http://talks.golang.org/2012/splash.slide

[19] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009). Scratch: programming for all. Communications of the ACM, 52(11), 60-67.

[20] Turing, A.M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem, Proc Lond Math Soc (2), 42, 230-265

[21] Victor, Bret. "Inventing on principle." Invited Talk at Canadian University Software Engineering Conference (CUSEC). 2012.

[22] Victor, Bret. "Learnable Programming." Available at http://worrydream.com/LearnableProgramming/

[23] Wing, Jeannette M. (2006). "Computational thinking." Comm. of the ACM 49.3: 33-35.

[24] W3Schools Online Web Tutorials. Available at http://www.w3schools.com

[25] The Empirical Modelling website. Available at http://www.dcs.warwick.ac.uk/modelling

[26] The JS-EDEN interpreter. Available at jseden.dcs.warwick.ac.uk/master

Doctoral Consortium

Given the strong ethos of student support/guidance provided by PPIG through the years, it is indeed appropriate that the group's 25th Anniversary gathering contains a Doctoral Consortium. This Consortium will allows students to present their work in a non-threatening environment where expert reviewers gently probe, provide guidance, and generate constructive critique that directs Ph.D. candidates as they progress their research.

This year, the expert panel includes Thomas Green, Clayton Lewis, Marian Petre and Michael English: world class reviewers in the areas of Psychology of Programming and Software Engineering. Each of these has volunteered for this role, their involvement a testimony to the effort that PPIG puts into guiding and developing its student community.

The papers this year focus on the pedagogy of programming, with papers assessing theoretical frameworks' roles in learning to program/improving programming, and papers assessing different pedagogical practices, such as video-recorded self-explanations and the utilization of novice programmer environment like Sonic Pi. There is also a paper on assessing how the move to collaborative spreadsheet generation (in the cloud) affects the quality of the resultant spreadsheets.

We look forward to hearing about these theories and approaches in more depth on the day followed by lively, but supportive debate.

Dr. Jim Buckley

Google Sheets v Microsoft Excel: A Comparison of the Behaviour and Performance of Spreadsheet Users

Karl Mernagh

School of Business and Humanities
Dundalk Institute of Technology
karl.mernagh@dkit.ie

Dr. Kevin Mc Daid

School of Informatics
Dundalk Institute of Technology
kevin.mcdaid@dkit.ie

Keywords: POP-I.C. end user applications, POP-I.C. web, POP-III.B. spreadsheets, POP-III.B. new language, POP-IV.B user interfaces

Abstract: Summary of Findings

Spreadsheet technology has traditionally been limited to a single user operating in a desktop environment and working in an isolated environment. With the advent of Cloud Computing, a paradigm shift has occurred in the way users utilise the collaborative sharing and communication of their work in both an educational and business environment. The opportunities for people to cooperate on multiple spreadsheets at the same time and in real time have grown significantly. However, the behaviour and performance of users working in this new paradigm has not been explored to a great extent in scientific. In comparison to desktop spreadsheet technologies such as Excel, Cloud based spreadsheet technologies have only started to be investigated in relation to user behaviour and performance.

The purpose of this paper is to examine user interaction with Cloud based spreadsheets compared to desktop based spreadsheets. Specifically, we have focused on the leading technologies in these fields, namely Microsoft Excel and Google Sheets.

This paper investigates the effectiveness and efficiency of user interaction with traditional Excel spreadsheets and Google Sheets using a group of third level business students that have been trained in both applications. The students were randomly assigned to a technology and then asked to complete a list of standard spreadsheet tasks. In both cases, user interaction with the spreadsheets was tracked and recorded on a cell by cell basis using automated technology.

The findings of this paper are that users completed a set of basic tasks assigned to them in a quicker and less error prone manner when working with Cloud based Google Sheets compared to desktop based Excel spreadsheets.

The Importance of Cloud based Spreadsheets

In a paper by Brian Sommer, 2014, "So long, spreadsheets -- hello in-memory cloud financial tools", he identified the following advantages of cloud computing:

Controls over data exist for only a portion of the reporting and data capture systems. Spreadsheet systems are often absent effective dating, restart/recovery technology.

Over time, Financial/Accounting groups have compounded the problem by increasing the number of interrelated spreadsheets. As stated by Brian Sommers, *"They've added baling twine, bubble gum and bandages to the imperfect solution and have created a monster."*

The problem has gotten even more troubling as new kinds of data are being utilized by businesses and merging these data types into legacy solutions and patchwork coding will only increase the problem.

Cloud based data is particularly safe as the data is stored across multiple servers, but in the off chance that this data is lost there is no way to retrieve it unless a local backup copy is maintained by the end user.

The issue of security and confidentiality is a factor when storing information in the Cloud. With hackers now focusing more of these large datasets it could be very easy to have a data breach that causes large quantities of data to be stolen or corrupted.

Outline of Experiment

This initial experiment was designed to examine the efficiency and effectiveness of spreadsheet usage for the tasks of text entry, data entry and formula entry for both users of Microsoft Excel and Google spreadsheets in a similar format to Olsin & Nilsen, 1987. The objectives of the experiment are specified as follows.

a) Objectives: Investigate efficiency and effectiveness of spreadsheet use for

i. Text Entry.

The data examines the time to enter text and the correctness of the entered data.

ii. Data Entry.

The data examines the time to enter numeric data and the correctness of the entered data.

iii. Formula Implementation.

The data examines the time to implement formulas and the correctness of the formulas such as SUM, COUNT, AVERAGE, MAX, MIN and COUNT

iv. Time Taken To complete Experimental Task

In addition to the elements above, the experiment examined the time taken to complete the entire experimental task from start to finish. The data examines how long it took the participants to complete the experiment from start to finish.

b) Experimental Structure

The experiment consisted of two groups of third level final year business studies students that were given a set of procedures to carry out. The groups were split randomly into those that would complete the set of tasks using Excel and those that would complete the tasks using Google spreadsheets. Both sets of tasks were identical. The students were both provided with a sample file with data that they had to insert into certain cells and use formulas to calculate formulas. The students were given 5 minutes to acclimatise themselves to the exam format and to log into Google spreadsheets using their Gmail accounts while the other group copied an Excel file from a Shared Directory. Once the files were opened the recording of the exam start time began in both cases.

Recording Technology

The recording technology for the Excel spreadsheets used VBA code to track which cells the user selected, which cells they entered data, text and formulas into and when they altered these values. The results were stored in a hidden spreadsheet. It recorded the duration the user stayed within a cell and what they did while in that cell. This technology has been used by a number of other researchers including Bishop (2003) and McKeever (2011).

The recording technology for the Google spreadsheets was limited to a small extent due to the fact that Google spreadsheets will only trigger an event when the user exits a cell. This means that we could track what the user has done inside the cell and when the user left the cell. Google spreadsheets are incapable of tracking when a user enters a cell or how long they stayed within the cell. If the user returns to the cell and makes changes we could record this information with the exception that if the result of their alteration resulted in the same value being produced. This would not trigger a recording action. All of this is because Google wants to reduce the load on the server which is hosting the spreadsheet. So only the OnEdit exit event gets fired for cell changes.

b) Results

A highly noticeable behaviour that was observed with the Excel user group was that a large portion of the group entered their data in a horizontal manner. This means that they filled out the first row of data in its entirety before moving to the next row of data to be inserted. This could explain the difference in performance in text, numerical and formula entry. This did not occur with the Google group. Overall the majority of users entered their data in a vertical manner. This is discussed further in the discussion section.

i. Text Entry

The measured times in seconds to complete the text entry task for the participants using Microsoft Excel and Google spreadsheet technologies is presented in the box plot below. In total 13 participants completed the task using Microsoft Excel and 14 completed the task using Google Spreadsheets. In completing the task all but two of those using Microsoft Excel completed this element of the task completely with all but one doing so using Google Spreadsheets.

While the boxplot shows the completion times for Excel participants to be higher than those for Google, the boxplot shows a single extreme value of 322 seconds in the completion times for Google Spreadsheets. Removal of this value yields a distribution of completion times for Google participants that follow a normal distribution.

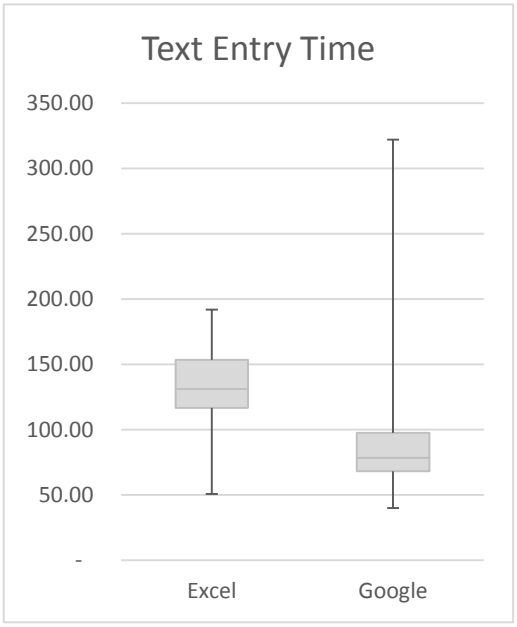
In formally comparing the performance of the two groups using an independent sample approach the one tailed p-value is 0.07. On removal of the extreme Google value the p-value drops to 0.0004, thus providing significant evidence at the 5% level that participants using Google were quicker to complete the task then those using Excel.

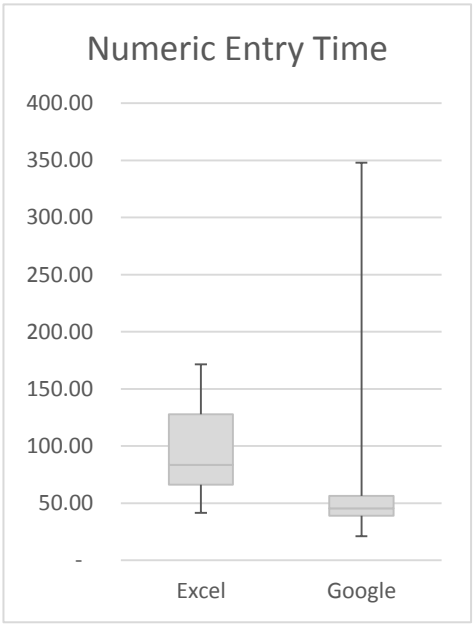
ii. Numeric Entry

The measured times in seconds to complete the numeric entry task for the participants using Microsoft Excel and Google spreadsheet technologies is presented in the box plot below. As before, the total number of participants is the same. In completing the task all but two of those using Microsoft Excel completed this element of the task completely with all but one doing so using Google Spreadsheets.

While the boxplot shows the completion times for Excel participants to be higher than those for Google, the boxplot shows a single extreme value of 348 seconds in the completion times for Google Spreadsheets. Removal of this value yields a distribution of completion times for Google participants that follow a normal distribution.

In formally comparing the performance of the two groups using an independent sample approach the one tailed p-value is 0.15. On removal of the extreme Google value the p-value drops to 0.0001, thus providing significant evidence at the 5% level that participants using Google were quicker to complete the task then those using Excel.





iii. Formula Implementation

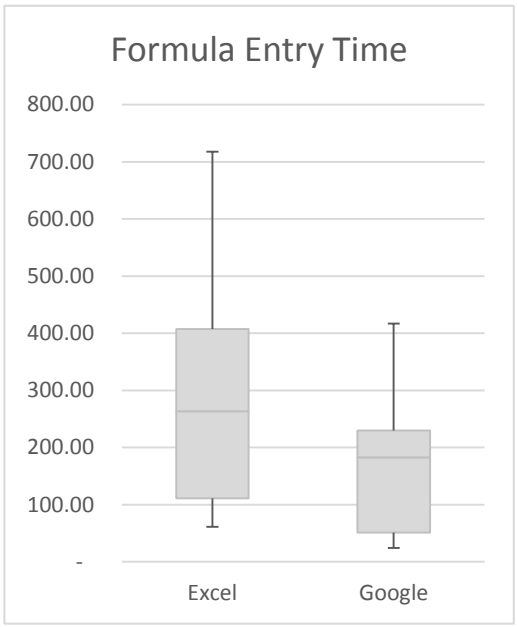
The measured times in seconds to complete the formula entry task for the participants using Microsoft Excel and Google spreadsheet technologies is presented in the box plot below. As before, the total number of participants is the same. In completing the task all but two of those using Microsoft Excel completed this element of the task completely with all but one doing so using Google The boxplot shows the completion times for Excel participants to be lower than those for Google,

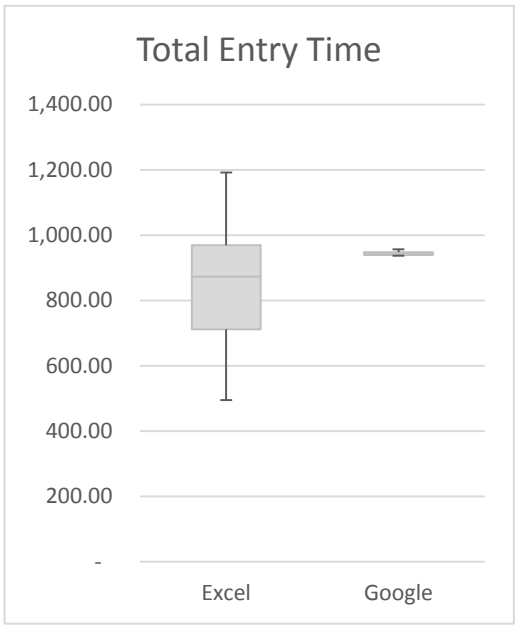
In formally comparing the performance of the two groups using an independent sample approach the one tailed p-value is 0.03.

iv. Time Taken To Completion

The measured times in seconds to complete the experiment for the participants using Microsoft Excel and Google spreadsheet technologies is presented in the box plot below. In total 13 participants completed the task using Microsoft Excel and 14 completed the task using Google Spreadsheets. In completing the task all but two of those using Microsoft Excel completed this element of the task completely with all but one doing so suing Google Spreadsheets. Note that the average score out of 20 for Excel users was 19.8 with the average score for Google users of 19.9.

In formally comparing the performance of the two groups using an independent sample approach the one tailed p-value is 0.38.





Discussion

Less user interaction was recorded for Google spreadsheets due to the ability of Google spreadsheets to record every interaction without the use of third part libraries such as the GData library. This was expected due to the limitations currently surrounding recording of interactions with Cloud technology. Cloud technology only records user interaction after a cell has been exited and there isn't a native way to get around this without a third party library. Excel used VBA code to record user interactions which allowed for a much more detailed set of results to be recorded.

The experiments provided some evidence to support the conclusion that participants using Google sheets outperformed those using Excel in so far as they took significantly less time to complete the tasks. Also, in completing the tasks they scored the same on the basic tasks of text and numeric entry but significantly higher for the task of formula implementation. These results are somewhat surprising for two reasons. Firstly, the participants were probably more familiar with Excel than Google technology. Secondly, distributed spreadsheets such as Google sheets are browser based and with this there is a slight delay in the execution of tasks such as entry of data and formulae to cells which might have been expected to adversely affect the performance of the Google sheets group.

There are a number of issues that need to be discussed relating to the generalizability of these results.

Firstly, the participants were students and while their background was in business their knowledge of spreadsheets is most likely not at the same level of real daily spreadsheets users.

Secondly, the task selected was short and, while it included the key elements of text, numeric and formula entry, a longer more involved task may better reflect the real development tasks performed by spreadsheet users.

Thirdly, the Excel user group behaviour of entering data in a horizontal manner had a significant time to completion factor for each set of tasks. The Google user group filled out the spreadsheet in a vertical manner. They worked from column to column, inserting textual data, numerical data and using the fill down functionality for formulaic entry.

We can only speculate that the Excel users had difficulty with absolute cell references as it appears in the recorded data that they made multiple attempts to fill down once they had a formula correct but had not used the appropriate absolute referencing to get the correct answer. The Google groups' data shows that they had no difficulty with cell referencing and that very few errors were made when filling down formulaic entries.

Bearing these points in mind, and considering that this is the first experiment that seeks to compare the performance of users of these technologies, the authors feel that this research provides some support for the argument that using Google Sheets is as least as effective as using Excel to develop spreadsheets but that further work is needed to provide stronger evidence.

Conclusion

In conclusion, this initial experiment of comparing user behaviour and performance using a Cloud based spreadsheet technology versus traditional Excel technology has produced a surprising set of results.

Users that are more familiar with Excel seem to have behaved in a less efficient manner. Users that had limited experience and exposure to Google spreadsheets outperformed their Excel rivals in all tests recorded. As this is a small sample group using only two technologies, it is worthy of further investigation in both a larger user base and a more experienced user base.

Also worth noting is that the spreadsheet error checking technologies are in the main not compatible with Cloud based spreadsheets which leaves both market gap and more significantly for industry an absence of error checking. This is important as Microsoft Excel 365 has moved to a Cloud based environment.

As applications such as Office 365, which are pushing users into the Cloud, emerge and are adopted more widely by industry it would be interesting to see if Excel in the Cloud can match Google Spreadsheets or one of the other Cloud based spreadsheet applications that are emerging at present.

Another point that would need to be studied further is the programmability of these applications in a Cloud based environment. Without the use of the GDATA API, Google spreadsheet programming abilities are severely hampered. This issue may become less relevant as Google expands the capabilities of its programming environment with new iterations of their technology. Office 365 has a vast amount of programmability using technologies ranging from the REST API to HTML5 and XML to implement VBA but will they suffer the same deficiencies as Google Spreadsheet programming.

References

[1] Sommer, B. (2014). So long, spreadsheets -- hello in-memory cloud financial tools http://www.zdnet.com/so-long-spreadsheets-and-hello-in-memory-cloud-financial-tools-7000026176/

[2] Leonard, J. (2013) Office 365 and Google Apps fight for supremacy http://www.computing.co.uk/ctg/analysis/2269524/office-365-and-google-apps-fight-for-supremacy

[3] Rust, A., Mc Daid, K., Bishop, B. (2008) Investigating the Potential of Test-Driven Development for Spreadsheet Engineering http://arxiv.org/ftp/arxiv/papers/0801/0801.4802.pdf

[4] McKeever, R., (2011) How do Range Names Hinder Novice Spreadsheet Debugging Performance? http://arxiv.org/ftp/arxiv/papers/1009/1009.2765.pdf

[5] McKeever, R., (2011) Effect of Range Naming Conventions on Reliability and Development Time for Simple Spreadsheet Formulas http://arxiv.org/ftp/arxiv/papers/1111/1111.6872.pdf

[6] Olsin, JR & Nilsen, E., (1987) Analysis of the Cognition involved in Spreadsheet Software Interaction, Human-Computer Interaction, Volume 3 Issue 4, December 1987, Pages 309-349

Computational Linguistics Vice Versa

Sebastian Lohmeier

M.Sc. Informatik degree course
Technische Universität Berlin
sl@monochromata.de

Keywords: POP-II.B. program comprehension, POP-III.B. new language, POP-V.A. linguistics and mental models, POP-V.B. eye movements

Abstract

Computational linguists use tools developed by computer scientists to analyse language in ways not practically possible in a manual way. Can this exchange be reversed? How could computer scientists use theories developed by linguists to improve programming? To describe my work, I compiled a number of questions concerning a cognitive linguistics of object-oriented programming.

1. What do I want to know?

I want to know how the structure of source code influences how programmers comprehend it and whether ways of structuring text can inspire the structure of source code in a way that makes source code shorter but not harder to comprehend or re-use.

2. What is programming, then?

Based on my research interest, I regard programming as an activity performed by humans. A theory of programming therefore needs to be psychological. When engaged in the activity of programming, programmers manipulate structures that bear certain resemblances to natural languages which is why they are called programming languages. As a consequence, a theory of programming needs to be psycholinguistic. Linguistic aspects studied in psycholinguistics, especially when related to meaning, target many issues also investigated in philosophy, neuroscience, artificial intelligence and can be subsumed under one of the branches of cognitive linguistics. A theory of programming thus needs to be based on a form of cognitive linguistics. I limit my work to object-oriented general-purpose programming languages. I prefer object-oriented languages for their similarities with schemata used in cognitive linguistics and to limit the complexity of my work. I target a mainstream general-purpose language in contrast to e.g. end-user programming (Pane & Myers, 2006) or interactive fiction (Nelson, 2006) because the programming languages that I use myself are general-purpose programming languages. That does not mean that the empirical basis of end-user programming languages and Inform 7 is less relevant for my work.

Three fundamental aspects of a cognitive linguistics of object-oriented programming will be treated in the following: (1) the cognitive gap between compiler and programmer, (2) the information systems metaphor of the mind that influenced cognitive science and cognitive linguistics and (3) that the compiler enforces its "meaning" of a program.

1. Aspects of the cognitive gap between compiler and programmer have been touched upon a number of times already (Green, 1980; Détienne, 2001, 13f.; Knöll, Gasiunas, & Mezini, 2011, 34) and should be subject to a comprehensive analysis. Here, I will confine my analysis to identifiers used to name variables, classes and methods, however.
   To the compiler, an identifier is a sequence of characters that is declared to refer to either a variable, a type or a method. The choice of characters in the sequence does not affect the operation of the compiler as long as it is distinct from other identifiers subject to certain constraints. From the perspective of a compiler, an identifier fits a definition of proper names occurring in natural language: it refers to

a unique entity that is highlighted within a class of entities by being given a name and the meaning of the name does not (anymore) determine what the name refers to (van Langendonck, 2007, 87ff.). To a programmer, on the other hand, many identifiers are meaningful, i.e. function like common nouns and verbs, because nouns, verbs, or noun phrases like *menu*, *close*, and *EditableDataSource* are often chosen as identifiers.

Because programmers comprehend the meaning of identifiers that is not accessible to a compiler, the structures built by the compiler may not correspond to the mental representations constructed by the programmer. An identifier that functions as a common name to the programmer functions as a proper name to the compiler. Is that a problem? One can imagine situations in which a name of a class is changed such that the names of variables used to store its instances do not correctly hint at the type anymore. In functional programming this problem does not occur because the type of a variable is not declared.

Would it be possible to do the opposite, i.e. to omit the identifier and keep the type name? If there is no declaration that relates an identifier to a type and an initial value, what should be used to refer, then? In texts, unlike source code, proper names are not the only means of reference. Noun phrases are often used to refer, too. Both proper names and noun phrases are used to refer to so-called referents – entities in a mental representation constructed by the reader of the text. A second reference to a referent will make the second referring expression function as anaphor: an expression whose meaning is related to the meaning of another related expression in the prior text. Both proper and common names can function as anaphors, i.e. a proper name can be used to refer to what has been referred to by that proper name before and a common name can be used to refer to what has been referred by using that common name before. This works in object-oriented source code, too: `JMenuItem item = new JMenuItem("Hello"); item.setEnabled(false);` works just as well as `new JMenuItem("Hello"); .JMenuItem.setEnabled(false);` In the latter case no identifier holding a proper name needs to be assigned. Instead, the recurrence of the type name in the constructor invocation and the anaphor (prefixed with a dot) relates the constructor invocation and the anaphor to each other and allows readers and programmers to understand that they refer to the same referent, an instance of `JMenuItem`. I put aside the question of how often reference via type names is applicable. Work on definite descriptions in source code (Knöll et al., 2011) could overcome this issue besides keeping local variables for cases in which an actual proper name is required.

2. While the cognitive gap is relatively wide in the case of the comprehension of identifiers, it is narrow when it comes to the representation of concepts in source code and in cognitive linguistics. Structures and processes during language comprehension are described in a branch of cognitive linguistics (e.g. van Dijk & Kintsch, 1983; Kintsch, 1988; Kintsch, 1998; Schwarz, 1992; Schwarz-Friesel & Consten, 2011) and as part of some psycholinguistic research when aiming to explain experimental results (e.g. Garrod & Terras, 2000). Both forms of research are well suited to have their results transferred to object-oriented programming languages, because these languages use a knowledge representation close to conceptual knowledge and the description of psychological processes operating on conceptual knowledge structures can be compared to how compilers use knowledge encoded in source code. I.e. while the information-processing metaphor of the human mind may at least require supplementation with sub-symbolic activation processes to explain psycholinguistic evidence, the information-processing metaphor of the mind provides a suitable basis for applying cognitive linguistics to object-oriented programming languages.

   Because knowledge structures used in cognitive linguistics and in object-oriented programming share a number of features, not only those anaphors can be implemented in programming languages that are based on recurrence (i.e. identical repetition) of proper or common names. Because sub- and super-class relationships are declared in the source code and are therefore accessible to the compiler, a referent can also be referred to using its supertype. Likewise, the relations between classes that are encoded in return types of methods, field declarations, and declarations of accessor methods can be used to implement so-called associative or indirect anaphors (Garrod

& Terras, 2000; Schwarz-Friesel, 2007) in the compiler (Lohmeier, 2011). Instead of writing `void addOne(ServiceRegistrar registrar) { new RegistrarMenuItem(host, registrar.getServiceID()); }` one could use the indirect anaphor `.ServiceID` to access the ServiceID that is a part of the instance stored in the parameter `registrar` in `void addOne(ServiceRegistrar registrar) { new RegistrarMenuItem(host, .ServiceID); }`. While these forms of anaphors are only the most simple of the more complex forms that are possible, they show that complex forms of reference from natural language can be implemented in programming languages.

These indirect forms of reference also highlight a potential possible effect of reference modelled after linguistic theory: to shorten source code without making it more abstract but at the same time hiding irrelevant information. What information is irrelevant is of course dependent on programmer-related factors, as will be discussed later.

3. It also helps to note that the compiler enforces its "meaning" of a program when a programmer and a compiler create different representations, one reading and the other processing the program. When a compiler is written in a way to permit anaphors in source code that resemble anaphors known from natural language, it may process these anaphors in ways that have been found to adequately model human comprehension of anaphors. If the compiler fails to process these anaphors, it may be that humans would have a hard time or be unable to understand the anaphor written by the programmer. Thus, the compiler may be used to detect and reject undesired use of anaphors, e.g. the anaphor `.JMenuItem` that is referentially ambiguous in the presence of two variables that can hold instances of that type. The compiler therefore needs to be able to limit the input to valid code, like is typically done for programming languages. This is different from natural language, where typically there is no single communication participant who is able to enforce her interpretation of an utterance. As a result, there is no risk of introducing ambiguity to a programming language by adding anaphors, if the compiler detects and rejects referentially ambiguous anaphors.

What is programming, then? Programming is a human activity that can be modelled at a formal and at a psychological level. During programming, programmers read and write source code that is also processed by a compiler. In a psychological model of code reading, information from long-term memory is activated and integrated into a representation of the code read. The distinct parts of the representation have activation levels than increase when they are brought to attention or are read. The activation levels decrease with time. Activation levels also influence how long a word is gazed at. The gaze durations, activation levels, memory access and construction of the mental representation of the text can be modelled with a cognitive model. A formal model in the compiler will be restricted to integrating knowledge from the source code into a representation of the source code and detecting and rejecting referential ambiguity. Even though I consider programming a human activity, I do not wish to study decision making during programming.

3. Will I need my linguist friends?

Although every student of computer science hears of Chomsky and generative grammar, nothing at all is taught about contemporary linguistics in courses on programming languages and compiler construction. The only words on natural language that one may expect in such a course may be on syntactic ambiguity of natural language. There is research on programming languages that incorporates or at least touches upon further ideas from linguistics, though. How is this research related to linguistic theory? Lopes, Dourish, Lorenz, and Lieberherr (2003) list a number of works from linguistics. I am not able to judge how useful these works are for implementing anaphors in programming, but they provide a starting point into some linguistic research. Knöll and Mezini (2006) were obviously inspired by cognitive theories of the mind, but do not cite any such theories and use seemingly custom wording like "ideas" instead of concepts, as well as "implicit" and "explicit" reference that could also be replaced by standard terminology from psychology and linguistics. Knöll et al. (2011) describe the counterparts of complex noun phrases in programming but do not refer to any literature from (psycho-) linguistics at all, that

could have provided them with the explanation that 1-character identifiers are harder to understand than identifiers with known nouns because they ease association-based memory access. Instead, they suppose that name assignment is an indirection that occurs in the mind of the programmer, no only in the code. Détienne (2001, 19) states that anaphors are a case of "indirect reference" and that only pronouns like *he* and *it* and pronominal adjectives like *this* function as anaphors, which is both wrong, as can not only be seen from psycholinguistic studies like Garrod, Freudenthal, and Boyle (1994). Based on her description of anaphors, Détienne concludes that anaphors are rare in source code and procedural text, while it is actually pronouns and pronominal adjectives that are rare in both kinds of text. In summary, in the few works where it is due, reference to linguistic theory is at times missing or incorrect. These works can hardly help reduce negative preconceptions of natural language and linguistics while it would be necessary to invite curiosity for and provide access to linguistic theory. I am therefore thankful to my linguistic friends who keep me in touch with contemporary linguistics.

4. What about naturalistic programming?

Lopes et al. (2003, 199) proposed the concept of naturalistic programming and stated that "the primitive abstractions in programming languages should be drawn from the study of Natural Languages, rather than from Computer Engineering or Mathematics or ad-hoc metaphors such as Objects." Lopes et al. also claim that naturalistic programming is not for "'natural language programming,' an idea that has been around for some decades and that has been instantiated occasionally [...]. We don't advocate implementing English! The languages we are proposing are naturalistic, but not natural. (Lopes et al., 2003, 204)" Still, their paper uses code to illustrate a naturalistic programming language that "reads like English (Lopes et al., 2003, 202)". In another deviation from real-world programming practice, Knöll et al. (2011) use the description of a house and a garden to exemplify their "naturalistic types". I do instead start from a commercially used programming language, so that even the addition of features from natural language will not make the new language look like English. I am also interested in evaluating the new language features using the abstract technical concepts found in real-world source code and in paying attention to problems peculiar to complex evolving software systems, e.g. the fragile base class problem (Mikhajlov & Sekerinski, 1997). Because computer programming is a very artificial activity compared to gardening, e.g., I would also like to avoid associating nature and programming.

There is another label, "programming linguistics" under which an early work comparable to what I intend to do was written (Kanada, 1981) – in Japanese, unfortunately. There has also been a book under that label, but its use of the word "linguistics" is metaphorical only (Gelernter & Jagannathan, 1990). I used another work from Japan that studies programming from a semiotic perspective (Tanaka-Ishii, 2010) as the blueprint for the label "linguistics of programming".

5. What to do?

I use linguistic theory to construct *and* implement a narrowly-defined new language feature (indirect anaphors) for an existing mainstream programming language. That way I avoid the problem described by Pane and Myers (2006, 36): that results from the psychology of programming are not reflected in new programming languages.

Indirect anaphors in source code could make programming more efficient by shortening the source code, but will not be understood equally well by all programmers, for it depends on domain knowledge. Based on a previous implementation of a compiler for indirect anaphors in Java (Lohmeier, 2011) and prior work on cognitive models of text comprehension (Lohmeier & Russwinkel, 2013), I prepared an eye-tracking experiment in which programmers read indirect anaphors in source code (see the paper in the proceedings of this workshop). The gaze durations on and after the indirect anaphors are treated as indicators of processing difficulty that will be manipulated by how often and recently a programmer read the information required to comprehend the indirect anaphor. I expect that indirect anaphors whose comprehension incorporates less familiar domain knowledge are harder to understand and will be gazed

at for a longer duration. The results of the experiment will be compared to predictions generated using a cognitive model.

I assume that eye tracking is a suitable method for studying the comprehension of indirect anaphors in source code, because it has been used to study indirect anaphors in linguistics (Garrod & Terras, 2000) and because it allows on-line data collection without interrupting the main task. It may be possible to extend the experimental setting after repetitions of the experiment with further independent variables: E.g. a Java source code editor may be created to support efficient source code reading and writing using indirect anaphors. The editor will display indirect anaphors instead of corresponding normal Java expressions only if the programmer is able to comprehend the indirect anaphors. Comprehension will be predicted using a cognitive model that reads the eye tracking record of the code that the programmer read and wrote previously.

Using the example of indirect anaphors, I am going to try out whether linguistics can advance the development of programming languages, comparable to the interdisciplinary relationship in computational linguistics, but vice versa.

6. References

Détienne, F. (2001). *Software design – cognitive aspects*. London: Springer.

Garrod, S., Freudenthal, D., & Boyle, E. (1994). The role of different types of anaphor in the on-line resolution of sentences in a discourse. *Journal of Memory and Language*, *33*(1), 39–68.

Garrod, S., & Terras, M. (2000). The contribution of lexical and situational knowledge to resolving discourse roles: Bonding and resolution. *Journal of Memory and Language*, *42*, 526–544.

Gelernter, D., & Jagannathan, S. (1990). *Programming linguistics*. Cambridge, MA: MIT Press.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith & T. R. G. Green (Eds.), *Human interaction with computers* (pp. 271–320). London: Academic Press.

Kanada, Y. (1981). *Toward programming linguistics*. Master's thesis (in Japanese with english abstract and table of contents), University of Tokyo Graduate School. Retrieved 2014/05/13, from http://www.kanadas.com/papers-e/1981/03/toward_programming_linguistics.html

Kintsch, W. (1988). The role of knowledge in discourse comprehension: A construction integration model. *Psychological Review*, *92*(5).

Kintsch, W. (1998). *Comprehension: a paradigm for cognition*. Cambridge: Cambridge University Press.

Knöll, R., Gasiunas, V., & Mezini, M. (2011). Naturalistic types. In *Onward '11: Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (pp. 33–47).

Knöll, R., & Mezini, M. (2006). Pegasus: first steps toward a naturalistic programming language. In *Companion to the 21st ACM SIGPLAN symposium on object-oriented programming, systems, languages & applications (OOPSLA '06)* (pp. 542–559). New York: ACM.

Lohmeier, S. (2011). *Continuing to shape statically-resolved indirect anaphora for naturalistic programming.* Retrieved 2014/06/01, from http://monochromata.de/shapingIA/

Lohmeier, S., & Russwinkel, N. (2013). Issues in implementing three-level semantics with ACT-R. In *Proceedings of the 12th international conference on cognitive modeling (ICCM).*

Lopes, C. V., Dourish, P., Lorenz, D. H., & Lieberherr, K. (2003). Beyond AOP: toward naturalistic programming. *SIGPLAN Notices*, *38*(12), 34–43.

Mikhajlov, L., & Sekerinski, E. (1997). *The fragile base class problem and its solution* (Tech. Rep.).

Nelson, G. (2006). *Natural language, semantic analysis, and interactive fiction.* Retrieved 2014/06/10, from http://www.inform7.com/learn/documents/WhitePaper.pdf

Pane, J. F., & Myers, B. A. (2006). More natural programming languages and environments. In H. Lieberman, F. Paterno, & V. Wulf (Eds.), *End-user development* (pp. 31–50). Dordrecht: Springer.

Schwarz, M. (1992). *Kognitive Semantiktheorie und neuropsychologische Realität: repräsentationale und prozedurale Aspekte der semantischen Kompetenz*. Tübingen: Niemeyer.

Schwarz-Friesel, M. (2007). Indirect anaphora in text: A cognitive account. In M. Schwarz-Friesel, M. Consten, & M. Knees (Eds.), *Anaphors in text : cognitive, formal and applied approaches to anaphoric reference* (pp. 3–20). Amsterdam: Benjamins.

Schwarz-Friesel, M., & Consten, M. (2011). Reference and anaphora. In *Foundations of pragmatics* (Vol. 1, pp. 347–372). Berlin: De Gruyter.

Tanaka-Ishii, K. (2010). *Semiotics of programming*. Cambridge: Cambridge University Press.

van Dijk, T. A., & Kintsch, W. (1983). *Strategies of discourse comprehension*. New York: Academic Press.

van Langendonck, W. (2007). *Theory and typology of proper names*. Berlin: Mouton de Gruyter.

Applying Educational Data Mining to the Study of the Novice Programmer, within a Neo-Piagetian Theoretical Perspective

Alireza Ahadi

University of Technology, Sydney
Australia
Alireza.Ahadi @student.uts.edu.au

Keywords: *POP-I.B. barriers to programming, POP-II.A. novices, POP-V.A. Neo-Piagetian, POP-VI.E. computer science education research*

1. Introduction

It is well known that many first year undergraduate university students struggle with learning to program. Educational Data Mining (EDM) applies data mining, machine learning and statistics to information generated from educational settings. In this PhD project, I will apply EDM to study first-semester novice programmers, using data collected from students as they work on computers to complete their normal weekly laboratory exercises.

This PhD project is within its first six months.

2. Background

My doctoral project is informed by past research from two broad sources, described below.

2.1 Educational Data Mining and the Novice Programmer

Over the last ten years, a number of systems have been developed systems for routinely logging data about computing students (Winters & Payne, 2005; Jadud, 2006; Norris et al., 2008; Edwards et al., 2009; Edwards, 2013). Most of these systems are used primarily to grade student assignments, both summatively and formatively. Consequently, much of the data collected by these systems relates to frequency of submission and number of test cases passed. Other systems have been developed to collect richer data (Romero et al., 2004 & 2005; Brown et al., 2014).

2.2. Neo-Piagetian Theory

Neo-Piagetian provides the theoretical context for this project. Lister (2011) proposed a three stage model of the early stages of learning to program, which are (from least mature to most mature) the sensorimotor stage, the preoperational stage and the concrete operational stage. For the benefit of readers unfamiliar with neo-Piagetian theory, we refer to these three stages by different names from Lister (2011), and define each as follows:

1) **Pre-tracing Stage (i.e. sensorimotor):** The best known problems faced by novices at this level are misconceptions about the semantics of programming language constructs. For example, a novice might believe that assignment statements assign values from left to right. There also exist other problems, which result in the novice being unable to accurately trace (i.e. manually execute) small pieces of code (du Boulay, 1989). When attempting to write code, novices at this stage may struggle to produce code that compiles successfully. When they do produce a "clean compile" the code may bear little resemblance to the problem to be solved, and the code may even contain aspects that are bizarre to an experienced programmer.
2) **Tracing stage (i.e. preoperational):** The novice can reliably manually execute ("trace") multiple lines of code. Furthermore, novices at this stage rely almost exclusively on their tracing skill when reasoning about programs. When reading code, these novices often make inductive guesses about what the code does, by performing one or more traces, and examining the relationship between the input and resultant output. When writing code, these novices tend to patch and repatch their

code, on the basis of either (1) their results from tracing through their code specific initial variable values, that may be chosen somewhat at random chosen, or (2) after a tutor has pointed out a bug (Ginat, 2007). These novices struggle to truly design a solution.

3) **Post-tracing Stage (i.e. concrete operational):** These novice programmers reason about code deductively, by reading the code itself, rather than using the preoperational inductive approach. They can relate diagrams to code. This stage is the first stage where students begin to show a purposeful approach to writing code.

Since that first paper by Lister in 2011, he and his colleagues have collected empirical support for these neo-Piagetian stages (Corney, 2012; Teague et al., 2012a, 2012b, 2013; Teague and Lister, 2014a, 2014b, 2014c and 2014d). However, all these studies have involved students completing programming tasks with pencil and paper. An example of a task given to novices in these pencil and paper exercises is as follows:

> The purpose of the following Java code is to move all elements of the array x one place to the right, with the rightmost element being moved to the leftmost position:
>
> ```
> int temp = x[x.length-1];
>
> for (int i = x.length-2; i>=0; --i)
>    x[i+1] = x[i];
>
> x[0] = temp;
> ```
>
> Write code that undoes the effect of the above code. That is, write code to move all elements of the array x one place to the left, with the leftmost element being moved to the rightmost position.

Other examples of tasks used to identify the neo-Piagetian stage of a novice programmer can be found in this conference proceedings (Ahadi, Lister and Teague, 2014; Teague and Lister, 2014d; Teague, 2014e).

My research will be the first to look for evidence for these neo-Piagetian stages in novices as they write programs on a computer. One or both of two broad approaches will be used for detecting relationships between on-machine programming behaviour and neo-Piagetian stages:

- **Supervised Learning:** Volunteer students will first complete paper-based tests, while thinking out loud, as in Teague's research, using the existing tasks designed by Teague and Lister. These tests will be used to assign novices to a neo-Piagetian stage. Their subsequent on-machine programming behaviour will then be analysed by data mining algorithms to try to detect differences between students assigned to the different neo-Piagetian stages.
- **Unsupervised Learning:** Automatic cluster analysis will be applied to data collected from the on-machine programming behaviour of novices. Manual analysis will then attempt to link clusters to neo-Piagetian stages.

What neo-Piagetian related patterns might we find in the data, and what problems could there be in automatically detecting those patterns? With reference back to the neo-Piagetian stages described in section 2.2:

- **Pre-tracing Stage (i.e. sensorimotor):** *These novices struggle to produce code that compiles successfully. When they do produce a "clean compile" the code may bear little resemblance to the problem to be solved*. Analysis of compilation frequency and the ratio of all compiles to clean compiles may be useful here. Identifying code of little resemblance to the problem to be solved might involve static analysis, but looking at test failures on simple general cases might be just as effective.

- **Tracing stage (i.e. preoperational):** *When writing code, these novices tend to patch and repatch their code. These novices struggle to truly design a solution*; whereas ...
- **Post-tracing Stage (i.e. concrete operational):** *This stage is the first stage where students begin to show a purposeful approach to writing code*. Thus a method for the automatic detection of a "purposeful approach" is the key issue for distinguishing between tracing and post-tracing novices.

3. The Research Question

The current formulation of my research question is as follows:

> *Do novices who are reasoning at each of these three different neo-Piagetian stages go about writing programs differently, in ways automatically detectable by the computer they are using?*

The significance of this research is two-fold:

1) It will establish that neo-Piagetian stages do not just occur in "artificial" pencil and paper exercises, but also occur in the "natural" environment – that is, as novices write code on the computer.
2) If the neo-Piagetian stage of a student can be automatically identified, then a computer-based tutoring system could use that information. (However, the development of such a tutoring system is beyond the scope of my project.)

4. A Candidate for the Primary Data Collection Tool — Web-CAT

My most recent work has focussed on identifying a suitable, existing data collection tool for conducting a pilot. The current preferred data collection tool is Web-CAT, which is an established automated grading system that can be used to assess student's programs. Other researchers have already used Web-CAT to study the behaviour of novice programmers (Edwards, et al., 2009; Edwards, 2013). Web-CAT can store student software testing activities, in addition to simply judging student work by comparing program output to target output. Web-CAT is implemented as a web application with a well-defined API plug-in-style architecture so that it also can serve as a platform for providing additional student support and/or data collection. It supports static analysis tools to assess documentation and coding style and manual grading with direct on-line markup of assignments.

The students at my institution are first taught the Java programming language. Web-CAT supports the following features for processing Java programs:

- Support for static analysis of student code using Checkstyle to identify documentation, formatting, naming, and stylistic errors.
- Support for static analysis of student code using PMD to identify additional stylistic and coding errors.
- Support for optional instructor-provided reference tests that will be automatically executed against the student code.
- The ability to write new PMD checks using xpath expressions (or provide instructor's designed Java implementations of PMD or Checkstyle checks).

5. Granularity of Data

At this early stage of the PhD project, an issue under current consideration is the type of data I should collect for each student as they work on a specific programming problem. Some forms of data under consideration for collection are, from least demanding and sparse to more demanding and dense:

- **Every clean compile:** The amount of data collected would easily be manageable. However, this is the type of data that has been collected in many earlier systems and thus the novelty is

low. Also, there is a risk that this type of data may not be enough to identify the different neo-Piagetian stages.

- **Every compile:** At this time, we suspect that this project will focus on distinguishing between students at the tracing and post-tracing stages, and we suspect that information about syntax errors is mostly an issue for pre-tracing students.
- **Every keystroke and mouse click:** While this is very rich data, it will pose data management issues. Also, it is not clear to us how we would use such data, apart from its simple use to estimate whether the student is actually working on the given programming task. One possibility is to look for a "rhythm" to user key presses and mouse clicks. We tentatively suspect that post-tracing (concrete operational) novices will display a steady "rhythm", while novices at the two earlier neo-Piagetian stages will display more sporadic behaviours.
- **Periodic Facial Images:** This is of interest for two reasons, assuming suitable software for analyzing the images already exists. First, such data would help establish that the novice under study is actually looking at the computer screen, and is not distracted. (Otherwise, time on task is hard to estimate automatically.) Second, the emotional state of the novice may be salient to detecting how well a student is coping with a given programming exercise (Good et al., 2011; Rodrigo and Baker, 2009). We suspect that post-tracing (concrete operational) novices will display more deliberate and systematic approaches to code writing and debugging, and will show less negative emotion than students at the two earlier neo-Piagetian stages. There already exists expertise at our university in analyzing facial images to detect emotions (Tan, Leong and Shen, 2014).
- **Eye-tracking data:** Eye-tracking data from novice programmers has already been extensively studied (e.g. Bednarik and Tukainine, 2004; Bednarik et al., 2005 & 2006; Nevalainen and Sajaniemi, 2004), but not from within a neo-Piagetian framework. We suspect that a pre-tracing (sensorimotor) novice student may either be focussed on the wrong part of the program, or may be glancing rapidly and perhaps randomly around the whole program. As our eventual aim (beyond this PhD) is to develop a tutoring system that is deployable on off-the-shelf computers, we expect we will only collect eye-tracking data if we can do so with easily accessible hardware and software.

Capturing data at every compile or every clean compile is easily compatible with the Web-CAT system, but the other options would require more sophisticated and more custom software.

Capturing facial images may present difficulties with getting clearance from our university's ethics committee.

6. Pilot Study (July – November, 2014)

In the coming southern hemisphere semester (i.e. July – November, 2014), we will conduct a pilot study in an introductory programming subject, which has over 300 enrolled students. A major component of the subject's assessment will be in the form of regular paper- and lab-based tests, held every second week. Furthermore, some of these tests are based upon the past work of Teague and Lister, and thus already have a neo-Piagetian component. The pilot will focus upon collecting and analyzing data from these tests, using Web-CAT.

The granularity of the pilot data collected from the introductory programming subject will be at the compile level. During this pilot phase, we will also evaluate the technical feasibility of collecting keystroke, facial and eye-tracking data, but we do not plan to collect such data with the aim of performing serious analysis during this pilot.

After the pilot, we expect that the "serious" data collection for the project will occur in the second year of PhD candidature, in the two southern hemisphere semesters of 2015 (i.e. Feb – June and July – November).

7. References

Ahadi, A. and Lister, R. (2013) *Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant*? In Proceedings of the ninth annual international ACM conference on International computing education research (ICER '13). ACM, New York, NY, USA, 123-128. http://doi.acm.org/10.1145/2493394.2493416

Ahadi, A., Lister, R. and Teague, D. (2014) *Falling Behind Early and Staying Behind When Learning to Program*. Paper presented at the 25th Anniversary Psychology of Programming Annual Conference (PPIG), Brighton, England , 25th-27th June 2014.

Bednarik, R. and Tukainine, M. (2004) *Visual attention and representation switching in Java program debugging: a study using eye movement tracking.* 16th PPIG Conference. pp. 159-169

Bednarik, R., Myller, N., Sutinen, E. & Tukiainen, M. (2005) *Effects of experience on gaze behaviour during program animation*. 17th PPIG Conference. pp 49-61.

Bednarik, R., Myller, N., Erkki Sutinen, E., and Tuki, M. (2006) *Program Visualization: Comparing Eye-Tracking Patterns with Comprehension Summaries and Performance*. 18th PPIG Conference. pp. 68 – 82

Brown,N., Kolling, M., McCall, D. And Utting, I. (2014) *Blackbox: A Large Scale Repository of Novice Programmer' Activity*. In Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14). pp. 223-228. http://doi.acm.org/10.1145/2538862.2538924

Corney, M., Teague, D., **Ahadi, A.,** & Lister, R. (2012). *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Paper presented at the 14th Australasian Computing Education Conference (ACE 2012). pp. 77-86. http://www.crpit.com/confpapers/CRPITV123Corney.pdf

Crosby, M. Scholtz, J. Wiedenbeck, S. (2002) *The Roles Beacons Play in Comprehension for Novice and Expert Programmers*. 14th PPIG Conference. pp. 58-73

Du Boulay, B. (1989). *Some Difficulties of Learning to Program*. In E. Soloway & J. C. Sphorer (Eds.), Studying the Novice Programmer (pp. 283-300). Hillsdale, NJ: Lawrence Erlbaum.

Edwards, S., Snyder, J., Pérez-Quiñones, M., Allevato, A., Kim, D. and Tretola, B. (2009) *Comparing effective and ineffective behaviors of student programmers*.In Proceedings of the fifth international workshop on Computing education research workshop (ICER '09). pp. 3-14. http://doi.acm.org/10.1145/1584322.1584325

Edwards, S. (2013) Continuous Data-driven Learning Assessment. In Future Directions in Computing Education Summit White Papers (SC1186). Dept. of Special Collections and University Archives, Stanford University Libraries, Stanford, Calif. http://www.stanford.edu/~coopers/2013Summit/EdwardsStephenVaTech.pdf

Ginat, D. (2007). Hasty design, futile patching and the elaboration of rigor. *SIGCSE Bull.* 39, 3 (June), 161-165. http://doi.acm.org/10.1145/1269900.1268832

Good, J., Rimmer, J. Harris, E. and Balaam, M. (2011) *Self-Reporting Emotional Experiences in Computing Lab Sessions: An Emotional Regulation Perspective*. 23rd PPIG Conference.

Jadud, M. (2006) *An Exploration of Novice Compilation Behaviour in BlueJ.* PhD thesis, University of Kent. http://kar.kent.ac.uk/14615/

Khan, I. A., Brinkman, W-P. and Hierons, R. (2008) *Towards a Computer Interaction-Based Mood Measurement Instrument.* 20th PPIG Conference.

Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011). pp. 9-18. http://crpit.com/confpapers/CRPITV114Lister.pdf

Nevalainen, S. and Sajaniemi, J. (2004) *Comparison of Three Eye Tracking Devices in Psychology of Programming Research* .16th PPIG Conference. pp. 151-158

Norris, C., Barry, F., Fenwick Jr., J. B., Reid, K., and Rountree, J. (2008). *ClockIt: collecting quantitative data on how beginning software developers really work*. In *Proceedings of the 13th*

annual conference on Innovation and technology in computer science education (ITiCSE '08). pp. 37-41. http://doi.acm.org/10.1145/1384271.1384284

Rodrigo, M. and Baker, R. (2009) *Coarse-grained detection of student frustration in an introductory programming course*. In Proceedings of the fifth international workshop on Computing education research workshop (ICER '09). pp. 75-80. http://doi.acm.org/10.1145/1584322.1584332

Romero, P., du Boulay, B., Cox, R., Lutz, R. and Bryant,S. (2004) *Dynamic rich-data capture and analysis of debugging processes*. 16th PPIG Conference. pp. 140-150

Romero, P, du Boulay, B., Cox, R., Lutz, R & Bryant, S. (2005) Graphical visualisations and debugging: a detailed process analysis. 17th PPIG Conference. pp. 62 - 76

Tan, C., Leong, T. and Shen, S. (2014) *Combining think-aloud and physiological data to understand video game experiences*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). pp. 381-390. http://doi.acm.org/10.1145/2556288.2557326

Teague, D., Corney, M., **Ahadi, A.**, & Lister, R. (2012). *Swapping as the "Hello World" of Relational Reasoning: Replications, Reflections and Extensions*. Paper presented at the Australasian Computing Education Conference (ACE 2012). pp. 87-94.

http://www.crpit.com/confpapers/CRPITV123Teague.pdf

Teague, D., Corney, M., Fidge, C., Roggenkamp, M., **Ahadi, A.**, & Lister, R. (2012). *Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming*. Paper presented at the Australasian Association for Engineering Education Conference (AAEE 2012). http://www.aaee.com.au/conferences/2012/documents/abstracts/aaee2012-submission-22.pdf

Teague, D., Corney, M., **Ahadi, A.**, & Lister, R. (2013). *A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers*. Paper presented at the 15th Australasian Computing Education Conference (ACE 2013).

Teague, D., & Lister, R. (2014a). *Longitudinal Think Aloud Study of a Novice Programmer*. Paper presented at the Australasian Computing Education Conference (ACE 2014). pp. 41-50. http://crpit.com/Vol148.html

Teague, D., & Lister, R. (2014b). *Manifestations of Preoperational Reasoning on Similar Programming Tasks*. Paper presented at the Australasian Computing Education Conference (ACE 2014). pp. 65-74. http://crpit.com/Vol148.html

Teague, D. and Lister, R. (2014c) *Programming: Reading, Writing and Reversing*. 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'14), June 23-25, 2014, Uppsala, Sweden.

Teague, D. and Lister, R. (2014d) *Blinded by their Plight: Tracing and the Preoperational Programmer*. Paper presented at the 25th Anniversary Psychology of Programming Annual Conference (PPIG), Brighton, England, 25th-27th June 2014. (i.e. this conference)

Teague, D. (2014e) Neo-Piagetian Theory and the Novice Programmer. Doctoral Consortium paper presented at the 25th Anniversary Psychology of Programming Annual Conference (PPIG), Brighton, England, 25th-27th June 2014. . (i.e. this conference)

Winters, T. and Payne, T. (2005). What do students know?: an outcomes-based assessment system. In Proceedings of the First international Workshop on Computing Education Research (Seattle, WA, USA, October 1–2). ICER '05. ACM, New York, NY, 165–172.

Neo-Piagetian Theory and the Novice Programmer

Donna Teague

Faculty of Science and Engineering
Queensland University of Technology
d.teague@qut.edu.au

Keywords: POP-I.B. barriers to programming, POP-II.A. novices, POP-V.A. Neo-Piagetian, POP-V.B. protocol analysis, POP-VI.E. computer science education research

Abstract

This PhD research draws on neo-Piagetian theories of cognitive development to explain how novices learn to program. From an interpretive perspective, we used think aloud studies to observe novice programmers completing simple programming tasks in order to determine the reasoning skills they had utilised. The concurrent verbal reports from the think aloud studies are triangulated with in-class test data of large student cohorts to maximise the quality of the research data and validity of the results. The outcome of the research will be a mapping of the neo-Piagetian stages to behaviours exhibited by novice programmers.

1. Motivation

Why do so many students find programming hard to learn?

One might define an "expert programmer" (borrowing from Bloom's taxonomy) as someone who is able to remember, understand and apply programming concepts, analyse and evaluate programs, and ultimately create their own (Krathwohl, 2002). Expert programmers exhibit a high level of abstract reasoning, and we need to know how that reasoning is developed in order to influence the transition of students from one stage to the next more complex level of cognition in that domain. The results of this study will have pedagogical implications.

2. Background

Piagetian-based cognitive development theories (Flavell, 1977; Morra, Gobbo, Marini, & Sheese, 2007; Piaget, 1952) provide a framework for describing the domain-specific development of cognition. There has been some work in the mathematics domain (Keats, Collis, & Halford, 1978; Ojose, 2008) and also in identifying misconceptions with programming concepts (Clancy, 2004; du Boulay, 1989; Pea, 1986) which help us to understand the problems students have with programming. However, little is known about when, why and how abstract reasoning skills are developed in the programming domain.

It has been hypothesised (Lister, 2011) that programming students exhibit characteristics at each of the sensorimotor, preoperational, concrete operational and formal operational levels described in neo-Piagetian theory. If so, this would explain why some novice programmers struggle with programming, because neo-Piagetian theory considers adequate exposure to the domain of knowledge as paramount to the progression to the next more complex level of abstract reasoning.

At the least most mature stage of cognitive development, sensorimotor, a novice programmer has difficulty tracing (hand executing) code and has a fragile model of program execution. At the next more mature stage, a preoperational novice has overcome any early misconceptions and can now trace code with some accuracy. However, a preoperational programmer is as yet unable to reason about the code's purpose because they are also unable to see any relationship between various parts of the code. They rely on specific values and inductive inference, based on input/output pairs, to determine the outcome of code. A concrete operational programmer, however, is able to reason about code's

purpose simply by reading it, as they are able to conceive the unified whole, rather than just a collection of parts. A defining characteristic of the concrete operational stage is the ability to reason about concepts of conservation, reversibility and transitive inference. By the time a programmer reaches the formal operational stage, they can reason logically, consistently and systematically about code. They can understand and use abstractions to reason about and create complex programs of their own.

We each progress through the stages at our own rate. So by virtue of our current teaching practices, students who develop slowly through the early stages would likely be expected by their teachers to operate at a more mature level of reasoning before they are developmentally able.

3. Research Questions

In order to operationalise the question:

> *Why do so many students find programming hard to learn?*

We decompose it into the following questions:

1. Over time, do students tend to exhibit characteristics of each of the neo-Piagetian stages in order from least to most mature?
2. If a student is found to exhibit characteristics of one of the neo-Piagetian stages on a particular programming problem, does that student tend to manifest that same neo-Piagetian stage on similar programming problems?
3. Does a student's programming ability improve with progression into the next more mature neo-Piagetian stage?

4. Objectives

The main objectives of this research project are to:

1. document the reasoning skills and behaviours of novice programmers;
2. analyse the manifestation of those behaviours using neo-Piagetian cognitive development theory; and
3. develop a mapping between neo-Piagetian cognitive development stages and programming reasoning skills and behaviour

The expected outcome of this research is the formalisation of the development trajectory from novice programmer in terms of the evolution of reasoning skills. The formalisation will include behaviours likely to be exhibited, and artefacts likely to be produced by a programmer at each stage of development. Should it be found that novices develop programming skills according to the sequential, cumulative neo-Piagetian stages of development, we can then talk about the "when" of novices' ability becoming expert, rather than "if". It will mean that, given time, everyone can learn to program.

The pedagogical significance of this research is in providing the framework upon which to identify students' cognitive development in programming based on their abstract reasoning behaviour. Novices' learning can then be supported and scaffolded in a stage-appropriate manner in order to better influence their progression to the next more complex level of cognitive development in programming.

5. Method

Normally the only true artefacts we have of our programming students' work are their exam scripts. That is, the exam paper on which they write their solutions to the exam questions. Many assumptions are made by academics about the reasoning employed by students to produce their solutions. From an exam script, we actually know very little about the thought processes, problem solving and reasoning

skills the students employed. Likewise, we can only guess about any misconceptions they may have had. In order to gather that sort of rich data, it is necessary to observe the phenomenon as it occurs.

This research uses microgenetic analysis to closely observe novice programmers as they complete programming exercises. The microgenetic research method has been used in other domains to study cognitive development and has been defined as having three key characteristics (Siegler & Crowley, 1991):

1. observations span a period of rapidly changing competence;
2. the density of observations is high relative to the rate of change in competence; and
3. observations are subjected to intensive analysis, with the goal of inferring the processes that gave rise to the change.

The microgenetic research for this project involved interviews with novice programmers. Each interview session was audio-taped, and the participants use a SmartPen (LiveScribe, 2014) and dot paper to complete each exercise while thinking aloud (Ericsson & Simon, 1993). The SmartPen produces a replayable "pencast" which is then able to be encoded for analysis.

Data was also collected from in-class tests and final exams of entire cohorts of novice programmers using the same or similar programming exercises as used in the think aloud studies. Because of this, we can employ triangulation of the qualitative and quantitative methods to check the validity of our findings. From the in-class test artefacts we can make generalisations about the entire cohort, and any problems/misconceptions identified here have informed a specific investigation with the think aloud studies. Conversely, the interesting behaviour or misconceptions observed in the think aloud sessions have lead us to deploy appropriate tests to the entire cohort in order to identify patterns and test research theories or assumptions.

In our final analysis, think aloud students' performance will be categorised according to the skills exhibited while processing tasks associated with certain levels of abstract reasoning. Analysis of the verbal reports will allow quantification of certain behavioural components for each of the students, and allow comparison of students operating at the same and different levels of reasoning.

6. Results to Date

We designed programming tasks which tested for the existence of reasoning skills described in neo-Piagetian theory and then gave them to introductory programming students in the classroom ("in-class tests"). We found evidence that many students continued to struggle with very simple programming tasks which tested them at the concrete operational level (Teague, Corney, Fidge, et al., 2012). The data we have collected from these in-class tests, as well as final exams, supports our initial claim (Corney, Lister, & Teague, 2011) that the problems some students face in learning to program start very early in the semester (Teague, Corney, Ahadi, & Lister, 2012).

We then presented empirical results (Corney, Teague, Ahadi, & Lister, 2012) in support of the neo-Piagetian perspective that novice programmers pass through at least three stages: *sensorimotor*, *preoperational*, and *concrete operational* stages, before eventually reaching programming competence at the *formal operational* stage. The programming exercises we gave novices tested for the concrete operational abilities to reason with quantities that are conserved, processes that are reversible and properties that hold under transitive inference. Examples of each of these types of exercises are shown in Figure 1, Figure 2 and Figure 3. The empirical results from these tests demonstrate that many students struggle to answer these problems, despite their apparent simplicity.

Below is incomplete code for a method which returns the smallest value in the array `x`. The code scans across the array, using the variable `minsofar` to remember the smallest value seen thus far. There are two ways to implement remembering the smallest value seen thus far: (1) remember the actual value, or (2) remember the value's position in the array. Each box below contains two lines of code, one for implementation (1), the other for implementation (2). First, make a choice about which implementation you will use (it doesn't matter which). Then, for each box, draw a circle around the appropriate line of code so that the method will correctly return the smallest value in the array.

```
public int min(int x[] ){

  int minsofar =  (a) 0
                  (b) x[0]      ;

  for ( int i=1 ; i<x.length ; ++i )
  {
    if ( x[i] <   (c) minsofar
                  (d) x[minsofar]   )

       minsofar =  (e) i
                   (f) x[i]       ;
  }
    return        (g) minsofar
                  (h) x[minsofar]   ;
}
```

Figure 1: Test of Conservation

The purpose of the following code is to move all elements of the array `x` one place to the ***right***, with the ***rightmost*** element being moved to the ***leftmost*** position:

```
int temp = x[x.length-1];

for (int i=x.length-2; i>=0; --i)
    x[i+1] = x[i];

x[0] = temp;
```

Write code that undoes the effect of the above code. That is, write code to move all elements of the array `x` one place to the ***left***, with the ***leftmost*** element being moved to the ***rightmost*** position.

Figure 2: Test of Reversibility

In plain English, explain what the following segment of Java code does:

```
bool bValid = true;

for (int i = 0; i < iMAX-1; i++)
{
  if (iNumbers[i] > iNumbers[i+1])

     bValid = false;

}
```

Figure 3: Test of Transitive Inference

From our observational studies, we have pencasts of students at various levels of competency, completing exercises similar to those shown in Figure 1, Figure 2 and Figure 3 which tested their level of abstract reasoning. These think aloud sessions confirmed that students can still be at the sensorimotor and preoperational stages even after two semesters of learning to program (Teague, Corney, Ahadi, & Lister, 2013). These results are the first observational data that is described explicitly in neo-Piagetian terms. Further microgenetic research have provided evidence of novice programmers' evolving ability to reason abstractly which has been analysed using the neo-Piagetian framework (Teague & Lister, 2014a, 2014d).

What has emerged from the think aloud studies is evidence for three different ways in which students reason about programming which correspond to the first three neo-Piagetian stages (Lister, 2011). At the sensorimotor stage, novices programmers exhibit misconceptions and other errors that are already well established in the literature (e.g., Du Boulay (1989)). At the next stage, known as the preoperational stage, students can correctly trace a program, but they can neither reason about code nor see a relationship between parts of a program. Preoperational programming students are not yet equipped with skills which allow them to reason about conservation, transitive inference and reversibility. They rely heavily on specific values in order to reason about, trace and write program code. Many of our think aloud students have exhibited behaviour which is consistent with this type of preoperational reasoning.

We have evidence of students exhibiting characteristics of each of the neo-Piagetian stages in order from least to most mature (Teague & Lister, 2014b). However, our data supports the overlapping waves model which explains why students can exhibit characteristics from two or more stages as they

develop skills in the domain (Boom, 2004; Feldman, 2004; Siegler, 1996). In this overlapping waves model, characteristics of the early stage dominate behaviours initially, but as cognitive progress is made there is an increase in use of the next more mature level of reasoning and a decrease in the less mature. In this way, there is concurrent use of multiple stages of reasoning.

One of our series of think aloud sessions used two exercises that required very similar programming skills and we discovered that students who manifested preoperational behaviour were able to complete one, but not the other (Teague & Lister, 2014c). This was because the second task, although functionally equivalent, required the ability to reason about concepts that only someone at the concrete operational level was likely to be able to do.

In-class test data supports our hypothesis that preoperational reasoning may be the norm for novice programmers rather than being peculiar to the small number of students in our think aloud studies. We observed one particular student for several semesters in think aloud studies using a wide variety of programming tasks and were able to find evidence that programming ability improved with the increased ability to reason abstractly about programming code. Our evidence also suggests that it may take several semesters or even years of exposure to programming to develop operational reasoning in this domain (Teague & Lister, 2014b, 2014c). Yet it is at this concrete operational level that we often assume our students are capable of working, and as a result students may be struggling due to inappropriate teaching resources rather than an inability to learn to program.

7. References

Boom, J. (2004). Commentary on: Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 239-247.

Clancy, M. (2004). Misconceptions and Attitudes that Interfere with Learning to Program *Computer Science Education Research*. London, UK: Taylor & Francis.

Corney, M., Lister, R., & Teague, D. (2011). *Early Relational Reasoning and the Novice Programmer: Swapping as the "Hello World" of Relational Reasoning*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011), Perth. http://crpit.com/confpapers/CRPITV114Corney.pdf

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Paper presented at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia. http://crpit.com/confpapers/CRPITV123Corney.pdf

du Boulay, B. (1989). Some Difficulties of Learning to Program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the Novice Programmer* (pp. 283-300). Hillsdale, NJ: Lawrence Erlbaum.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology.

Feldman, D. H. (2004). Piaget's stages: the unfinished symphony of cognitive development. *New Ideas in Psychology, 22*, 175-231.

Flavell, J. H. (1977). *Cognitive Development*. Englewood Cliffs, NJ: Prentice Hall.

Keats, J., Collis, K., & Halford, G. (1978). Operational Thinking in Elementary Mathematics *Cognitive Development: Research Based on a Neo-Piagetian Approach* (pp. 221-248). Chichester: John Wiley & Sons.

Krathwohl, D. R. (2002). A Revision of Bloom's Taxonomy: An Overview. *Theory into Practice, 41*(4), 212-218.

Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Paper presented at the 13th Australasian Computer Education Conference (ACE 2011), Perth, WA. http://crpit.com/confpapers/CRPITV114Lister.pdf

LiveScribe. (2014). Retrieved March 17, 2014, from https://www.smartpen.com.au/

Morra, S., Gobbo, C., Marini, Z., & Sheese, R. (2007). Cognitive Development: Neo-Piagetian Perspectives. *Psychology Press*.

Ojose, B. (2008). Applying Piaget's Theory of Cognitive Development to Mathematics Instruction. *The Mathematics Educator, 18*(1), 26-30.

Pea, R. D. (1986). Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research, 2*(1), 25-36.

Piaget, J. (1952). *The Origins of Intelligence in Children*. New York: International University Press.

Siegler, R. S. (1996). *Emerging Minds*. Oxford: Oxford University Press.

Siegler, R. S., & Crowley, K. (1991). The Microgenetic Method: A Direct Means for Studying Cognitive Development. *American Psychologist, 46*(6), 606 - 620.

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2012). *Swapping as the "Hello World" of Relational Reasoning: Replications, Reflections and Extensions*. Paper presented at the Australasian Computing Education Conference (ACE 2012), Melbourne. http://crpit.com/confpapers/CRPITV123Teague.pdf

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2013). *A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers*. Paper presented at the 15th Australasian Computing Education Conference (ACE 2013), Adelaide, Australia. http://crpit.com/confpapers/CRPITV136Teague.pdf

Teague, D., Corney, M., Fidge, C., Roggenkamp, M., Ahadi, A., & Lister, R. (2012). *Using Neo-Piagetian Theory, Formative In-Class Tests and Think Alouds to Better Understand Student Thinking: A Preliminary Report on Computer Programming*. Paper presented at the Australasian Association for Engineering Education Conference (AAEE 2012), Melbourne. http://eprints.qut.edu.au/55828/

Teague, D., & Lister, R. (2014a). *Blinded by their Plight: Tracing and the Preoperational Programmer*. Paper presented at the Psychology of Programming Interest Group (PPIG) 2014, Sussex, UK.

Teague, D., & Lister, R. (2014b). *Longitudinal Think Aloud Study of a Novice Programmer*. Paper presented at the Australasian Computing Education Conference (ACE 2014), Auckland, New Zealand. http://crpit.com/confpapers/CRPITV148Teague.pdf

Teague, D., & Lister, R. (2014c). *Manifestations of Preoperational Reasoning on Similar Programming Tasks*. Paper presented at the Australasian Computing Education Conference (ACE 2014), Auckland, New Zealand. http://eprints.qut.edu.au/67314/

Teague, D., & Lister, R. (2014d). *Programming: Reading, Writing and Reversing*. Paper presented at the ITiCSE '14, Uppsala, Sweden.

Self-explaining from videos as a methodology for learning programming

Viviane Cristina Oliveira Aureliano

Center of Informatics - Federal University of Pernambuco
Federal Institute of Pernambuco - Campus Belo Jardim
vcoa@cin.ufpe.br

Keywords: POP-I.A. Learning to program; POP-I.B. Choice of methodology; POP-II.A. Novices.

Abstract

The literature has shown that novice learners have several difficulties in acquiring programming skills. As a way to overcome these difficulties, novice learners should be guided in the process of studying the programming process. One evidence-based practice for improving student learning by guiding them while they are studying from some instructional material is through the use of self-explanations. Video recordings have been proposed as an ideal instructional material for presenting the dynamics of programming to novices. In this context, my research proposes that novices should be guided in the process of studying video recordings by means of self-explanation. In order to evaluate the benefits of using the proposed methodology, two studies will be realized as part of this research.

1. Introduction

Years of research have shown that novice learners experience several difficulties in acquiring programming skills. They have fragile knowledge in learning programming (Perkins and Martin 1986). One type of fragile knowledge is the inert knowledge, which is the knowledge novice learners have, but fail to retrieve when it is necessary (CTGV 1992). Novices use inappropriate study methodologies and do not take constant effort to develop their programming skills (Gomes and Mendes 2007). However, the major difficulty experienced by novices is how to combine and properly use the basic structures to build a program (Caspersen & Kölling 2009).

Stepwise Improvement is a conceptual framework that describes programming as an incremental process that encompasses three different activities, extension, refinement and restructuring (Caspersen 2007; Caspersen & Kölling 2009). In the area of programming education, the authors advocate the application of the framework in order to guide the process of teaching and learning programming. In this case, it provides guidance regarding the structure of the instructional materials in a way that novices learn programming by extending, refining and restructuring small pieces of code systematically and incrementally during the course.

Bennedsen and Caspersen (2008) have proposed the use of videos as an ideal medium for revealing the programming process for learners. Instead of textbooks that are static, the authors advocate that videos are an important instrument to expose the programming process since they are essentially a dynamic medium. Caspersen (2007) extended this idea by proposing in his doctoral dissertation that worked examples in videos could be used as an instructional material in an introductory programming course. The application of worked examples and practice exercises that are similar in structure but diverse in situations of use has been demonstrated as a good strategy to promote the transfer of learning (Clark, Nguyen and Sweller 2006).

Self-explanation is a type of dialog that learners have with themselves while they are learning from different instructional materials (Bielaczyc, Pirolli & Brown 1995; Chi, Bassok, Lewis, Reimann & Glaser 1989; Clark et al. 2006). It is an effective learning activity that leads learners to move further in their knowledge about a subject (Fonseca & Chi 2011) by building new knowledge through the refinement of the given information (Chiu & Chi 2014).

As a way to overcome the aforementioned difficulties, novice learners should be guided in the process of studying the programming process. This guidance will be provided by means of the use of self-explanation practice. Since video recordings have been proposed as an ideal instructional material for

exposing the programming process to novices, the self-explanation practice will be applied with this kind of instructional material. In the presented context, my PhD research proposes that novice learners should be guided in the process of studying video recordings by means of self-explanation. This methodology of using videos in conjunction with self-explanations will adopt the programming pathway defined by the Stepwise Improvement framework.

2. Theoretical framework

2.1 Cognitive Load Theory

Cognitive Load Theory (CLT) provides a set of learning principles and related instructional guidelines that are proven to result in instructional environments that are more effective regarding students learning (Clark et al. 2006). In order to provide such a set of principles and guidelines, CLT is closely aligned with the human cognitive structure and the way that people learn. In their book, the authors of the theory explain that the human cognitive structure has schemas or patterns as units of information. In order to manipulate and store this information, the human cognitive structure has also two memory systems, the working memory and the long-term memory. The working memory is the active partner of our cognitive structure. It is responsible for processing new information while we are learning and also for organizing new schemas from the ones that we already have. The working memory works closely together with the long-term memory. While the working memory has a very limited capacity, long-term memory has a huge capacity for storing information and, once we learn, the resultant schemas are stored on it. Because of this, the long-term memory is our main knowledge repository.

According to Clark et al (2006), while students are learning they have to deal with three different types of cognitive load: intrinsic, germane and extraneous. Intrinsic cognitive load results from the complexity of the instructional content. Germane cognitive load is the productive mental effort required for building new schemas and relevant to the learning process. Extraneous cognitive load is defined as the unproductive (or irrelevant) mental effort that does not lead to learning. In order to have efficient learning environments, teachers have to manage intrinsic cognitive load, increase germane cognitive load and reduce extraneous cognitive load. Doing that, they can free the working memory from irrelevant mental effort. Consequently, it is possible to apply that free space for the work demanded to integrate new information into schemas in the long-term memory.

2.2 Stepwise Improvement

Stepwise Improvement is a conceptual framework that describes programming as a systematic and incremental process that encompasses three different activities, extension, refinement and restructuring (Caspersen 2007; Kölling & Caspersen 2009). Extension activity occurs when the specification is expanded in order to cover more use cases. Refinement activity occurs when abstract code is modified towards an executable code in order to implement the current specification. Restructure activity occurs when an improvement of nonfunctional aspects of a solution is made, but this modification does not involve a change in its apparent behavior. These activities are organized in a three-dimensional space that is explored by programmers while they are building programs.

Caspersen and Kölling (2009) advocate the application of Stepwise Improvement by using guided tours rather than random walks. In order to conduct learners' steps in the programming space offered by the framework, the authors state that teachers should be concerned with the right amount of guidance and scaffolding (Collins, Brown & Newman 1989) given during the instruction. It guarantees to balance important aspects of programming education: train learners' programming skills in a proper manner and, at the same time, maintain the cognitive load while students are learning under control.

In that sense, Stepwise Improvement brings a significant contribution to the programming education field by providing guidance in the three activities that it encompasses. Regarding the extension and restructure activities, it provides guidance in the way that the instructional material is organized. Therefore, programming textbooks, assignments, lectures and examples, among other instructional materials, can be structured using the set of steps defined in the framework. For instance, Stepwise

Improvement has been employed in the organization of the Greenfoot's textbook (Kölling 2010) and the problem solving videos of the Joy of code channel on Youtube (Kölling 2012). In addition, regarding the refinement activity, Caspersen (2007) defined in his doctoral dissertation an object-oriented programming process for teaching novices.

2.3 Worked examples

One effective manner of maximizing the germane cognitive load is by using worked examples. Worked examples are sequences of steps defined to demonstrate one particular task performance or to solve a specific problem (Clark et al. 2006). They are especially important for novice learners, once they represent a step-by-step guide necessary to execute a procedure or reach a goal. For this reason, worked examples help novice learners to build new schemas while they are acquiring a new knowledge or skill.

In that sense, worked examples are an effectively evidence-based instructional material. In their book, Clark et al. (2006) demonstrate some advantages of using worked examples. For instance, learners using pairs of worked examples-problems could spend less time studying and still have equivalent learning results rather than learners using all problems as practice exercises. In addition, the authors showed that the application of worked examples and practice exercises that are similar in structure but diverse in situations of use is a good strategy to promote the transfer of learning.

Worked examples have been used in different subjects, such as Physics (Chi et al. 1989), Chemistry (Crippen & Earl 2007) and Programming (Abdul-Rahman & du Boulay 2014; Caspersen 2007), among others. In addition, they can be shown in various formats and modalities, including text (Chi et al. 1989), web (Crippen & Earl 2007), learning environment (Abdul-Rahman & du Boulay 2014) or video recordings (Caspersen 2007).

2.4 Self-explanations

Self-explanation is a kind of dialog that learners have with themselves while they are learning from different instructional materials (Bielaczyc et al. 1995; Chi et al. 1989; Chi et al. 1994; Clark et al. 2006; Crippen & Earl 2007). Fonseca and Chi (2011) presented a literature review about the self-explanation effect (Chi et al. 1989) and its effectiveness as a learning activity that leads people to produce overt output that goes further the presented information. In this sense, the process of self-explaining benefits the building of new knowledge through the refinement of the given information, associating this new information with prior knowledge, reasoning about it and connecting it with other different pieces of information (Chi & Chiu 2014). Self-explanations also promote the exploration of worked examples by learners in a deeper way and maximize the germane cognitive load (Clark et al. 2006). Because of their evidence-based benefits, self-explanations are considered one of the seven practices recommended for improving student learning (Pashler et al. 2007).

The self-explanation effect was demonstrated in different subjects, such as Biology (Chi et al. 1994), Physics (Chi et al. 1989) and Programming (Bielaczyc et al. 1995), among others. In addition, self-explanations can be elicited from various formats and modalities of instructional material, including text (Chi et al. 1994; Chi et al. 1989; Bielaczyc et al. 1995) and worked examples (Clark et al. 2006). The activity of self-explanation may be spontaneously generated (Chi et al. 1989). Also, it can be prompted and trained (Bielaczyc et al. 1995; Crippen & Earl 2007).

3. Statement of thesis

As mentioned before, Stepwise Improvement is a conceptual framework that describes programming as a systematic and incremental process that takes place in a three dimensional space of extension, restructure and refinement activities. In order to move in this space, Caspersen and Kölling (2009) advocate guided tours instead of random walks. Guidance has been provided regarding the manner that the instructional material for programming courses may be presented to students and also the programming activity itself. However, the framework does not provide any guidance regarding the way that students may study and comprehend this instructional material.

In addition, the concept of self-explanation was cited previously as well as its benefits as a learning activity. Self-explanations promotes the building of new knowledge through the refinement of the given information, associating this new information with prior knowledge, reasoning about it and connecting it with other different pieces of information (Chi & Chiu 2014). Explaining to oneself is an effective practice recommended for improving students learning (Pashler et al 2007). In particular, this practice is especially important for novice learners in programming and contributes to their learning while they are studying alone (Bielaczyc et al. 1995).

Nowadays, the use of videos for teaching programming is becoming more and more popular. For instance, numerous videos are available in Youtube channels or in organizations as Khan Academy (2014) and Code.org (2014). However, the use of videos for teaching programming is not a novelty. Some years ago, Bennedsen and Caspersen (2008) showed the benefits of using videos as a tool for revealing the programming process for learners. Instead of textbooks that are static, the authors advocate that videos are an important instrument to expose the programming process since they are essentially a dynamic medium. In addition, using videos students can manipulate these videos - play, forward or rewind - in their own pace and how many times they need.

Stepwise improvement has been employed in the incremental structure of different video collections for learning programming. For example, the structure of problem solving videos of the Joy of code channel on Youtube (Kölling 2012) followed this framework. Also, Caspersen (2007) adopted this idea by proposing in his doctoral dissertation that worked examples in videos could be used as an instructional material in an introductory object-oriented programming course.

I agree with Bennedsen and Caspersen (2008) that videos are an important medium for exposing the programming process. In particular, because of their aforementioned advantages, I believe that videos are an important resource when students are studying programming outside the classroom. Also, I believe that the process of studying programming can be guided by prompting self-explanations from the students while they are studying from these videos. In this context, the main objective of this PhD dissertation is to provide guidance regarding the way students are studying and comprehending videos for learning programming. This guidance will be provided by means of prompting self-explanation while novice learners are studying the programming pathway defined in the Stepwise Improvement framework.

In order to do that, the main research question that guides this PhD is *how novice learners in programming could benefit from using self-explanations while studying video recordings that expose the programming process?* In this sense, novice learners would benefit if they achieve better results, obtain higher grades, code faster or make fewer errors.

4. Research goals

Aiming to answer the question which conducts this research, I am realizing two different studies. Both studies are based on the previous researches developed in the works of Bielaczyc et al. (1995) and Chi et al. (1994), but instead of studying plain texts, learners are using videos as an instructional material.

In the first study, I am using Java as programming language, Greenfoot (2011) as development tool and the problem solving videos of the Joy of code channel on Youtube (Kölling 2012). It has the following objectives: (i) check whether the proposed self-explanations prompts are appropriate for the videos; (ii) analyse when to use the self-explanations prompts and the videos together; (iii) verify if there is any benefit of using self-explanations while studying videos that expose the programming process (if so, which are these benefits); and (iv) refine the self-explanations prompts and the manner that they should be used. In order to do that, the participants in the study should be divided in three different groups: (i) the instructional group in which the self-explanations are asked while showing the video; (ii) the instructional group in which the self-explanations are asked only when the video ends; and (iii) the control group.

This first study consists of three phases: (i) pre-intervention phase; (ii) instructional intervention phase; and (iii) post-intervention phase. The objective of the pre-intervention phase is to provide the participants with a common prior knowledge of the programming language Java and familiarize them

with the self-explanation prompts and the think aloud process. The objective of the instructional intervention phase is training the two instructional groups in how to use the self-explanation prompts. In that sense, for the case of the first instructional group, the video is divided in goals and for each goal the video is paused and the student is prompted with the self-explanation questions. On the other hand, for the second instructional group, the self-explanations questions will be prompted only when the video finishes. At the end of this study, the post-intervention phase aims at collecting data from all participants after the instructional intervention phase.

The students are assessed in two different ways. First, after studying each programming video, students are asked to solve similar problems in order to assess their process of transfer of knowledge. Second, they are assessed by means of the explanations provided by them while studying the programming videos in the instructional and post-intervention phases. In addition, I am getting feedback from the students about their usage of the videos and self-explanation prompts for learning programming.

As mentioned previously, worked examples are an effectively evidence-based instructional material that accelerates the novice's process of learning when adopted together with practice exercises. The concept of worked examples is closely connected with the self-explanations' concept, since self-explanations consist in a manner of promoting deeper learning of worked examples. Aiming to answer the question which conducts this research, I believe that an effective approach is to vary the kind of video recordings exposing the programming process adopted on it. For this reason, the second study will be extremely similar in structure to the first one regarding the objectives, types of group participants, sequence of phases, programming language and development tool. However, it will vary regarding the following characteristics: (i) a larger number of participants; (ii) a refined set of self-explanations prompts and practice problems; and (iii) worked examples in videos as instructional materials instead of using the problem solving videos of the Joy of code channel on Youtube. To do that, I am producing these worked examples in videos in a similar structure proposed in the Joy of code's videos.

5. Dissertation status and expected contributions

I am enrolled in an academic PhD programme in Computer Science at the Informatics Centre of the Federal University of Pernambuco, Brazil. I am also a teacher at the Federal Institute of Pernambuco, Brazil, where I mainly teach programming language subjects. My supervisor is Professor Patrícia Tedesco. The PhD programme in which I participate must be completed in four years and I am at the beginning of the fourth year. I have already completed all six courses that I needed. Also, I defended my research proposal in November 2013. Currently, I am working as a visiting PhD student with Professor Michael Caspersen at Centre for Science Education at Aarhus University, Denmark.

The expected contributions of this PhD research are: (i) a methodology for studying programming from videos that expose the programming process in a deeper manner by self-explaining them; (ii) a study to analyse how novice learners in programming would benefit from using self-explanations while studying videos that expose the programming process; and (iii) a suite of worked examples using Java as programming language and Greenfoot as development tool, enriched with self-explanations prompts. All these contributions use the Stepwise Improvement as the principal conceptual framework.

6. Acknowledgments

I would like to thank CAPES Foundation for the scholarship provided under report nº 1127-91-3.

7. References

Abdul-Rahman, S. S., & du Boulay, B. (2014). Learning programming via worked-examples: Relation of learning styles to cognitive load. *Computers in Human Behavior, 30,* 286-298. doi:10.1016/j.chb.2013.09.007

Bennedsen, J., & Carpersen, M. E. (2008). Exposing the Programming Process. In J. Bennedsen, M. E. Carpersen, & M. Kölling (Eds.), *Reflection on the Theory of Programming: Methods and Implementation* (pp. 6-16). Berlin, Germany: Springer-Verlag.

Bielaczyc, K., Pirolli, P., & Brown, A. L. (1995). Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and Instruction, 13,* 221-253. doi:10.1207/s1532690xci1302_3

Caspersen, M. E. (2007). Educating novices in the skills of programming. PhD dissertation PD-07-4, Department of Computer Science, University of Aarhus.

Caspersen, M. E., & Kölling, M. (2009). STREAM: A First Programming Process, ACM *Transactions on Computing Education, 9 (1),* 4:1-29. doi:10.1145/1513593.1513597

Chiu, J. L., & Chi, M. T. H. (2014). Supporting Self-Explanation in the Classroom. In V.A. Benassi, C.E. Overson, & C.M. Hakala (Eds.). *Applying science of learning in education: Infusing psychological science into the curriculum.* Retrieved from: http://bit.ly/KbYLtG

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, M., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13,* 145-182. doi:10.1207/s15516709cog1302_1

Chi, M. T. H, deLeeuw, N., Chiu, M., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science, 18,* 439-477. doi:10.1016/0364-0213(94)90016-7

Clark, R., Nguyen, F., & Sweller, J. (2006). Efficiency in learning: evidence-based guidelines to manage cognitive load. San Francisco, CA: John Wiley & Sons, Inc.

Cognition and Technology Group at Vanderbilt - CTGV (1992). The Jasper series as an example of anchored instruction: Theory, program description, and assessment data. *Educational Psychologist, 27,* 291-315. doi:10.1207/s15326985ep2703_3

Collins, A., Brown, J. S., & Newman, S. E. (1987). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. Technical Report No. 403, University of Illinois.

Code.org (2014) Retrieved from: http://code.org/learn

Crippen, K. J., & Earl, B. L. (2007). The impact of Web-based worked examples and self-explanation on performance, problem solving, and self-efficacy. *Computers & Education, 49 (3),* 809-821. doi:10.1016/j.compedu.2005.11.018

Gomes, A., & Mendes, A. J. (2007) Learning to Program - Difficulties and Solutions. In *Proceedings of the International Conference on Engineering Education, Coimbra, Portugal.* Retrieved from: http://bit.ly/1ldTqkE.

Greenfoot (Version 2.3.0) [Computer software] (2011) Retrieved from http://www.greenfoot.org/door

Fonseca, B. & Chi, M.T.H. (2011). Instruction based on self-explanation. In R. E. Mayer, & P. A. Alexander (Eds.), *The Handbook of Research on Learning and Instruction* (pp. 296-321). NY: Routledge Taylor and Frances Group.

Khan Academy (2014) Retrieved from https://www.khanacademy.org/

Kölling, M. (2010) Introdution to Programming with Greenfoot: Object-Oriented Programming in Java with Games and Simulations. New Jersey, USA: Pearson Higher Education.

Kölling, M. (2012). *The Joy of code* [Video recordings]. Retrieved from http://bit.ly/1jbTbHq

Pashler, H., Bain, P., Bottge, B., Graesser, A., Koedinger, K., McDaniel, M., and Metcalfe, J. (2007). Organizing Instruction and Study to Improve Student Learning (NCER 2007-2004). Retrieved from: http://ncer.ed.gov

Perkins, D. N.; Martin, F. (1986) Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers, First Workshop* (pp. 213–229). Norwood, NJ: Ablex.

Educational Programming Languages: The Motivation to Learn with Sonic Pi

Arabella Jane Sinclair[1]

Computer Laboratory 1, Cambridge University
ajs291@cl.cam.ac.uk

Abstract. This work explores the differences in novice users experiences when learning a simplified variant of the Ruby programming language for the first time. Sonic Pi and Kids Ruby both aim to teach users via a media-oriented set of programming exercises and environment, on the premise that this accessible domain will motivate novice learners. Users retention of the language and interaction style with each environment were measured and the results indicate that Sonic Pi facilitates a greater level of user experimentation and concept retention.

1 Introduction

This study explores the differences in learning motivation and effectiveness between the programming environments of Sonic Pi and Kids Ruby. Both are a simplified version of Ruby, a language which was designed in 1994 with the intention of being easy and enjoyable to use, yet still having powerful performance[11].

Kids Ruby uses turtle graphics as part of their lesson plans[6], following in the footsteps of LOGO, one of the first widely adopted educational languages in schools[9]. Like Logo, one of the aims of Sonic Pi is that learners have permission to explore using their own individual styles whilst creating something concrete[1]. However in its case this is done through allowing the learner to create their own sounds, emphasising the importance of creativity in the learning process and allowing learners immediate concrete feedback during the implementation of their ideas. The element of immediate feedback as applies to live-coding has been discussed as one attractive feature of Sonic Pi in terms of maintaining student engagement[2], however this study focuses more on the effects that learning programming through making music have on students.

There has been a lot of work on developing programming languages which lower the barrier to gaining programming proficiency, so there are various theories about the best types of language to do so, a taxonomy of which has been made for some of the better known educational languages[8]. The main point of agreement and conclusion in the hunt for a more accessible entry point into programming that can be drawn from that study is that a social barrier is in finding a real reason to program, and that the ideal educational language would (domain specific or not) have an application with a broad appeal for the more under-represented groups in computer science.

Even within the current novice programmer demographic, there is no single programmer archetype. The links between different programmer types and the environment that they are working in is an interesting one. It has been shown that in some environments, that levels of *tinkering*, the trial and error of new interactions with a computer interface, shown by the subject can have an effect on their performance on completing tasks[4]. Elsewhere it has been observed in a larger-scale study of novice computer science students that there are three distinct types of programmer, namely *stoppers, movers* and *extreme movers*[7]. Although in these studies a slightly different vocabulary is used to describe programmer behaviour, the underlying thought remains the same; that too much "brainless" interaction with the code produces negative effects

on performance, yet that a certain level is required in order for the subject to learn.

I believe that with Sonic Pi, the simplified syntax and broad appeal of the musical domain are very well-suited for the novice programmer. The interface of Sonic Pi has been specifically designed to make it simple and uncluttered, allowing the user to interact with it using only simple commands, directly linked to a response, removing the need for the normal wrapper commands needed within a typical coding interface. This is so as to allow novices to focus on key concepts for which they can see results initially rather than have to wade through many commands to create something simple. I would like to therefore explore whether the positive effects of this motivation help it achieve its more pedagogic goals, doing so via the teaching and examination of the core principles learned in combination with subject feedback and the tracking of interaction patterns whilst completing tasks.

Both languages are taught through a media-oriented environment, both encouraging learners to learn through exploring their creativity in order to complete the lesson material. It has been widely discussed that there should be more emphasis on the benefits of creativity in learning in schools [3] and it would be interesting to find out whether this is the case for these languages and whether the retention of the concepts learnt was more pronounced in those students who learned them in a more creative manner.

2 Research Questions

The aim of this research is to explore the effects the two learning environments have on the novice programmer in terms of motivation and assimilation and retention of information. The main research question is whether there is a significant improvement in the learning of programming concepts when Sonic Pi, rather than Kids Ruby is the teaching environment. This will be explored through the following questions:

- Is there a significant improvement in the learning of programming concepts when Sonic Pi is the teaching environment?
  - The hypothesis being that since the concept of music being "encoded" (notated) is already so normal *that the jump to programming it will be less than for example images. This should be explored in terms of measuring the confidence and interaction style with the enviroment of participants whilst performing programming tasks.*
- Is there a link between the amount of tinkering or tweaking of code a novice does whilst learning and their retention of concepts? And is there a higher retention of concepts when learning with Sonic Pi?
  In terms of the percentage of concepts retained approximately 3 weeks after initial tuition. Each environment will have its own set of keywords, in addition to those common to both; in the delayed recall of the concepts these will be measured and compared through keywords retained.
- In terms of the motivation whilst learning and engagement with the task, are there interesting traits we can observe in learners of Sonic Pi?
  This was be measured in terms of the correlation between the users own self efficacy reporting before and after the session, and in their level of interaction (in terms of prolificacy and tinkering) with the code

3 Experimental Method

This study took place in the form of a controlled experiment, where novice programmers were taught and made to perform programming exercises with either Sonic Pi or Kids Ruby. The

analysis made is a between-subject comparison of the two environments, as educational programming tools. The participants were selected to represent an equal distribution of half male, half female and of that half science and half arts first year undergraduate students. 12 unpaid volunteers were used, a number a little under that of the initial study of Logo by Seymour Papert[9]. Ideally a subsequent study would analyse the performance of school children in the same manner as here, since this is the target student of both packages, and could follow from both this study and the testing and material developed during the development of Sonic Pi. Although both the programming environments involved in this study are aimed at school children, the concepts and teaching are still applicable to these older equally inexperienced users, as they are very recent school leavers and are new to programming.

Tasks

The participants were initially asked to fill out a questionnaire, which aims to measure how confident they felt about learning a programming language. This questionnaire was a modified replication of the one proposed by Compeau and Higgins 1991 to measure self-efficacy [5]. The questionnaire was presented alongside an explanation of its origin, and the instruction to treat the questions of everyday life like the technology discussed was modern. Only the scale of the original questionnaire was changed, to make it from 1 to 5 rather than 1 to 10 to tie in with the scale on the other questionnaire participants received.

The participants then followed a tutorial sheet (see Appendix A and Appendix B) for a timed period of half an hour before being asked to then carry out a loosely constrained set of tasks in ten minutes, meant to assess their understanding of what they had learnt. These sessions were individually supervised, with the participant aware they could ask for help with any of the tasks. The tutorials consisted of a selection of relevant material from both languages' lesson guides, aimed to teach the concepts of conditionals, random numbers, and iteration. The tutorials were written in the same format, covering the same sets of pedagogic tasks; only the keywords and the descriptions of shape vs. sound differed. The tasks which followed were written and delivered in the same style as the tutorial sheet, and the participants interactions with the software were recorded through capturing the screen using CamStudio[1]. The video of their interaction with the software was then analysed and details such as the number of lines of code typed and alterations made were recorded.

After the participants' interactions with the software were complete, they filled out a second questionnaire about how interesting they found the lessons, how well they felt they had understood the concepts covered in the session and how well they might remember them. They were also asked if they had any creative hobbies, how much programming or mathematical experience they had, and whether they would view what they had just done as creative rather than following a set of instructions.

The participants were asked to provide contact details in case of any questions which might come up about the study, and were contacted approximately three weeks later and were asked to freely try to recall and list any of the commands they had learnt from their programming session. They had no expectation of this second test, which was done in the manner of the Rundus, Loftus and Atkinson[10] in their paper exploring the differing effects of immediate and free recall. The number of keywords recalled were then compared with the total number each participant had been exposed to. Of the 12 students used in the initial study, for this follow-up free recall test, only 10 participated.

[1] http://camstudio.org/

4 Results and Analysis

On the examination of the screen recordings from the experiment, the number of lines of code written in total by each participant were counted, as well as how much they were prone to *tinker* with their code. The criteria used to qualify an action as tinkering used here are:

- Tweaking a line already written
  These tweaks or modifications, if on the same line would be separated by the participant running the program, or changing another line before coming back to the tweaked line to perform a secondary modification.
- The deletion of a line of code
- The addition of a single line of code within a block
- The movement of a chunk of code (either the copying and pasting, or deletion and re-insertion)

Since the amount of tinkering will be dependent on the amount of code written and the two have a correlation coefficient of greater than 0.5 when compared, the value used to measure it is that of the percentage of lines tinkered with. There is a positive correlation between the quantity of tinkering and the number of lines typed, which was expected, from what intuitively you could expect, and the self-efficacy results discussed in the previously mentioned paper on tinkering [4]. To measure the participants' retention of the concepts learnt, the percentage of keywords recalled was measured. Each correctly remembered keyword was tallied and the inability to remember the specific name, but the convincing description of the function and how it was used would count as half a point.
The results of the experiment are displayed in the table below:

Table 1. Participant Results

Participant	Lines Written	Code Changes	Environment	% Tinkered	% Retained
1	73	19	S	26.03	56.25
2	33	9	R	27.27	27.27
3	43	8	S	18.60	81.25
4	44	7	S	15.90	85.50
5	53	14	S	26.41	-
6	47	14	S	29.78	75.00
7	35	9	R	25.71	63.63
8	53	32	S	60.37	87.50
9	32	18	R	56.25	-
10	38	6	R	15.79	54.54
11	33	3	R	9.09	63.63
12	63	15	R	23.80	72.72

Table 2. Where the Environment is either SonicPi or KidsRuby, %Tinkered is the percentage of lines tweaked out of the number written and %Retained is the percentage of commands or codewords retained after a period of approximately 3 weeks after the tutorial

From the data collected from the experiment, it is shown that those who learned with Sonic Pi not only typed more, but remember a significantly higher percentage of the commands used, with a p-value of 0.039 in comparison to KidsRuby.

When a comparison of lines of code vs. concepts retained is compared, there is no significant correlation, which cannot confirm the original second hypothesis mentioned in section 2.

Within the media-oriented languages considered here, there was no significant difference in motivation between learners between the two, however within the comments section of the

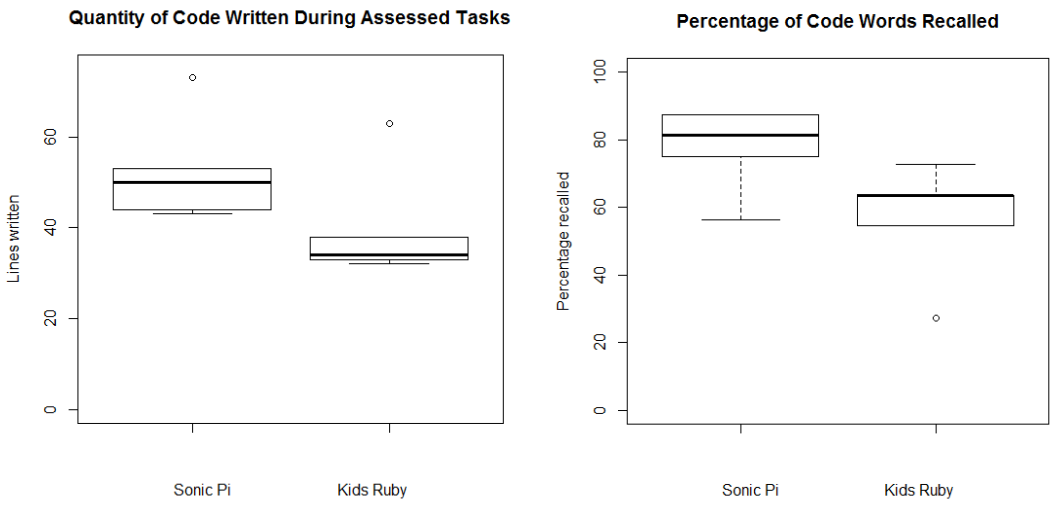


Fig. 1. The boxplots show the differences in distribution of lines of code written and percentage of codewords recalled between the two environments; both differences between the performances are significant with p-values of 0.043 for the quantity of code written and 0.039 for the percentage of code words recalled.

questionnaires, a qualitative conclusion which can be drawn is that the motivation levels for these sorts of creative, media-oriented methods of learning programming are much higher than other methods the participants could think of, however since this was not one of the hypotheses, there is no data to support this conjecture, and it would be an interesting parameter to explore in any follow-up experiments.

5 Discussion

The results indicate that using the environment of Sonic Pi is preferable to using Kids Ruby due to both a significantly greater ($p < 0.05$) number of codewords retained and lines written by students learning with Sonic Pi. The benefit of more codewords being retained long term is an obvious one. That of quantity of code typed less so, however, intuitively the more a learner is encouraged to interact constructively with a new environment the more they are expected to become familiar with its working.

There was no significant correlation (using Spearman's p correlation coefficient) between tinkering/lines of code written and concept retention; this was one of the hypotheses of the experiment, and it is surprising that there is no link. However, there is a greater level of code written, code tinkered with, and concepts retained in users of Sonic Pi.

There is a significant correlation between the quantity of code written (not just how many lines the resulting program is) and the level of tinkering (number of times code is tweaked over the course of its creation) with Spearman's p correlation coefficient greater than 0.5. This suggests that the fact Sonic Pi promotes more prolific code writing will also encourage greater levels of tinkering (it is also shown in the data that users of Sonic Pi have a significantly higher percentage of tinkering to code written) and this, if considered as in Beckwith et al.[4] as a valuable activity in user interaction with code when used constructively and not to the extreme[2], means the interactions recorded within the Sonic Pi environment can be said to be more beneficial to learning than that of Kids Ruby.

[2] Learners exhibiting this problem are referred to as *extreme movers* in this paper.

The data collected from the questionnaires was disappointing, as the responses were so similar as to be uninteresting (The average level of motivation being 4 on a scale of 1 to 5 with standard deviation of 0.43, with that of confidence being 3.92, s.d. 0.66), however on the whole the responses were that the participants were relatively confident and that the fact that they were learning programming in a creative manner increased their motivation. The other form of feedback which was not measured in this study was the questions participants asked about wanting more programming techniques to allow them to achieve what they wanted to create, both visually and aurally. This would have been interesting to have thought of and tried to measure or record, as during the tutorials I observed that many of the more creatively inclined students wanted to and were guessing how to program these more complicated structures.

6 Concluding Remarks

In conclusion, the results indicate that Sonic Pi is the slightly more favourable environment for a novice programmer to learn in as it both encourages more prolific code writing, and tinkering. These in turn promote a better retention of the concepts learnt; and this is shown by the Sonic Pi learners' far better scores in the delayed free recall test. Future experiments might want to measure the difference between Sonic Pi and a non-media-oriented environment, and have a different metric for measuring motivation, as these would be interesting aspects to consider, which were not addressed in this study. It would also be interesting to perform the experiment with a larger group of participants, to explore more subtle correlations, which was not possible within this study.

References

1. Sam Aaron. Sonic pi http://www.cl.cam.ac.uk/projects/raspberrypi/ sonicpi/, 2013.
2. Samuel Aaron and Alan F. Blackwell. From sonic pi to overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 35–46, New York, NY, USA, 2013. ACM.
3. Romina Cachia Anusca Ferrari and Yves Punie. Innovation and Creativity in Education and Training in the EU Member States: Fostering Creative Learning and Supporting Innovative Teaching. Literature review on Innovation and Creativity in E&T in the EU Member States (ICEAC), 2009.
4. Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 231–240. ACM, 2006.
5. Deborah Compeau and Christopher A. Higgins. Computer self-efficacy: Development of a measure and initial test. *MIS Quarterly*, 19(2):189–211, 1995.
6. The Hybrid Group. Kidsruby http://www.kidsruby.com/, 2011-2014.
7. Matthew C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15:1–25, 2005.
8. Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.
9. Seymour Papert. An Evaluative Study of Modern Technology in Education or LOGO Memo No. 26, 1975-6.
10. Loftus G. R. Rundus, D. and R. C. Atkinson. Immediate free recall and three-week delayed recognition. *Journal of Verbal Learning and Verbal Behaviour*, 9:684–688, 1970.
11. Mark Slagell Yukihiro Matsumoto, Gogo Kentaro. Ruby User's Guide translation-http://www.rubyist.net/ slagell/ruby/, 1995.

A KidsRuby Tutorial

In this tutorial, you will learn to program through writing code which makes simple computer graphics in the KidsRuby environment. KidsRuby is a language which is based on the computer programming language Ruby, but has been made simpler for novice learners.

Please complete the various exercises, giving yourself time between each to fiddle around and experiment with the new concepts youve learnt

1. Instructions

Programs operate because they carry out the sequence of instructions they are given in the code which makes up a program.
To set things up for creating computer graphics, KidsRuby needs to be told that this is what the set of instructions are supposed to do. In order to do this, you type the following instructions:

```
 Turtle.start do
       |
end
```

All the code that you type from now on should be inside the first line and the end keywords. The package used to make the graphics is called Turtle graphics, which is why the keyword turtle is needed.

Now, within the do and the end keywords, add the following command:

```
Turtle.start do
          background yellow
end
```

When you now click the [start/play] button to run (or execute) the code youve written, you should see the colour change.

Try swapping or adding to the lines of code with the following:

```
 background black
   pencolor white
 ...blue...green...red\newline
```

2. Syntax and order matter

The order and way you write instructions in the code is very important, like words in any language: if you wrote a set of instructions such as a recipe for someone, the order would be important. The spelling is also important; your program needs to give the computer instructions in a language it understands, and it only understands a small, precisely defined set of words.

Try copying this set of commands:

```
 Turtle.start do
     background yellow
     turnright 90
     forward 50
     turnright 90
```

```
        forward 50
        turnright 90
        forward 50
        turnright 90
        forward 50
end
```

The commands tell the pen what lines to draw on the background: the numbers specify either how many pixels (the unit which your screen resolution is measured in) the length of the line should be, or by how many degrees the direction of where the line is drawn should change.

Try drawing changing the order of some of the commands to see the different shapes you can make, then try misspelling a word or forgetting the space between the numbers and the words, and see what happens.

3. Loops

A useful thing to be able to do is tell computers to carry out a sequence of instructions multiple times. This saves writing out the same commands a lot of times, like in the previous code example. Programmers do a lot of reading of code, and it is neater to say "do something ten times" than to say "do something" followed by "do something" etc. ten times.

Type the following code and run it, it produces exactly the same thing as in the previous example.

```
  Turtle.start do
  background yellow
  4.times do
        turnright 90
    forward 50
  end
end
```

The highlighted part is the part which defines the loop: you need the do and end in much the same way that you need capital letters and full stops, or that you use parentheses, they show the computer the start and the end of the loop, and therefore which instructions to carry out 4 times. Try changing the number 4 to other numbers, and changing the commands within the loop.

4. Random numbers

Computers can generate random numbers, through the following command:

```
  rand(10)
```

This particular command will generate a number which is zero or larger, but less than ten. (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Try putting some random numbers into your loop:

```
  Turtle.start do
  background yellow
  4.times do
```

```
          turnright rand(90)
    forward 50 + rand(10)
  end
end\newline
```

This will result in the degree being some random number under 90, and the length of the lines drawn to vary between 50 and 59 (because each time we are adding 0 to 9 on to 50)

5. Conditionals

Sometimes in a program you will not want all of the instructions to be carried out (executed by the computer) you will only want some of them to be, and only in certain situations. This is handled by conditionals, or if-statements.

An if statement looks like this:

```
if A
    do instruction B
else
    do instruction C
end\newline
```

The A here is the condition which will determine which of either B or C will be carried out (performed)

To make an example using something you just learnt, lets make it random. Type the following code and run it:

```
Turtle.start do
  if rand(10) < 5
        turnright 90
        forward 40
        turnright 90
        forward 40
        turnright 90
        forward 40
        turnright 90
        forward 40
      else
        turnright 90
        forward 50
        turnright 60
        forward 50
        turnright 60
        forward 50
      end
end\newline
```

This program is equivalent to saying: if a randomly generated number between zero and ten is less than 5, then draw a square, if it is not, then draw a triangle

Try changing the program a bit, and make some other conditional statements.

6. Nesting commands

To make more complicated programs, a lot of these commands will have to be executed in different combinations. Putting different commands inside other commands is called nesting commands. For example, a loop inside a loop, or a conditional within a loop. The fact that you have been typing all of your commands within the Turtle.start do . end commands is nesting.

Try typing and running the following:

```
  Turtle.start do
  background yellow
  pencolor red
  8.times do
    forward 50
          if rand(2) < 1
      turnright rand(90)
    else
    turnleft rand(90)
    end
  end
end
```

This makes use of all three more complicated programming concepts covered in this tutorial.

Try experimenting and making more crazy shapes with different combinations.

B Sonic Pi Tutorial

In this tutorial, you will learn to program through writing code which makes simple electronic music in the SonicPi environment. SonicPi is a language which is based on the computer programming language Ruby, but has been made simpler for novice learners.

Please complete the various exercises, giving yourself time between each to fiddle around and experiment with the new concepts youve learnt

1. Instructions

Programs operate because they carry out the sequence of instructions they are given in the code which makes up a program.

Type the following commands into SonicPi:

```
play 50
sleep 1
play 60
```

When you now click the green button to run (or execute) the code youve written, you should hear the noises which correspond to the numbers 50 and 60. the sleep command tells the computer to wait for 1 second between carrying out the next play instruction, otherwise theyll be played really close together.

Try changing the numbers to hear the different notes which you can play, and changing or removing the sleep command.

2. Syntax and order matter

The order and way you write instructions in the code is very important, like words in any language: if you wrote a set of instructions such as a recipe for someone, the order would be important. The spelling is also important; your program needs to give the computer instructions in a language it understands, and it only understands a small, precisely defined set of words.

Try typing in some of the following individually and running the program:

```
pllay 50
play50
sleep
```

Now try this longer set of instructions (or commands)

```
play 50
sleep 0.5
play 51
sleep 0.5
play 50
sleep 0.5
play 51
sleep 0.5
play 50
sleep 0.5
```

```
play 51
sleep 0.5
```

3. Loops

A useful thing to be able to do is tell computers to carry out a sequence of instructions multiple times. This saves writing out the same commands a lot of times, like in the previous code example. Programmers do a lot of reading of code, and it is neater to say do something ten times than to say do something followed by do something etc. ten times.

Type the following code and run it, it produces exactly the same thing as in the previous example.

```
3.times do
   play 50
  sleep 0.5
  play 51
  sleep 0.5
end
```

The highlighted part is the part which defines the loop: you need the do and end in much the same way that you need capital letters and full stops, or that you use parentheses, they show the computer the start and the end of the loop, and therefore which instructions to carry out 3 times.

Try changing the number 3 to other numbers, and changing the commands within the loop.

4. Random numbers

Computers can generate random numbers, through the following command:

```
rand(10)
```

This particular command will generate a number which is zero or larger, but less than ten.
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
Try putting some random numbers into your loop:

```
3.times do
   play 50
  sleep 0.5
  play 50 + rand(10)
  sleep 0.5
  play 50
  sleep 1
end
```

This will result in the first play playing a number which is equal to 50 plus a random number of less than 10.

5. Conditionals

Sometimes in a program you will not want all of the instructions to be carried out (executed by the computer) you will only want some of them to be, and only in certain situations. This is handled by conditionals, or if-statements.

An if statement looks like this:

```
if A
    do instruction B
else
    do instruction C
end
```

The A here is the condition which will determine which of either B or C will be carried out (performed)
To make an example using something you just learnt, lets make it random. Type the following code and run it:

```
if rand(10) < 5
    play 50
    sleep 0.5
    play 50
    play 52
    sleep 0.5
    play 50
  else
    play 60
    sleep 1
    play 55
    sleep 0.1
    play 55
end
```

This program is equivalent to saying: if a randomly generated number between zero and ten is less than 5, then play one sequence of notes, and if not, then play a different one.

Try changing the program a bit, and make some other conditional statements.

6. Nesting commands

To make more complicated programs, a lot of these commands will have to be executed in different combinations. Putting different commands inside other commands is called nesting commands. For example, a loop inside a loop, or a conditional within a loop.

Try typing and running the following:

```
 3.times do
    play 50
   sleep 0.5
if rand(10) < 5
    play 50
    sleep 0.1
       play 50
  else
    play 57
    sleep 0.5
    play 50
  end
  sleep 0.5
```

```
  play 60
  sleep 0.5
  play 50
  sleep 1
end
```

This makes use of all three more complicated programming concepts covered in this tutorial.

Try experimenting and making more crazy sounds with different combinations.