Psychology of Programming Interest Group Annual Conference 2012

London Metropolitan University, UK

21 - 23 November 2012

Proceedings

Edited by

Yanguo Jing

Message from the Chair

Welcome to the Psychology of Programming Interest Group (PPIG) Annual Conference 2012, taking place at London Metropolitan University, London, UK on 21st – 23rd, November 2012.

The Psychology of Programming Interest Group (PPIG) 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 in computational aspects of psychology. The group attracts cognitive scientists, psychologists, computer scientists, software engineers, software developers, HCI people et al., in both universities and industry.

This year, the annual conference involves a wide range of topics. The paper presentations were divided into seven groups spanning across the three days: the learner's mind, personality, AI and knowledge representation, professional expertise, learning to program, tools and their evaluation part 1 and part 2. Authors come from eight countries including Brazil, Finland, Germany, Ireland, Sweden, Switzerland, Uruguay, and the UK.

The conference continues a tradition of hosting a doctoral consortium specifically to allow research students in the relevant disciplines to come together, give presentations and exchange ideas. This year, we are able to accept 11 submissions for this programme.

On the final day, we welcome the keynote from Alan Blackwell from Cambridge University with the title: Extreme notation design – creating a hybrid of Photoshop and Excel. This is followed by a live demo and a panel discussion from Alex McLean, Thor Magnusson, Sam Aaron and Alan Blackwell.

We are grateful to the members of the technical committee, organizing committee, doctoral consortium chair and the reviewers for their hard work preparing this high quality technical program. Finally, we would like to thank the authors and speakers, without whom such a high quality technical program would not be possible at all.

Yanguo Jing

Contents

Organization

Programme Chair and Local Organiser
Yanguo Jing, London Metropolitan University

Organising Committee:
Maria Kutar University of Salford
Thomas Green, University of York
Yanguo Jing, London Metropolitan University

Chair of the Doctoral Consortium
Ben du Boulay, University of Sussex

Technical Committee
Alan Blackwell, University of Cambridge
Alistair Edwards, University of York
Babak Khazei, Sheffield Hallam University
Ben du Boulay, University of Sussex
Chris Douc, Open University
Chris Exton, University of Limerick
Christ Roast, Sheffield Hallam University
Elizabeth Uruchurtu, Sheffield Hallam University
Enda Dunican, Institute of Technology, Carlow
Helen Sharp, Open University
Jasna Kuljis, Brunel University
John Rooksby, St Andrews University
Jorma Sajaniemi, University of Joensuu
Judith Good, University of Sussex
Judith Segal, Open University
Lindsay Marshall, University of Newcastle upon Tyne
Marian Petre, The Open University
Maria Kutar, University of Salford
Markku Tukiainen, University of Joensuu
Sally Fincher, University of Kent
Susan Wiedenbeck, Drexel University
Thomas Green, University of York
Yanguo Jing (London Metropolitan University

Extreme notation design – creating a hybrid of Photoshop and Excel

A decade of collaborating closely with artists inspired two unusual starting points for a programming language research project. The first of these was: what would a programming language look like if it was designed specifically for visual artists? The second was: if a programming language designer were to work in the same manner as an artist, would the resulting product actually work? Fortunately, the answer to these questions turned out to be: 1) rather interesting; and 2) yes. However, although inspired by the arts in both goals and methods, the Palimpsest system is perhaps even more interesting as a case study in language design that has been informed throughout by Psychology of Programming research. In particular, it illustrates the use of attention investment strategies to support first steps in programming, and the distinctive notational choices that result from constraints such as users who prefer not to use a keyboard. This talk will centre around a demonstration of the Palimpsest system, with a discussion of its theoretical motivations and trade-offs from a Psychology of Programming perspective. In response to audience demand, the final part of the talk will either discuss the research methods applied during the implementation of the system, or outline the technical factors involved in implementing systems of this kind.

Alan Blackwell developed his first commercial visual and end-user programming tools in the early 1980s, and has worked with these technologies ever since. This included a mid-career PhD investigating the psychology of visual programming with Thomas Green at the MRC Applied Psychology Unit. Since 1998, Blackwell has been at the Cambridge University Computer Laboratory, where he teaches design-related topics, and directs the Crucible network for research in interdisciplinary design. He and members of his research group have consulted on end-user programming and interdisciplinary design topics for international research consortia and corporations including Microsoft, Google, Hitachi, Intel and Autodesk. Blackwell collaborates widely with social scientists and creative artists, and has published over 100 articles and books on design, visual languages and end-user programming. The Palimpsest system was created during sabbatical leave at the University of Auckland in 2012.

Paper Session 1

The Learner’s Mind

A Study about Students' Knowledge of Inductive Structures

Sylvia da Rosa[1] and Alejandro Chmiel[2]

[1] Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
darosa@fing.edu.uy

[2] Instituto de Filosofía - Facultad de Humanidades y Ciencias de la Educación
Universidad de la República
Montevideo, Uruguay
alejandro.chmiel@gmail.com

Abstract. This article describes two stages of a study carried out with pre-university students, to gather information about the learning of the concept of inductive structures. The study complements two previous investigations focusing on the design of recursive algorithms, from which the study of students' understanding about the input structures of the algorithms arises as a necessity. The theoretical framework used in the three studies is the epistemology of Jean Piaget, specially works about recursive reasoning on the series of natural numbers. Our methodology of research follows principles of Piaget's experiments in which the clinical method from psychiatry was adopted. In this sense, the instructional instance is a tool for obtaining information about cognitive processes. In the first stage, two instructional instances with eight voluntary participants were conducted, in which a problem about an inductively defined set is presented and some questions are posed. The analysis of the responses of the students reveals some difficulties casting doubts on students' conceptual knowledge on the series of natural numbers. Investigating this point is the goal of the second stage where one instructional instance is conducted with seven students, and new information is gathered and analyzed. The results of current and previous studies will be used to elaborate didactic material to introduce inductive definitions, recursive algorithms and proof by induction at pre-university level. This article describes the main theoretical guidelines, the development of both stages of the study and the analysis of the difficulties and progress observed. Some conclusions and future work are included.

Keywords: Data structures, induction-recursion, constructivism.

1 Introduction

In [11] Piaget and colleagues give empirical evidence about the psychogenetic evolution of mental structures[1] corresponding to *reasoning by recurrence* on the series of natural numbers. They show that the source of both calculating on the series of natural numbers (knowledge of algorithms) and inferring properties about its elements (knowledge of proof by induction) is inherent to the construction of the series of natural numbers (knowledge of inductive definitions). Regarding the meaning of induction and recursion in computer science and in mathematics, it can be said that this is a single concept nourished by knowledge on three areas: inductive definitions of structures, recursive algorithms defined on said structures and proofs by induction on the elements of those structures. We use the expression induction-recursion to mean that concept and our motivation follows from tenets of above cited work. In previous works [4, 3] we apply those principles to learn about the relationship between the learning of recursive algorithms on inductive structures different from natural numbers, and students' understanding of said structures. We have found important obstacles on the latter which play a central role in the learning of induction-recursion.

Accordingly to those results the goal of this work is to learn about the construction by the students of the concept of structure isomorphic to natural numbers, that is to say, generated

[1] The term structure has two different meanings in this paper: on the one hand it refers to the mental structures as defined by Piaget (schemes) and on the other hand it refers to inductive structures common in mathematics.

by inductive rules: there are initial elements (base case rule), other elements are generated by the application of constructive functions on previous elements (inductive rule) and these are the unique elements of the structure (clousure rule).

This kind of study is relevant for computer science education research because there is a broad consensus about the relevance of induction-recursion in computer science studies and, at the same time, it is considered both by students and by teachers, hard to learn and teach. Investigations about the learning and teaching of the concept of induction-recursion can be found at least since the 1970s, both in mathematics education and in computer science education based on theories as mental models, phenomenography and constructivism.

The term mental model is used by cognitive psychologists such as Johnson-Laird to define cognitive representations of knowledge. Karl Schwamb in "Mental Models: A Survey"(1990), indicates that mental models are subjects' representations of knowledge about particular situations or phenomena. In the case of learning recursion, several authors refer to mental models to describe the knowledge that the students acquire when introduced to the concept, in most of the cases using some programming language or environment. A mental model is said to be *viable* if it allows the students to accurately and consistently represent the mechanism of recursion and is said to be *non viable* if their representations show misconceptions. On the other hand, a *conceptual model* is designed by the teacher to teach the concept of recursion, while a *mental model* represents the understanding that the learner constructs. Within this theory, several misconceptions about the flow of control or the behavior of recursive procedures and its relationship to iteration are detected and classified in *active*, *step*, *syntactic or magic*, *copies* and *loop* mental models, both for the case of novice and expert students [12, 8, 7, 13].

In [2] the author follows a phenomenographic tradition of research developed in Sweden, based on exploring and describing the cognitive relations between individuals and the world. In the chapter about recursion, she describes that this topic is taught to students using the programming language SML. Then, the students are asked to solve some problems and answer some questions about their solutions and their works are analyzed and classified into three conceptions of recursion: as a construct in SML, as repetition and as self reference.

Constructivist researchers both in mathematics education and in computer science education often refer to the theory of Jean Piaget. For instance in [1] the author says "these concepts come from the seminal work of Jean Piaget" referring to knowledge construction (page 4). In [5] the learning of induction is described by mathematics education researches using Piaget's theory, genetic epistemology. Works like this have influenced our adoption of some general principles of that theory, regarding the construction of concepts and specific explanations about the process of generalizing assimilation [9, 10]. The main ideas are briefly described below.

- The processes involved in the construction of knowledge are associated with mental structures (schemes) which are generated as a result of, and operate with mental structures already formed, for example, in relation to a new problem.
- Instrumental knowledge: It is the knowledge constructed by subjects in the process which is activated when they attempt to solve a problem given by a specification.
- Conceptual knowledge or conceptualization: the construction of conceptual knowledge includes essentially two components: on the one hand, subjects become aware of their coordination of actions as well as the modifications of the objects. On the other hand, facing new problems requires transforming the cognitive resources to take into account variations and similarities. This transformation produces new mental structures (the assimilated concept).
- Comprehension: It is the result of applying conceptual knowledge to solve any problem of appropriate complexity.

The specific explanations taken into account to design this study are due to Cellérier's description in "Piaget Today" [6]. Cellérier explains the reciprocal assimilation of schemes which occurs with a new problem which is assimilated, first of all, to a familiar scheme. The novelty of the problem may or may not be an obstacle in the application (constructive generalization) of the scheme [10]. The obstacle may be represented as a new subproblem, which in turn is assimilated to the subscheme which may solve it, and thus the original scheme may be reapplied (until a new obstacle appears). There are no guarantees as to the solution of all obstacles (subproblems) by a subscheme. The effect of this cognitive strategy (unconscious to the individual) is to combine previous solutions in a new solution with its own function (process), resulting in the synthesis which is the actual core of constructivism. From the perspective of constructivism, structures are coordinated schemes, applicable to a wide range of problems, adapted to cooperate with other schemes within the internal epistemological universe. The concept of number is an example of a structure in this sense.

This article is divided into the following sections: section 2 includes the methodology of research; section 3 includes the development of the stages of the study; section 4 presents conclusions and further work and the list of references follows.

2 Methodology of research

The methodology of research is based on Cellérier's considerations and consists on investigating the pertinence of the following proposition: the knowledge that the students have constructed on the concept of the series of natural numbers can be generalized (in the sense of Cellérier) to a new inductive structure. Therefore, the students should be able to find out the defining rules the latter. To determine to what extent this proposition is correct, two instructional instances were conducted posing a problem about a structure isomorphous to natural numbers, in the sense that there are an initial element and a constructive function of an element to another one (successor).

The subproblems with combined solutions that can provide useful information have been identified as:

1. Identifying the first element.
2. Passing from one specific element to its successor.
3. Passing from a generic element to its successor.
4. Identifying the elements generated in this passage as the only elements of the structure.
5. Identifying the predecessor of a specific element (inverse to what is described in 2).
6. Identifying the predecessor of a generic element (inverse to what is described in 3).

Each subproblem is presented to the students as questions they must respond in writing. The purpose of these questions is to activate, in the students' thought, a generalization and specialization process of previous schemes, corresponding to the solutions of subproblems for the series of natural numbers. If one of the subproblems becomes an obstacle which cannot be assimilated to a subscheme, the students are presented with new questions or reformulated questions in order to direct their cognitive strategy towards overcoming the obstacle. The written answers are compared with the correct ones (provided by the investigators) to assess the difficulty level and to reformulate the question or to formulate a new one. We measure the progress in conceptualization as the distance of the students' answers with the correct answer.

The first and second stages were conducted with 8 and 7 voluntary students of Technical High School in Montevideo, aged 15 to 17. They had no previous formal instruction on induction or recursion which is an advantage to avoid preconceived ideas (often erroneous).

3 Development of the study

The study was conducted in two stages, where the second one arises from the analysis of the information gathered in the first one. In the first stage the students participated of three-hour meetings on two occasions during a period of forty five days. In the first meeting, the inductive definition of an infinite set is given by enumerating some elements and several questions are posed. In the second meeting, questions are reformulated or new questions are posed based on the analysis of the first answers. The goal was to encourage the students to find out the rules of the definition (base, inductive, clousure). The analysis of the information questioned the validity of supposing that the students comprehend the series of natural numbers as an inductive structure. Hence, a second stage of the study was designed and conducted to clarify the point. In the second stage seven students participated in one meeting of one hour and a half.

3.1 First stage

In the first stage, the definition of a set is presented in the problem below.

Problem: *In a faraway planet, there are some living creatures whose DNA is similar to ours, formed by chains of adenine A, thymine T, cytosine C and guanine G. We have little information on these creatures; we only know that they are very primitive, and that they divide in different and separate groups. After many explorations, robots collected DNA information of different groups of extra-terrestrial creatures, and sent it to earth hoping computer science experts would be able to establish the particularities in the DNA of the different groups of creatures from this faraway planet. The data sent by robots are DNA chains of different individuals of the group, a DNA chain being a succession of letters A, T, G, C, such as AGGCGGTAAT. We have received the DNA data of a subset of living creatures defined as:* **Group = {AAAGCTAAA, AGCTA, AAGCTAA, GCT, AAAAGCTAAAA, . . . }**. *This Group could have infinite creatures and thus we cannot indicate them all, but if we observe the known elements (such as the referred elements), we may see that these elements have been constructed or formed in a particular way, starting from an initial element.*

After the problem statement is given, questions from the three main following categories are asked:

- Questions 1 to 4, are related to the rules which define the set. The purpose of these questions is to have the student know that there is an initial element, GCT and that each element is generated from a predecessor by applying an inductive rule (to add an A on each side of the predecessor)
- Questions 5 to 7, are related to application of the rules. The purpose of these questions is, on the one hand, to help students understand the concept of closure, and on the other, to identify the relation between a generic element (different to the first element) and its predecessor (inverse to the inductive rule)
- Questions 8 to 10, related to the properties of the elements of the set. The purpose of these questions is to establish how much students rely on the series' regularity, for their understanding of proofs by induction. This category is included because the concept of induction has two aspects: on the one hand, the inductive definitions of sets and on the other hand the proofs by induction of properties of the elements of the set. As pointed out in [11] the source of reasoning by induction lies in the processes of constructing the inductive structures.

The following is the description of the questions of each category (designed considering the subproblems identified, described in the previous section) and the analysis of the students' answers for which the regulatory criteria, originated in the theoretical framework concepts, were used. Qi stands for question nr i. Recall that the questions of the second meeting were formulated after analyzing the responses of the first meeting.

3.2 Rules that define the set (questions 1 to 4)

This category of questions studies the conceptualization of the students of the initial element and of the relation of one element with the successor.

First meeting
Q1: Which one is the initial element for the Group?
Answer: . . . is the initial element for the Group.
Q2: If you could add only two letters and we provide you with the AGCTA element, how do you reach a new element in the Group set?
Q3: Given a generic element of the Group, is it possible to construct another element from it? In which way?

The purpose of the first question is to ask the students to identify the base case of the inductive structure. Although there is more than one correct answer, we expect that the students write GCT in the dotted points of question 1, because we believe that it is possible to generalize the knowledge of the series of natural numbers to this structure, with the mechanisms described by Cellérier [6], briefly mentioned in section 2. For the same reason, many questions are deliberately imprecise.

However, three students answered that the initial element is A, and the others answered that it is GCT. Regarding question 2, five students generated an element that DOES NOT belong to the Group. Incorrect answers included alteration of the order of letters, for example, stating that ACTAG is a new element constructed from another element. In Q3 it is expected that students who have found the answer of question 2 for a particular case, discover the rule for a generic case and succeed in its formulation. The study shows that this construction is not simple: only two students describe how to generate a new element correctly (adding an equal number of A's on both sides)i and all students used particular cases in their answer to Q3. These answers make us to believe that there is a visual factor involved. On the one hand, in Q1, since the elements of a set are read from left to right and A is the first letter that students read, and on the other hand, in Q2 and Q3, since the ellipsis in the definition of Group leads students to believe that they may generate new elements in any way.

In the **second meeting** Q1 and Q2 are reformulated for each student based on their previous answers and Q3 is repeated.

Second meeting
Q1: Note that ALL must be constructed from that initial element, then, given A as initial element, how do you construct GCT from A? Also note that A is not an element of the set.
Answer: Rule: . . . is the initial element for the Group.
Q2: Does the element you indicated in the answer to question 2 in the first meeting, belong to the Group? Answer again, noting that the new element constructed by adding letters must belong to the Group.
Q3: Given a generic element of the Group, is it possible to construct another element from it? In which way?

Note that those questions force students to pay attention to the form of the visible elements and induces the idea of elements following the same pattern, even if not present.

There is evidence of some progress between the first and second meeting, since, on the one hand the students answered correctly both questions 1 and 2, which shows that the idea of the rules has been introduced, not without difficulties, as revealed by the fact that only one student improved his answer to question 3 although still using a concrete case. Since this question refers to the concept of "generic element" we recall the study of Matalon, summarized in next subsection.

3.3 The generic element

In [11] Benjamin Matalon publishes a chapter entitled *Recherches sur le nombre quelconque*, in which he analyzes the relation between the generic element concept and the reasoning by induction, since such requires to prove that P(n) → P (n+1) for a generic number n and a given property P. Matalon works with the structure of natural numbers, stating that it is necessary to abstract all the particular properties in n, except the property of being a number, that is, an element belonging to the series of natural numbers. Matalon explains that Fermat made his arithmetic demonstrations using a particular number, but taken as a generic number, for example, the number 17. If none of the specific properties of the number 17 are involved in the demonstration, then the demonstration could be considered valid for all numbers. He adds that in geometry, when a property is to be proven and the statement is "given a generic triangle" a particular triangle is drawn, avoiding right triangles, equilateral triangles or isosceles triangles, and not involving particular properties of the triangle in the demonstration of the property. Among other things, Matalon concludes that to construct the concept of the "generic" element, it would be necessary to perform a generic action, that is, the repeated action to generate a generic element[2]. To extend this result, in the second meeting with students, they are asked to fill in a table which introduces repetition of the action that generates new elements from given ones, or which writes elements which are predecessors to given elements, as shown below:

Second meeting
Q4: Complete the table below and rewrite the set Group, ordering elements based on their length (number of letters). Then, fill in the dots:
Rule: Given a generic element . . . of the Group, then . . . is a new element of the Group.

Predecessor	New element
AGCTA	
	AAAAGCTAAAA
GCT	
	AAAGCTAAA
	AαA
α	

The rows left blank are aimed to induce the students in writing their own elements. All students were able to state correctly the rule in Q4 using α as the generic element. The study of the construction of a relation between one element and its successor, specially the evolution of the transmission between the repetition of actions and their iterative results, that is, between the action of the individual and the result which transforms the object, has been studied in [11]. Each action repeated after its predecessor differs from this in its range within the order of succession of actions and at the same time adds one element to the collection formed until the previous iteration. Once the subjects establish coordination between the succession of their actions and their results, a local synthesis specific for these actions is created, between the order of the succession of actions and the growing number of the collection of objects. This extends the construction of the structure with an aspect of recurrence reasoning, in which the most significant generalization is passing from one element to its successor. In this way, Piaget explains the construction of the series of natural numbers, by a synthesis between serialization and inclusion of classes [11]. For this case of construction of the inductive set, question 4 (where in each row of the table an action is performed and elements are ordered from shorter to longer), induces students to establish a similar synthesis, with which they may construct relations between one

[2] We recall that in genetic epistemology, the actions of the subject play a central role in generating knowledge.

element and its successor or between one element and its predecessor. The construction of said relations is the basis of inductive reasoning, both for the definition of inductive structures and recursive algorithms [3] as for proofs by induction.

We end this subsection with the answers used to compare students' responses to questions 1 to 4:
Q1: The initial element of the Group is GCT.
Q2: From AGCTA, AAGCTAA can be generated.
Q3: From a generic element α of the Group, AαA can be generated.
Q4: Group = {GCT, AGCTA, AAGCTAA, AAAGCTAAA, AAAAGCTAAAA, . . . }
Given a generic element α of the Group, then AαA is a new element of the Group.

3.4 Applying rules (questions 5 to 7)

Once students have worked in the construction of answers of questions 1 to 4 giving rise to base and inductive rules, questions 5 to 7 are asked. The purpose is to draw the students' attention to the application of the **rules already defined by themselves**. Q5 is aimed to induce the students to realize the difference between those elements which may be constructed with the rules and those which may not. Q6 is aimed for the student to become aware that every chain different from GCT, has been generated from an element which is its predecessor.

First meeting
Q5: Given the element AGCT, may you construct a new one in the Group by applying your previous statements? Why or why not? Write down some chains which cannot be formed by applying the rules you stated.
Q6: Verify that all the chains from the Group have been constructed by applying the rules you wrote and underline the predecessor element in each case, when applicable.

All students pointed out that GCT (initial element) is a predecessor to all the other elements. This fact, confusing the initial element with the immediate predecessor is revealed as one of the most significant obstacles in the conceptualization of the structure. To help students overcome this obstacle, in the second meeting, after filling in the table (Q4), a new question about the structure of chains is asked (Q7 below) before going further into the predecessor issue by repeating question 6.

Second meeting
Q7: Given that all the chains of the Group have the structure you mentioned in Q1 and Q4, could there be a chain different to the initial one, which does not end with an A?
Could there be a chain with more than one C?
(New Q6): Verify that all the chains in the Group have been constructed by applying the rules you wrote in Q1 and Q4. Indicate, in each case, which is the predecessor element, when applicable. You may use the table.

We see progress of the students as to the first meeting, which would be explained by the use of the table. All students answered Q7 correctly, revealing some progress in conceptualization. However, answers to new Q6 show how hard is the construction of the relationship between an element and its predecessor. One of the students asked: Is GCT α? This means: Is GCT the predecessor of a generic element? This uncertainty of the student indicates an analysis of the relation of the predecessor with the current element. However, this student, and all students, answered that the initial element GCT is the predecessor to each element. This obstacle appears as one of the most important obstacles in the construction of the concept of induction, and thus further research is necessary to learn about its source. For example, which subscheme should this subproblem be assimilated to, and why it is not, and which other subproblems should be previously posed.

We end this subsection with the answers used to compare students' responses to questions 5 to 7:
Q5: No, I cannot because AGCT does not belong to Group.
Examples of chains that cannot be formed with the rules: AGCT, AAGCT, GCTA, AGCTAA.
Q6: Group = {GCT, $\text{A}\underline{\text{GCT}}\text{A}$, $\text{A}\underline{\text{AGCTA}}\text{A}$, $\text{A}\underline{\text{AAGCTAA}}\text{A}$, $\text{A}\underline{\text{AAAGCTAAA}}\text{A}$, . . . }
Q7: No, all the chains different from GCT end with A. No, there cannot be more than one C in every chain.

3.5 Questions regarding properties (8 to 10)

The last group of questions (8 to 10) posed in the second meeting is related to the following property of the elements of the set: all elements have an equal number of A's on the left and on the right. One of the objectives is that students express correctly the statement of the property and the other one is that they express confidence about that all elements meet this property. Both constitute the basis to learn the method of proofs by induction of any property. The questions are the following:

Q8: Fill in the dots:
If a chain of the Group includes n A letters on the left of GCT, then the chain has a total of . . . A letters.
If a chain in the Group includes a total of n A letters then it has . . . A letters on the left of GCT.
Q9: Fill in: Let . . . be a chain of the Group, then said chain has the property . . .
(Write all the properties you believe apply).
Q10: Based on the previous information, could you say that all chains of the Group have the property . . . ?

All students use the variable n to answer correctly question 8. There were some events which prove some progress regarding understanding of the structure:

- All students but one mentioned that the property is to have an equal number of A's on the right and on the left and that the initial element is GCT
- Some students used the symbol α in the second subquestion of question 9. We transcribe one of the answers to highlight the progress made: "Given an α chain, it has the properties of being constructed from GCT and of having an equal number of A letters on both sides"
- All the students believe that every element of the set has the property (affirmative answers to question 10). This proves they rely on the regularity of the series of inductive chains, which for Piaget is a positive event in the study of the series of natural numbers [11].

We end this subsection with the answers used to compare students' responses to questions 8 to 10:
Q8: The total number of A in a chain with n A letters is 2*n. If the total number of A letters is n, there are n/2 A letters on the left of GCT.
Q9: Let α be a chain then either α is GCT or α has equal number of A letters to the left and to the right of GCT.
Q10: All the chains satisfy the property of Q9.

3.6 Summary of results of the first stage

In this section we summarize the answers of students to the questions and the main problems that need further investigation. The answers of the students to the questions reveal fundamentally three main facts:

- In the answers to question 2 of the first meeting most of students included in Group, elements that do not follow the pattern of the visible elements.

- The reformulation of questions of the first meeting posed in the second meeting seems to help the students to correctly answer Q1 and Q4. That means that they succeed in stating the base and inductive rules of the definition of the structure. The correct answers to Q2 and Q7 (second meeting) reveal advances in the conceptualization of the closure rule, as well.
- However, the students did not succeed to overcome the confusion between the initial element of the Group (GCT) and the predecessor of each element: in the answers to Q6 (both meetings) all students pointed out GCT as the predecessor of each element. Our interpretation is that, despite the correct definition of the rules, the concept of inductive structure is not attained, that is to say the corresponding mental scheme has not been constructed.

These considerations have provided insight on some of the problems which need further investigation. To begin with, we recall our proposition which is that knowledge about the series of natural numbers may help in constructing knowledge about other inductive structures, based on Cellérier's work. Our start point is then that the students already have conceptualized the series of natural numbers because they work and succeed on solving problems from early education. Why do they fail in generalization and specialization the schemes of the series of natural numbers to construct knowledge about the structure of the problem posed in this study? One of the possible answers is that our premise is wrong: the students have not constructed *conceptual knowledge* about the series of natural numbers, despite the *instrumental knowledge* that they reveal in solving problems.

This new perspective leads us to pose the following question: do students have similar difficulties if problems are about natural numbers? Depending on the answer, we can obtain information about whether the obstacle is the original scheme or the process of its generalization.

In order to find an answer, we carried out a second stage of this study in which the students were asked to solve two problems with six questions each, during a meeting of an hour and a half. A brief description of the problems is included in next section.

3.7 Second stage of the study

The main goal of the second stage of the study is to learn about whether difficulties similar to those already detected appear when the students work with inductive structures of natural numbers. The base, inductive and closure rules of the definition of two subsets of natural numbers are given. Several questions are posed to encourage students to recognizing the initial elements as the only ones that have no predecessor, to generating new elements from previous ones and to pointing out the predecessor of any element.

The questions are similar in both problems, as shown below, and the students must respond in writing.

Problem 1: Generating consecutive even natural numbers.

A set A of natural numbers is generated by the following rules:

- Rule 1: $4 \in$ A
- Rule 2: if $\alpha \in$ A then $\alpha + 2 \in$ A
- Rule 3: No other numbers are included in A.

Answer the following questions:

- Q1: Write down some elements of A.
- Q2: Is 67 an element of A? Why or why not?
- Q3: If $36 \in$ A, which is the predecessor of 36?
- Q4: How do you find it?
- Q5: If x is an element of A different from 4, which is the predecessor of x?
- Q6: Complete the table below (all the elements belong to A).

Predecessor	New element
6	
	24
58	
	316
	x
x	

– Q7: Is there any element in the set A with no predecessor? Which one?

Problem 2: Generating non-consecutive odd natural numbers.

A set B of natural numbers is generated by the following rules:

– Rule 1: $7 \in$ B
– Rule 2: if $\alpha \in$ B then $2 * \alpha + 1 \in$ B
– Rule 3: No other numbers are included in B.

Answer the following questions:

– Q1: Write down some elements of B.
– Q2: Is 11 an element of B? Why or why not?
– Q3: If $63 \in$ B, which is the predecessor of 63?
– Q4: How do you find it?
– Q5: If x is an element of B different from 7, which is the predecessor of x?
– Q6: Complete the table below (all the elements belong to B).

Predecessor	New element
15	
	63
7	
	255
	x
x	

– Q7: Is there any element in B with no predecessor? Which one?

In the responses the following facts are detected:

– Regarding the initial element (questions Q1, Q6 and Q7).
  - In the answers to the first question of both problems, elements not belonging to the sets were included, for instance, 2 for the case of A and 1, 3 for the case of B (recall that initial elements are 4 and 7 respectively).
  - The empty rows of the table were filled out with wrong elements.
  - One student answered question 7 of problem 1 correctly, while the other responses to that question were wrong. For instance, many answered that the element with no predecessor is 0 or 2 for the case of set A.
– Regarding predecessor of concrete elements (Q2, Q3, Q4, Q6).
  - Question 2 was correctly answered in problem 1, but incorrectly in problem 2. For instance, some students said that 11 belongs to B because 11 = 5 * 2 + 1, without noting that 5 does not belong to B.
  - In question 3 one student answered that the predecessor of 63 is 61 and three students gave not answer at all. Three students gave the correct answer for questions 3 and 4.
– Regarding predecessor of generic elements (Q5, Q6).

- One student answered that 4 is the predecessor of x in problem 1 and gave no answer for this question in problem 2. Two students gave correct answers in both problems, but one of them filled the table (Q6) incorrectly. The remainder of students answered the question just for the first problem.

It was observed that in the second problem several questions have been left unanswered and there were found more errors than in the first. The first problem is simpler than the second one in the sense that the inductive rule involves just the addition while in the second two operations are involved: multiplication and substraction.

We end this subsection with the answers used to compare students' responses to the questions of the problem 1 and of the problem 2:

Table 1. Answers used to compare students' responses

Q	Subset A	Subset B
1	4 6 8 10	7, 15, 31
2	No, all are formed adding 2 to an element, starting in 4	No, because 5 does not belong to B.
3,4	$34 = 36 - 2$	$31 = \frac{63-1}{2}$
5	$x + 2$	$\frac{x-1}{2}$
6	table of elements of A (see below)	table of elements of B (see below)
7	4	7

Table 2. Tables of Q6 for subsets A and B

Predecessor	New element	Predecessor	New element
6	8	15	31
22	24	31	63
58	60	7	15
314	316	127	255
10	12	255	511
$x - 2$	x	$\frac{x-1}{2}$	x
x	$x + 2$	x	$2 * x + 1$

4 Conclusions and Further Work

There follows a classification of some of the types of errors appearing during the first stage of the study and a summary of them related the regulatory criteria from the theory (table below).

- Type 1: There is an error influenced by a visual factor (defining A as the initial element)
- Type 2: There are two types of type 2 errors: a) given an element of the Group, construct another one which IS NOT part of the Group (for example from AGCTA to AAGCT) and b) pass from an element which is NOT from the Group to a new element (for example from AGCT to AGCTA). (In the table below, x and x + 1 are used to denote an element and its successor respectively.)
- Type 3: Using particular cases for the generic element
- Type 4: Confusing the predecessor with the initial element
- Type 5: An influence of preconceived ideas (confusing property with element, confusing language with metalanguage)

Table 3. Summary of types of errors of the first stage

Question	Objective	Regulatory criteria	Errors
1 to 4	Identifying the first element	Local synthesis/serialization	Type 1
1 to 4	Passing from x to x+1	Local synthesis/serialization	Type 2
1 to 4	Clousure	Regularity of the series	Type 2
1 to 4	The generic element	Repetition of the action	Type 3
5 to 7	Application of the rules	Constructive generalization	Type 2
5 to 7	Passing from x+1 to x	Local synthesis/serialization	Type 4
8 to 10	Thinking properties	Regularity of the series	Type 5

The different types of errors are related. In general terms, it could be said that one type of error leads to other types. To set an example, students who cannot understand the relation between one element and its successor (or predecessor), add to the set, elements which do not belong there. Confusing the initial element of the structure and the predecessor to a generic element is an obstacle both for the proof by induction – since one element (different to the initial element) satisfies a specific property because the predecessor does – and for the definition of recursive algorithms – where the result for each element is constructed using the result for the previous element–.

Although the information gathered in the second stage has to be more deeply analyzed, it can be said that the same types of errors have been detected in both stages of the study. We believe that the facts pointed out in previous section, show evidence that the obstacles partially lie in students' lack of *conceptual knowledge* of the series of natural numbers as an inductive structure, despite they are at pre-university level. We believe that this affects the learning on induction-recursion and that it is necessary to help students to construct *conceptual knowledge* on the series of natural numbers from their *instrumental knowledge*. The objectives of our next study shall focus on that issue.

References

1. Mordechai Ben-Ari. Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching, Vol. 20, Issue. 1, 2001, pp. 45-73*, 2001.
2. Shirley Booth. Learning to program - a phenomenographic perspective. *Gteborg Studies in Educational Sciences*, 1992.
3. Sylvia da Rosa. The learning of recursive algorithms from a psychogenetic perspective. *Proceedings of the 19th Annual Psychology of Programming Interest Group Workshop, Joensuu, Finland*, pages 201–215, 2007.
4. Sylvia da Rosa. The construction of the concept of binary search algorithm. *Proceedings of the 22th Annual Psychology of Programming Interest Group Workshop, Madrid, Spain*, pages 100–111, 2010.
5. Ed Dubinsky and G. Lewin. Reflective Abstraction and Mathematics Education: The Genetic Decomposition of Induction. *Advanced Mathematical Thinking*, 1986.
6. Guy Cellérier et al. *Structures and Functions in Piaget today.* Lawrence Erlbaum Associates Publishers, 1987.
7. Hank Kahney. What do Novice Programmers Know about Recursion. *ACM 0-89791-121-0/23/012/0235*, 1983.
8. Claudius M.Kessler and John R.Anderson. Learning Flow of Control: Recursive and Iterative Procedures. *Human-Computer Interaction, 1986, Volume 2, pp 135-166, Lawrence Erlbaum Associates, Inc.*, 1986.
9. Jean Piaget. *La Prise de Conscience.* Presses Universitaires de France, 1964.
10. Jean Piaget. *Recherches sur la Généralisation.* Presses Universitaires de France, 1978.
11. Jean Piaget and coll. *La Formation des Raisonnements Recurrentiels.* Presses Universitaires de France, 1963.
12. Vashti Galpin Tina Götschi, Ian Sanders. Mental models of recursion. *ACM 1-58113-648-X/03/0002*, 2003.
13. Susan Wiedenbeck. Learning Recursion As a concept and As a programming Technique. *ACM 0-89791-256-X/88/0002/0275*, 1988.

Gaze Evidence for Different Activities in Program Understanding

Kshitij Sharma, Patrick Jermann, Marc-Antoine Nüssli, Pierre Dillenbourg

CRAFT, École Polytechnique Fédérale de Lausanne
<firstname>.<lastname>@epfl.ch

Abstract We present an empirical study that illustrates the potential of dual eye-tracking to detect successful understanding and social processes during pair-programming. The gaze of forty pairs of programmers was recorded during a program understanding task. An analysis of the gaze transitions between structural elements of the code, declarations of identifiers and expressions shows that pairs with better understanding do less systematic execution of the code and more "tracing" of the data flow by alternating between identifiers and expressions. Interaction consists of moments where partners' attention converges on the same same part of the code and moments where it diverges. Moments of convergence are accompanied by more systematic execution of the code and less transitions among identifiers and expressions.

1 Introduction

In the last decade, off the shelf screen based remote eye-trackers have become readily available. These devices offer system designers and social scientists unprecedented access to the users' attention. Our long term goal is to use automatically collected gaze traces to provide feedback to the users and adapt the system to their level of expertise. A prerequisite for such undertakings is that we better understand the relationships between gaze-based behavioural indicators and task-based performance.

In this contribution we propose an analysis of pair-programming that illustrates the sensitivity of gaze traces to different levels of understanding as well as to different modes of interaction. This problematic is a two sided coin: it involves cognitive aspects related to program understanding and social aspects related to the interaction of two programmers.

Program understanding is central in many programming tasks, for example during software maintenance or software evolution where programmers have to read and extend code that they did not necessarily produce themselves. Program comprehension is a goal-oriented, problem-solving task that is driven by preexisting notions about the functionality of the given code [10]. It can be thought of a pattern matching at different levels of abstraction [20]. The different abstraction levels help understanding a program at different levels, for example, at syntactical level programmers can understand the relation between different programming constructs and at semantic level they can relate different programming structures to their real world counterparts. The potential of eye-tracking in diagnosing the quality or the strategies of understanding relies on the assuption that understanding strategies are reflected by different ways to "read" the code.

From the point of view of the interaction between pair programmers, we are interested in finding the effect whether different types of collaboration strategies are reflected by gaze indicators and whether they are linked to a program comprehension strategy. Previous experiments carried out using cross-recurrence [17] as a measure of gaze coupling showed that the higher the coupling, the better the outcomes of collaboration. This might not be true for more specific problem solving tasks. For example, in collaborative program understanding the task requires the participants in collaborative environment to develop their own understanding as well as to agree upon a solution. During an extended pair programming session, programmers do not necessarily attend to the same information at all times. In terms of interaction, this translates to the requirement for different phases in the interaction. During convergent phases, they look at the same part of program. During divergent phases, participants look at different parts of the program. Both of them can contribute to understanding a program in different ways. These phases might reflect different understanding strategies in program comprehension, for example individual hypothesis building and collaborative verification.

In next section we present related work about program understanding and dual eye-tracking for pair programming. We then describe the problem statement and research questions. The Methods section gives the details of the experiment and the algorithm that was used to find different phases of interaction. We then present and discuss the results from the experiment.

2 Related Work

2.1 Program Understanding

Program understanding is a special kind of problem solving. Like any problem solving task, program comprehension has a problem statement (to understand the given program) and a solution (description of functionality of the program) and different approaches to get the solution. The main approaches include top-down and bottom-up. Top-down approach involves decomposition of the problem in sub-problems and solving the sub-problems, while bottom-up approach involves integration of low-level details to come up with a solution.

J.Larkin [11] found in her studies about solving physics problems two approaches of problem solving; first, to approach the solutions from the problem givens(top-down) and second, to backtrack from the solution to the problem givens(bottom-up). Similarly, J.R. Anderson [1] found in his studies two ways to write programs; first, writing code line by line(bottom-up) and second decomposing the goal of the program into subgoals and implementing the subgoals(top-down).

In the case of program understanding, there are many strategies to understand a program, a top down approach [19] consists of starting with a hypothesis about the program and then validating or “end marking” the hypothesis with the individual components of the program. A Bottom-up approach [18] starts from a series of code fragmentation and then assigns a domain concept to each fragment. An Iterative approach [4] includes a ”while” loop of top-down process, i.e., having a set of preexisting notions or hypothesis, their verification and modification, until everything in the program can be explained within the set of notions with which the iteration started. There are some more strategies that are a hybridisation of top-down and bottom-up [13] [14]. These two strategies are used interchangeably during program comprehension as and when needed [13].

S. Letovsky [13] proposed a typical set of mental models needed to understand a program which includes specification, implementation and annotation of different parts of the program. [13] also emphasised that mental model for implementation consists of actions and data structures of a program. Understanding the entities/data/variables and relationship amongst them inside a program is very important in order to assign them a concept from the domain knowledge [21]. [8] advocates for having a programming plan to understand the program text (what is written?) and the program intent (why is something written?), and then divides the programming plan into two major parts “Variable plan” and “Control plan”. [8] then proposes the use of variable plan to understand the relation between program text and program intent.

In two different studies [3] and [10] describe the particular strategies for novice and expert programmers respectively. On one hand, [3] finds that for the understanding of novices while loops sometimes become ”while demons”. Novices have ”conflicts” in the strategies to be applied for giving the ”Natural Language Description” of a program. Novices tend to follow the ”systematic execution” of the program and increase their chances to get stuck. On the other hand, [10] finds that experts go for ”as-needed” strategy, where they limit their understanding to only those parts of the program that they find relevant to a given task. Experts do not follow a predefined strategy to understand a program. For example, experts do not decide beforehand to understand a program in ”top-down” or ”bottom-up” manner. Experts tend to use both of them as and when needed.

2.2 Dual Eye Tracking for Pair Programming

Two synchronous eye-trackers can be used for studying the gaze of two persons interacting to solve a problem. It gives a chance to understand the underlying cognition and social dynamics when people collaborate. Nüssli [15] gives a two way motivation for dual eye-tracking. Using statistics to find the relation between the gaze features and collaboration events and using machine learning for prediction of some collaboration attributes from gaze patterns. In their study of collaboration amongst a pair Richardson and Dale [5] found that when two persons talk about something they see, they tend to look at the same thing in the stimulus. [17] measured the ”togetherness” of the participants in a ”speaker-listener” pair using cross-recurrence plots and found that when the listener follows the gaze of the speaker (s)he had a better comprehension. This idea of ”looking together” may not be true to a pair of programmers looking at a program and trying to understand it because the task of program understanding is more specific than the task of listening to a speaker. ”Togetherness” of their eye-movements can help to define different phases of interaction the pair undergoes during the task of program comprehension. These different phases occur when the participants are looking at the same part of the program as opposed to the case when they are not. Both types of the phases play there role in the understanding of a program as we mentioned in the introduction.

Pair programming is usually done with co-located programmers. They play the roles of driver (actual typing) and navigator (more like a organisational activities). Spatially distributed pair programming have been studied with satisfactory results showing that the distance factor can be neglected [2]. Pair programming leads to high quality programs [15], hence a pair of expert programmers can obtain a better understanding of a program as well.

3 Problem Statement

3.1 Program Understanding and Gaze Transitions

Is it possible to detect different strategies for program understanding between the pairs with perfect versus low levels of understanding ? Do they build their understanding based on different semantic elements in the program than the pairs with the low level of understanding? There are many ways to go about solving the problem of program understanding as we mentioned in the related work. We also mentioned that program understanding strategies are different for the people who have better understanding than others who don’t. We are interested in finding this difference in terms of their eye-movements.

To measure exploration strategies, we adopt an approach based on gaze transitions between different types of program elements. More precisely, as a stimulus for eye-tracking, a program can be divided in three main semantic classes (or Areas of Interest). These semantic classes are identifiers (I), structural (S) elements and expressions (E) in the program. Typically, identifiers are the variable declarations, structural elements are the control conditions of the program and expressions represent the data flow and relationship amongst the variables. Thus we propose to say that a ”to and fro” shift in gaze between identifiers and expressions will depict the attempt to understand the data flow and/or the relation among the variables. Similarly, a gaze shift among all the three semantic classes will translate in an effort to understand the data flow according to the conditions in the program. In terms of program understanding strategies this behaviour is attributed to ”Systematic program execution”.

Our analysis aims at finding which type of transitions characterise pairs with different levels of understanding. Table 1 shows the categorisation of different transitions among different semantic classes in the program into data flow, control flow and data flow according to control flow. We consider ”3-way” transitions among the semantic classes as one 3-way transition reflects one unit of program understanding behaviour. For example, a 3-way transition ”E− >I− >E” reflects the ”reference lookup” for a variable in an expression.

Question 1 Is there a relation between the transitions among different semantic classes in the program and the levels of program understanding ?

3.2 Convergent and Divergent Episodes of Interaction

Collaboration consists of a series of convergent and divergent phases. When partners work as a team and put their joint efforts to understand the code we say that they are convergent in their interaction and we have a convergent episode of interaction. In a convergent episode participants in a pair look at the same part of the program in a ”stable” manner. ”Stable” manner of looking at a program is reflected by fixations in a small range (less than a threshold) of tokens (see section ”segmentation of eye-tracking data” for more details). On the other hand when the participants try to build their own understanding and they are looking at the different parts of the program, we say that they are diverging and we have a divergent episode of interaction.

Table 1: Categorization of different transitions among different semantic classes in the program into different types of flows in the program. (I=Identifier, S=Structural, E=Expression). $->$ denotes transition.

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

The basic question related to episodes is whether individuals use different reading strategies during convergent and divergent episodes. A complementary question is whether pairs with different levels of understanding behave differently in convergent and divergent episodes. To define convergent and divergent episodes of interaction, we need to segment the eye-tracking data for individual participants and then align them in time; so that we can compare the segments of the two participants on the scale of vicinity in terms of fixations.

Question 2 How do convergent and divergent phases of interaction affect the program comprehension strategies of pairs with different levels of understanding?

3.3 Segmentation of interaction

The segmentation of interaction into segments or ”meta-fixations” during which attention is focused on a stable set of objects is a novel approach in gaze analysis. Usually, fixation time is aggregated in predefined areas of interest and researchers report global proportions of attention time dedicated to the different types areas. To measure coupling, cross-recurrence analysis quantifies as a global measure how much the gaze of the collaborators follow each other with a given lag. These fixation based measures aggregate indicators measured in the 100ms range to the whole duration of the interaction. The segments that we propose to detect on the other hand are situated in between the short time range of a fixation and the long time span of the whole interaction. Figure 1 shows the conceptual difference between the fixations and segments. The main difference is in their respective durations in time and their use to analyse different types of behaviours.

We considered different time series segmentation algorithms [16] [9] [22] to implement a method that would segment the date into meta-fixation but none of these methods carried the notion of meta-fixations or a hierarchy of segments in terms of their duration. Hence, we computed the segments from the fixations using a very simple procedure. This procedure computes segments from the fixations in a similar way as fixations are computed from the raw gaze data. Moreover, none of the methods for segmentation or change point detection describe method for finding the segmentation on two simultaneous time series and to align the segments in terms of time. We give details of this procedure in next section.

4 Methods and Materials

4.1 Experiment

In the experiment, pairs of subjects had to solve two types of pair programming tasks. The first task was to describe the rules of a game (e.g., initial situation, valid moves, winning conditions, and other rules) implemented as a Java program. The only hint to the pairs was that it is a turn based arithmetic game. The second task was to find errors in the game implementation and to suggest a possible fix using a few lines of output to analyse the error and to find the location of it in the code.

Subjects Eighty-two students from the departments of computer science and communication science of the École Polytechnique Fédérale de Lausanne, Switzerland were recruited to participate in the study.

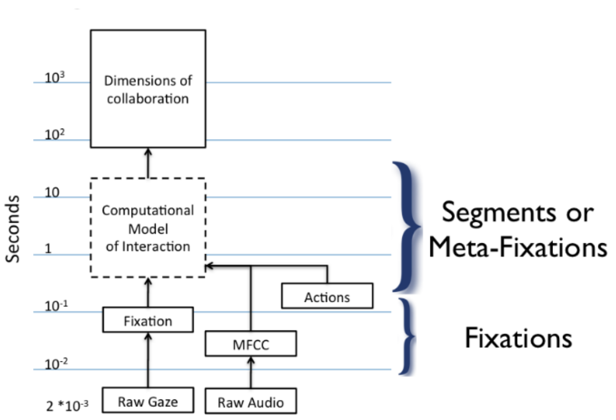


Figure 1: A typical Diagram to show the conceptual analogy between the fixations and the segments, and to show the analogy between different levels of raw gaze aggregation and the behaviour dimensions

They were each paid an equivalent of 20 USD for their participation in the study. The participants were typical bachelor and master students. The participants were paired into forty pairs irrespective of their level of expertise, gender, age or familiarity.

Procedure Subjects had to read and sign a participation agreement form, when they came to laboratory. Then, for the next 3 minutes, the experimenters calibrated the eye-trackers for each of the subjects. This simple procedure consists of fixating the centre of nine circles appearing on the screen. Once both subjects were ready, they individually filled a short electronic questionnaire about their programming skills and previous experience. The pretest which followed, consisted of individually answering thirteen short programming multiple choice questions.

Apparatus and Material Gaze was recorded with two synchronised Tobii 1750 eye-trackers that record the position of gaze at 50Hz in screen coordinates. The eye-trackers were placed back to back and separated from each other by a wooden screen. The synchronisation of the eye-trackers was done by using a dedicated server to log gaze via callback functions from the low-level API of the eye-trackers. The subjects heads were held still with an opthalmologic chin-rest placed at 65 centimetres of the screen. An adaptive algorithm was used to identify fixations and a post-calibration was done to correct for systematic offsets of the fixations with regards to the stimulus (see [10] for details about these procedures).

The JAVA programs were presented in a custom programming editor based on the Eclipse development environment. Text was slightly larger (18pt) than it is usually on computer screens and was spaced at 1.5 lines to facilitate the fixation hit detection at a word level precision. Scrolling was synchronised between the participants, such that when programmers scrolled, their partners' viewport was also updated at the same time. All other highlighting, search and navigation functionalities were disabled in the editor.

Level of Understanding We distinguish between three levels of understanding based on how well they performed the description task.

Good Pairs with a good understanding are able to describe the the rules of the game (initial situation, valid moves, winning conditions).

Medium Pairs with a medium level of understanding only describe partial aspects of the game structure, and often give algorithmic descriptions of the program and try to guess the detailed rules from the method names; but they failed to get the winning condition.

Poor Pairs with a poor understanding are not able to describe the functionality of the code. During analysis we saw that the pairs with poor level of understanding were only those pairs which had novices as both the participant and their gaze pattern was as good as a reading plain text and not a program.

Moreover, they could explain only some of the the syntactical things properly (e.g., they say that there is a while loop that runs until some condition but could not explain what it is doing), so we decided to analyse with only two levels of understanding.

Tokens, Semantic Classes and Transitions The program is comprised of tokens. For example, a line of code ”int i = 5;” contains 8 tokens (int, i, =, 5, ;, and 3 spaces). We recorded the time spent on the various tokens in the program and categorised them into three semantic classes:

Identifier this class includes the variable declarations.

Structural this class includes the control statements.

Expression this class includes the main part of the program, like the assignments, equations, etc.

We took the change of gaze from one semantic class to a different class as the transition between the semantic classes. Here, we do not consider the change of gaze position from one token of a particular semantic class to another token of the same class.

4.2 Segmentation of Eye-Tracking Data

In this section we present the approach to aggregate the fixations into the segments. The existence of segments first came to our attention when looking at the evolution in time of the JAVA tokens looked at by the programmers during a program understanding task. The curve in the figure 2 represents the evolution of the average token identifier in time (tokens were numbered in order of appearance in the code), for a particular pair. Stable exploration episodes clearly appear as ”plateaux” separated by ”valleys” and are reminiscent of the data patterns that characterise the organisation of raw gaze data into fixations and saccades. Deep valleys are due to programmers scrolling through the code while looking for particular methods whereas smaller valleys correspond to focus shifts between areas of code visible on one screen. Segmenting the fixation data into interaction episodes is a two step process; first we find the segments for individual participant in the pair and then we align them in time to find the episodes of interaction for the pair.

Finding Segments in the Gaze of Individual Participants For finding the segments from the individual fixation data, first of all we smooth the fixations using moving averages; and then used the following steps to find the segments from the individual fixation data:

1. Divide the smoothened fixation data into non-overlapping windows.
2. For fixations in each window find a best fitting line.
3. For each fitted line find the angle it makes with the time axis and for each window find the range of tokens looked at by the participant.
4. For each window find whether the angle between the line and the time axis and the range of tokens looked at are both less than the respective thresholds; if yes, then the window is deemed to be a part of a segment.
5. once we have the potential portions of a segment; we merge such windows that are consecutive in time, only if they are overlapping in terms of the range of tokens looked at.
6. The output of this step is the segments or the merged windows for both of the participants in a pair.

Figure 2 shows the segments computed from the fixation data (sampling rate 5Hz) for two participants in the same pair. The black lines depict the segments. These individual segments are used to define a set of different episodes of interaction during the whole interaction, we describe this step in next section.

Temporally Aligning the Segments for the Pair Using the segments for both the participants we align them in time and then again merge the segments so that we have longer (in terms of time) episodes of interaction to analyse.

For finding the episodes of interaction we use the following steps:

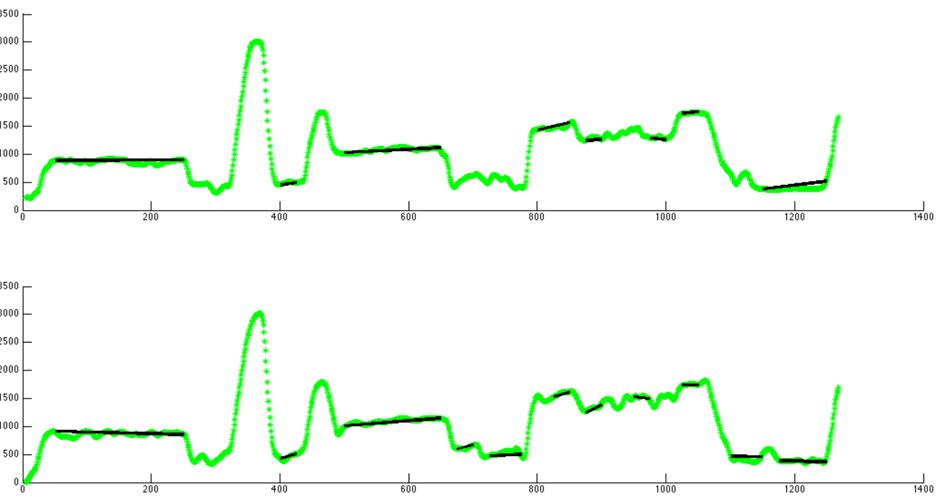


Figure 2: Segments computed for individual participants of a pair in the program understanding task. The x axis represents time (sampling rate 5Hz). The y axis represents the average token ID that was gazed at. A horizontal ”plateau” (black horizontal lines) means that the subject has been looking at a stable range of tokens.

1. Input to this step is the segments for both the individuals in a pair that we get as the output of the previous step.
2. For each segment of one participant find the temporal overlap of this segment with each of the segments of the second participant and make a binary overlap matrix where each element indicating whether the i^{th} segment of first participant overlaps (more than a threshold) with the j^{th} segment of the second participant,in terms of time and the range of tokens looked at (intuitively we can say that there is no temporal overlap between the non-consecutive segments).
3. Once we have the overlap matrix, we take the intersection of the segments for the two participants (in terms of their duration) and define the intersection to be the convergent episodes of interaction.
4. The output of this step is the set of convergent episodes of interaction for a pair.

Figure 3 shows an example of temporal alignment of the segments for the two participants in a pair and the episodes of interaction in terms of time. The episodes of interaction are then used to form a Contingency Table (see section 5.1) which is used to analyse the gaze inside an episode and overall interaction. We give details about the contingency tables and the analysis in next section.

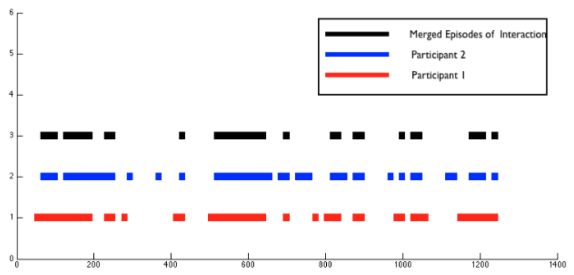


Figure 3: Segments of both the participants aligned in time and the episodes of interaction; time on X-axis; Y-axis: 1 for first participant, 2 for the second participant, 3 for the episodes of interaction

4.3 Data Preparation for Analysis

In this section we describe the pathway from raw gaze data to the contingency tables of transition between the semantic classes.

Raw Gaze and Fixations Raw data from eye-trackers come at a high sampling rate that is well above (typically at 50Hz and higher rates) the rate of gaze fixations. Hence, the first step in the analysis of gaze aggregates the gaze points given by the eye tracker into fixations (moments of relatively stable gaze positions).

Determining Areas of Interest : Tokens Once we have the fixations from the raw gaze data we define the areas of interest in our stimulus i.e., in the program.

Episodes of Interaction From the fixations we get the episodes of interaction using method described in section "segmentation of eye-tracking data".

Tokens to Semantic Classes After defining the tokens as our areas of interest we categorised them in 3 categories Identifiers, Structural and Expressions (see section "program understanding and gaze transitions" for details).

Sequence of Semantic Classes looked at We took the sequence (time series) of the Semantic classes fixated during the interaction for our analysis (we took the data inside and outside of the "segments" while analysing convergent and divergent episodes respectively), for example sequence "IIIESSEESSSIIIE" (I = Identifiers, S = Structural and E = Expressions) tells us that first 3 fixations were on identifiers, 4^{th} fixation was on an expression then next 2 fixations were on the structural elements and so on.

Lumping of Sequence As we are interested in the transitions between the semantic classes and not in the duration of time spent on the different semantic classes. We took the continuous fixations on the same semantic class to be one fixation and thus the above example sequence turned into a "lumped" sequence as "IESESIE".

Lumped Sequence to "3 way" Transitions Once we had the lumped sequence we simply counted the number of transitions from one semantic class to other and then to another one. For example the lumped sequence "IESESIE" has 5 transitions "IES", "ESE", "SES", "ESI" and "SIE".

Transitions to Control Flow Transitions "ISI" and "SIS" depict the activity of tracing the control of the program with the different states of the variables.

Transitions to Data Flow Transitions "IEI" and "EIE" depict the activity of tracing the data flow of the program. This reflect the task of looking for different variables and their interdependencies.

Transitions to Systematic Program Execution All the transitions involving the three semantic classes and the transitions "ESE" and "SES" reflect gaze transition amongst all the semantic elements in a program. This translates to the task of considering the modification of an entity as per the control flow of a program.

5 Results

5.1 Question 1

We return to our main interest of finding the difference between gaze transitions for the pairs with different levels of understanding. We first report a relation between the level of understanding of the pair, the pair composition and the gaze transitions using log linear models [6]. Log linear models use contingency tables [12] to find the relation between different variables and for comparing the two models for same contingency table [6] used a new statistics, called G^2 the "likelihood statistics" (or LRX^2), which is asymptotic to "chi square". G^2 can be calculated as following:

$$G^2 = 2\sum_i (observed)_i log \frac{(observed)_i}{(expected)_i}$$

There are two main methods for fitting the log linear model to a given contingency table. "Forward Selection", where we fit all hierarchical models that include the current model and differ it by one effect; and "Backward Elimination" leaves the term that incurs the least change in the LRX^2 value (for details see [6]). We combined both of the methods to achieve a fast consensus. According to the forward selection we fit all the hierarchical models that differ the current model by one term; and for the next iteration we keep the model with least change in the LRX^2 value (opposite to the backward elimination, but the idea is to delete the least change incurring term). The finally selected must have the maximum degrees of freedom with the least change in the "likelihood statistics" (or LRX^2).

Table 2: Hierarchical linear model fitting for Contingency Table with dimensions Transition (T), Pair Type (P) and Level of Understanding (UND), for the combined gaze of all the pairs

Model	G^2	DoF	Terms Deleted	$\triangle G^2$	$\triangle DoF$
$[T][P][U]$	7503	57			
$[TPU]$	0	0			
$[TP][TU][PU]$	32	22	$[TPU]$	32	22
$[TP][TU]$	7267	24	$[PU]$	7235	2
$[TP][PU]$	103	33	$[TU]$	71	11
$[TU][PU]$	54	44	$[TP]$	22	22
$[TU]$	8026	48	$[PU]$	7972	4
$[PU]$	8487	66	$[TP]$	8433	22

Table 2 shows the log linear model fitting using the method proposed above. The first 2 models $[T][P][U]$ and $[TPU]$ are the "independence model" and the "saturated model" respectively. We can see that the saturated model fits the data perfectly ($DoF = 0$, $G^2 = 0$). On the other hand, independence model shows a big variation ($DoF = 7503$, $G^2 = 57$) from saturated model. Removing the 3-way interaction term results in the model $[TP][TU][PU]$ ($DoF = 32$, $G^2 = 22$). Now, we see the effect of removing one 2-way interaction term at a time. Removing term $[PU]$ causes a big deflection from the all 2-way terms model with a small increase in the degrees of freedom ($\triangle G^2 = 7235$, $\triangle DoF = 2$). Removing $[TU]$ also causes some deflection from the all 2-way terms model ($\triangle G^2 = 71$, $\triangle DoF = 11$); but removing $[TP]$ term causes the smallest deflection and increases the degrees of freedom as well ($\triangle G^2 = 22$, $\triangle DoF = 22$). Further removing terms from $[TU][PU]$ causes greater deflections. The best fit model for a given contingency table is the one with the least $\triangle G^2$ and largest $\triangle DoF$ with respect to the saturated model ($[TU][PU]$ in this case with $\triangle G^2 = 22$ and $\triangle DoF = 22$).

It is clear from finally selected log linear model that there is a dependence between "Transitions" and the "Level of Understanding" as well as between the "Pair Type" and "Level of Understanding".The first dependency is reflected by the term $[TU]$ and the later one is depicted by the term $[PU]$. Where $[PU]$ reflects the fact that a pair of two experts can understand a program better than a pair of two novices. To better understand the dependency between transitions and levels of understanding we use ANOVA. Here, instead of using the transitions, we grouped them in categories as depicted in table 1. Figure 4 shows the differences between the two levels of understanding (medium and high) for the different types of flows in a program.

Table 3: Summary of results of ANOVA for the transitions from the whole interaction

Transition Type	$\mu_{UND=1}(\sigma_{UND=1})$	$\mu_{UND=2}(\sigma_{UND=2})$	F[1,28]	p
Data Flow	0.4 (0.018)	0.45 (0.016)	65.5	$< .01$
Systemaatic Execution	0.55 (0.020)	0.51 (0.017)	32.1	$< .01$

Pairs with medium level of understanding have relatively more transitions amongst all three semantic classes.In terms of gaze transitions this behaviour translates to reading each line of the program and trying to understand it. This shows that these pairs look simultaneously at the conditions in the program as well as the modification of the data elements according to them. They try to understand the data flow in accordance to the control flow of the program. This attempt of program understanding is similar to the "Systematic execution of program". This method is not very characteristic of the pairs with high level of understanding, as shown in an experiment by [10]. The pairs with high level of understanding have relatively more transitions among the identifiers and the expressions. They concentrate more on

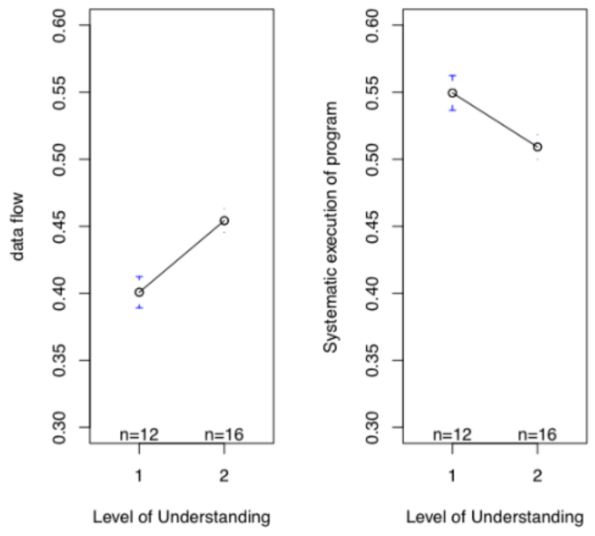


Figure 4: Mean plots for different transitions for the whole interaction (Level of Understanding $1 = Medium$ and $2 = High$)

the variable/entities and the relationship among them. Building up their understanding in this manner the pairs with the higher level of understanding are able to do a proper concept assignment from the program domain to the world domain [21].

5.2 Question 2

Taking our analysis one step ahead, to find the effect of the convergent and divergent episodes of interaction, we carried out 2×2 ANOVA for data flow and data flow according to control flow with two factors level of understanding and convergent/divergent interaction episodes.

Table 4 shows the descriptive statistics for the proportion of data flow transitions in different types of interaction episodes and for different levels of understanding. There were two single effects for the type of interaction episode ($F[2, 28] = 121, p < 0.01$) and for the levels of understanding ($F[2, 28] = 10.86, p < 0.01$), and there was no interaction effect. From figure 5 we see that all the pairs in divergent phases of interaction spend more time on understanding the data flow than that in convergent phases. The effect of the level of understanding on data flow is visible by the fact that the pairs with high level of understanding follow the data flow more than the pairs with medium level of understanding.

Table 4 shows the descriptive statistics for the proportion of systematic execution episodes in different types of interaction episodes and for different levels of understanding. There were two single effects of the type of interaction episode ($F[2, 28] = 106, p < 0.01$) and of the levels of understanding ($F[2, 28] = 8.36, p < 0.01$), and there was no interaction effect. From figure 5 we see that the pairs in convergent phases have a high ratio of transitions that correspond to systematic program execution. There is also an effect of levels of understanding on systematic program execution depicting more effort put by the pairs with medium level of understanding on systematic program execution.

Table 4: Proportions of data flow and systematic execution transitions (mean and standard deviation) by type of episode and level of understanding.

Transition Type	Episode Type	Understanding	
		Low	High
Data Flow	Convergent	0.59 (0.04)	0.56 (0.03)
	Divergent	0.51 (0.02)	0.49 (0.02)
Systematic Execution	Convergent	0.35 (0.04)	0.38 (0.03)
	Divergent	0.44 (0.02)	0.47 (0.02)

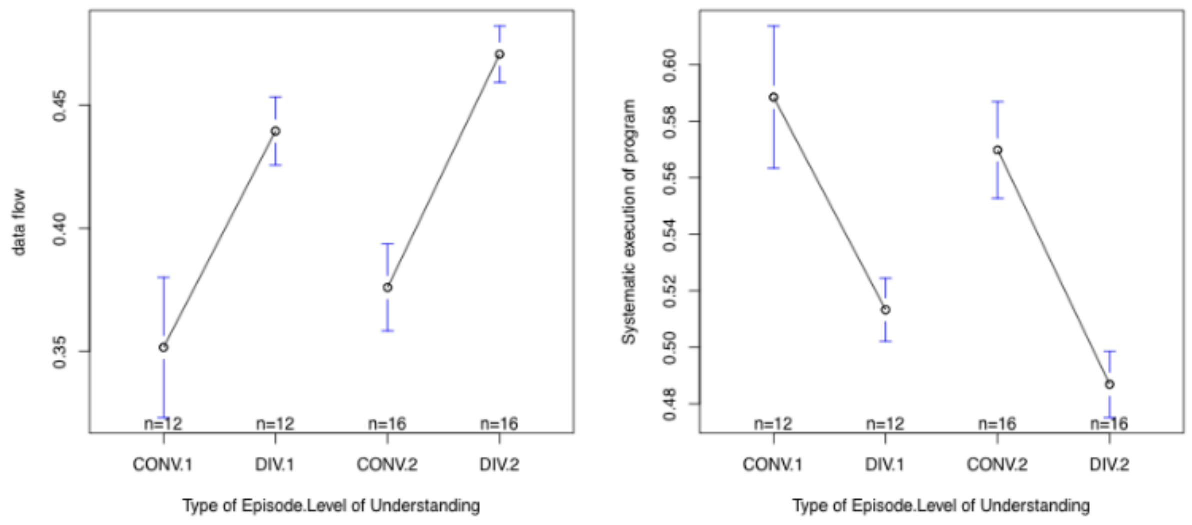


Figure 5: Mean plots for data flow and systematic execution of program for the episodes of interaction and levels of understanding (Level of Understanding $1 = Medium$ and $2 = High$)

6 Discussion and Conclusion

Concerning our first question, we have found evidence for the sensitivity of gaze patterns to the level of understanding. It appears that the gaze of individuals who understood the program better transition more frequently between identifiers and expressions, a transition type that reflects a data flow centred reading of the code. Conversely, individuals with a who got a sense of what the program is doing but were not able to provide the exact explanation, spent relatively more time parsing the program by systematically looking at all types of semantic elements. These findings are compatible with the findings from Jermann and Nüssli (2012) [7] who found that for individual programmers, experts look less than novices at structural elements (type names and keywords) which are not essential when understanding the functionality of the code. Experts look more than novices at the predicates of conditional statements and the expressions (e.g. v /= 10;), which contain the gist of the programs. Our current findings confirm these findings in the context of pairs by using an analysis of gaze transitions between semantic elements . Pairs with high level of understanding put relatively more individual efforts on understanding the entities and their relationships (data flow).

A possible explanation for this difference would be that for the pairs with medium level of understanding some structural elements can act as ”while demons” [3]. On other hand, pairs with high level of understanding show ”as-needed” strategy for building their understanding of the program based on their understanding of the relation between variables in the program [10].

We presented our study for getting the underlying process of the collaborative program comprehension using the eye-tracking data. We put our efforts to distinguish the strategy used for understanding the program by pairs having medium level of understanding from that of pairs having high level of understanding. Pairs with high level of understanding put relatively more individual efforts on understanding the entities and their relationships (data flow). In a convergent episode of interaction, pairs with high level of understanding try to understand the data flow of the program according to the control flow of the program (Systematic program execution). This is attributed to their gaze transitions among all the semantic elements of the program in a convergent phase of interaction and ”to and fro” gaze transitions between expressions and identifiers in the program for the whole interaction (when taken as a whole and in the divergent episodes of interaction).

On other hand, pairs with medium level of understanding put efforts in simultaneous understanding of data and control flow in the program without understanding the meaning of the data variables. This is depicted by their ”back and forth” transitions between expressions and structural elements of the program while they are in a convergent episode of interaction. This behaviour shows that for the pairs with medium level of understanding some structural elements can act as ”while demons” [3]. On other hand, pairs with high level of understanding show ”as-needed” strategy for building their understanding of the program based on their understanding of the relation between variables in the program [10].

Concerning our second question, we have shown that in convergent episodes of interaction, pairs with high level of understanding as well as pairs with medium level of understanding try to understand the program via a strategy of systematic program execution. This is depicted by their ”back and forth” transitions between expressions and structural elements of the program. In comparison, the data flow transitions are less frequent in divergent episodes.

A possible explanation for the differences between convergent and divergent episodes is that programmers are visually searching the code for variable and method names during the divergent phases and that in this case the augmentation of data flow transitions stems from a selective exploration of the code. Another explanation is that during divergent episodes, programmers focus on building basic knowledge about variables and expression which is then discussed during convergent episodes where structural elements of the code are used to define the joint focus of attention. An analysis of the dialogue between partners will help to understand these subtle differences.

Since there were no interaction effects between these factors, we can conclude that both the level of understanding and the type of episode affect the types of gaze transitions that are observed. It is striking that the differences between convergent and divergent episodes are twice as large as the differences between levels of understanding. This seems to indicate that gaze indicators are more sensitive to task-related aspects than to levels of expertise.

References

1. J.R. Anderson. *Cognitive Psychology and its Implications*. Worth Publishers, 1985.
2. P. Baheti and L. Williams. Exploring pair programming in distributed object-oriented team projects. In *In Proceedings of XP/Agile Universe 2002*. Springer Verlag, 2002.
3. J. Bonar and E. Soloway. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL ’83, 1983.
4. R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6), 1983.
5. R. Dale D. C. Richardson and N. Z. Kirkham. The art of conversation is coordination. *Psychological Science*, 18(5):407–413, 2007.
6. J. M. Gottman and A. K. Roy. *Sequential Analysis - A Guide for Behavioral Researchers*. Cambridge University Press.
7. P. Jermann and M.-A. Nussli. Effects of sharing text selections on gaze cross-recurrence and interaction quality in a pair programming task. In *In Proceedings of Computer Supported Collaborative Work 2012*, 2012.
8. W.L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *Software Engineering, IEEE Transactions on*, SE-11(3), 1985.
9. Y. Kawahara. Change-point detection in time-series data by direct density-ratio estimation. *Direct*, 4(2), 2009.
10. J. Koenemann and S.P. Robertson. Expert problem solving strategies for program comprehension. In *CHI ’91 Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, 1991.
11. J. Larkin. *Cognitive Skills and Their Acquisition*, chapter Enriching Formal Knowledge: A model for learning to solve textbook physics problems. Lawrence Erlbaum Associates, 1981.
12. S. L. Lauritzen. *Lectures on Contingency Tables*. University of Aalborg, 1989.
13. S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4), 1987.
14. A. Von Mayrhauser and A.M.Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8), 1995.
15. M.-A. Nussli. *Dual-Eye Tracking Methods for the Study of Remote Collaborative Problem Solving*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2011.
16. R. P. Adams and D. J. C. MacKay. Bayesian Online Changepoint Detection. *ArXiv e-prints*, October 2007.
17. D. C. Richardson and R. Dale. Looking to understand: The coupling between speakers’ and listeners’ eye movements and its relationship to discourse comprehension. *Cognitive Science*, 29(6):1045–1060, 2005.
18. B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8, 1979. 10.1007/BF00977789.
19. E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *Software Engineering, IEEE Transactions on*, SE-10(5), 1984.
20. S. Paul S.R. Tilley and D.B. Smith. Towards a framework for program understanding. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*.
21. B. G. Mitbander T. J. Biggerstaff and D. E. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37(5).
22. E. Terzi and et al. Efficient algorithms for sequence segmentation.

Paper Session 2

Personality

Computer Anxiety and the Big Five

Sarah J Crabbe

York St John Business School
York St John University
s.crabbe@yorksj.ac.uk

Peter Andras

School of Computer Science
Newcastle University
peter.andras@ncl.ac.uk

Keywords: Computer anxiety, Five Factors, Personality

Abstract

This paper explores the relationship between personality traits, as described by the Big Five Factors model, and the likelihood of someone suffering from computer anxiety. The research sample was a cohort of Business School Undergraduates. It was found that for this sample there was a small but significant correlation between two of the traits, agreeableness and emotional stability, and computer anxiety.

1. Introduction

In this era, of pervasive technology, people are increasingly being asked to interact regularly with computers. For some people this interaction causes anxiety and decreases their ability to work to their highest standards (M. J. Brosnan, 1998). It would be useful to be able to identify those people likely to suffer from computer anxiety so that they could be supported effectively in order to become more efficient workers. The personality of the individual might be a contributing factor to this. (Anthony, Clarke, & Anderson, 2000; Ceyhan, 2006; Korukonda, 2005, 2007; Wilt, Oehlberg, & Revelle, 2011)

Within a cohort of first year business undergraduates, the combination of emotional stability (inverse) and agreeableness accounted for 37.9% of the variance of computer anxiety. This suggests that some factors of personality do have an impact on the likelihood of computer anxiety for this particular cohort.

This paper explores the current research in this field and reviews the questionnaires available and explains why CARS and the 5 Factor model were chosen. It goes on to discuss the findings in more detail, and the limitations and implications that these have concluding with suggestions for further work.

2. Background

Personality has been described as "the combination of characteristics or qualities that form an individual's distinctive character:" (Oxford, 2012) although it cannot be measured, only the behaviours that are influenced by it can be measured. So personality, as far as Psychologists are concerned is not a tangible, measurable thing at all, but a construct.

While there are several different models to describe personality, most researchers are agreed that personality does not change very much over time (Maltby, Day, & Macaskill, 2007; Nettle, 2007). The different aspects of a personality are often referred to as factors and there are a range of self-reporting questionnaires which measure these factors each questionnaire relating to a particular model of personality.

MBTI (Briggs Myers, 2000) based on the Myers Briggs model of personality has to be administered by trained psychologists and is expensive to buy and time-consuming to deliver. The results also need to be delivered personally by a professional, specifically trained, psychologist creating a large time investment for the candidate and researcher. The 16PF developed by Cattell (Cattell & Schuerger, 2003) suggests that there are sixteen factors that combine to make one's personality. These were synthesised into themes by (McCrae & Costa, 1999)to find five themes which are referred to as the Big Five Factors (Nettle, 2007). They are more accessible and there are many open-source, well researched questionnaires based on this model that are available for general use.

The Big Five or the Five-Factor model of personality (Maltby, Day, & Macaskill, 2007:177; Nettle, 2007:9) examines behaviours which are indicative of particular types of personality and groups them together into five trait clusters. These trait clusters or factors contain six traits (McCrae & Costa, 1999) and it is the extent to which each trait, within a cluster, is manifested that defines a person's whole personality. The five factors are:

Extroversion: Someone who scores highly for extroversion is more likely to take risks and be extrinsically motivated than someone who has a low score. The traits that make up this cluster are warmth, gregariousness, assertiveness, activity, excitement-seeking and positive emotions The low end is referred to as introversion

Agreeableness: The very agreeable person will demonstrate a high level of trust, compliance, modesty, straightforwardness, tendermindedness and altruism. They may be too quick to concur with others. A low score is tending towards antagonism. Sometimes the scale is referred to as 'Adapter' (High in agreeableness) to 'Challenger' (low in agreeableness).

Conscientiousness: A high score here indicates a person who is competent and well-organised and although they take time to make decisions they are self-disciplined and motivated by achievement, often referred to as 'Focussed'. A low score may indicate a lack of direction or, in a more positive view, an ability to be 'Flexibile'.

Neuroticism: Someone who is a highly neurotic person is likely to react more strongly to negative stimuli than a less neurotic person and is often referred to as 'Reactive'. They will tend to worry more and be more adversely affected by bad news stories. A person with low levels of Neuroticism may not be careful about avoiding danger but will tend to be 'emotionally stable' or 'Resiliant'.

Openness: An open person has lots of ideas often straying into fantasy but always with an awareness of aesthetics and their own values. They are often excitable and active and can be referred to as 'Explorer'. A low score here suggests a person who is closed to experience, sometimes referred to as resistant to change or as a 'Preserver'.

(Adapted from Huczynski & Buchanan, 2007; Srivastava, 2011).

For this research the factors of agreeableness/challenger and neuroticism/emotional stability seemed to be the most important.

But what exactly is computer anxiety? For the purposes of this research it is not the extreme phobic reaction that some people have to technology (M J Brosnan, 1998; M J Brosnan & Thorpe, 2006) which is a reaction similar to that displayed by people suffering from arachnophobia when faced with a spider. Instead the focus will be on those cases where people feel uncomfortable and anxious when dealing with a computer (Howard, 1986).

A person who has computing anxiety will evidence "one or more of the following:

(a) anxiety about present or future interactions with computers or computer-related technology;

(b) negative global attitudes about computers, their operation, or their societal impact;

(c) specific negative cognitions or self-critical internal dialogues during present computer interaction or when contemplating future computer interaction." (Weil, Rosen, & Wugalter, 1990)

In spite of the views of some that the current generation should be quite comfortable around computers and technology (Friedl & Verčič, 2011; Judd & Kennedy, 2011; M. Prensky, 2001) computer anxiety is still prevalent across cultures, age groups and countries (Korukonda, 2007; Shah, Hassan, Embi, & Anxiety, 2011; Tekinarslan, 2008; Weil & Rosen, 1995). It seems to be something that can be passed on from teacher to pupil (Ceyhan, 2006; Elkins, 1985; Epstein & Klinkenberg, 2001) suggesting that at least some element of computer anxiety is a state of anxiety in a particular moment. There is some evidence to suggest that state anxiety manifests only if trait anxiety already exists (Beckers, Wicherts, & Schmidt, 2007) but as some mitigation strategies are successful for some people, (Bostrom & Huber, 2010; Rosen, Sears, & Weil, 1993; Torkzadeh & Van Dyke, 2002; Woszczynski, Lazar, & Walker, n.d.) this may not be the case for all computer anxiety sufferers

Rosen and Weil have developed a questionnaire for use in identifying people with computer anxiety known as the Computer Anxiety Rating Scale (CARS) (Rosen & Weil, 1992) which has been used in many studies around the globe (incl Anthony et al., 2000; Chu & Spires, 1991; Durndell & Haag, 2002; Karal, 2009; Korobili, Togia, & Malliari, 2010; Korukonda, 2005; Korukonda & Finn, 2003; D Mcilroy, Bunting, Tierney, & Gordon, 2001; David Mcilroy, Sadler, & Boojawon, 2007; Rosen et al., 1993; Rosen & Weil, 1995a, 1995b; Shermis & Lombard, 1998; Tekinarslan, 2008)

The results of this are numeric and therefore open to statistical analysis

3. Data analysis and results

We hypothesise that there is a link between the level of computer anxiety and an individual's personality profile. To test this deduction we found out what people's personalities are and whether they have computer anxiety or not. Then the results were analysed with non-parametric statistical tests.

In order to do the research we used the CARS questionnaire (Rosen & Weil, 1992) and a questionnaire based around the five-factor model from IPIP (Goldberg, 1992). These both have 5 point Likert scale responses and the numeric data can be statistically analysed.

The group to be studied was taken from first year undergraduates in a Business School. In the past lectures have found that typically students in this group are not always comfortable with technology and some find it challenging to use the Virtual Learning environment and other applications Because the sample is of undergraduates it is quite easy to gain access to them in whole cohort taught modules. The personality questionnaire was handed out in paper copy and collected in the same session so the rate of return was quite high- although participation was voluntary. The computer anxiety questionnaires were also in hard copy, but the administration of these was done by colleagues to smaller groups and the return rate was not as good.

The sample group consists of over one hundred level one students on an undergraduate Business Management course. They were approached at the end of semester 1 and the beginning of semester 2. The students are a mixture of international students and home students with a minority of mature students, the majority of the group being under 20 years old. All students were invited to take part in the research but participation was voluntary in line with ethical procedures within the university. For the computer anxiety questionnaire there were 55 useable returns. For the personality questionnaire there were 103 useable responses.

Computer anxiety

There are three ranges of computer anxiety, high, medium and low. They are bounded by the values high being greater than 60, medium between 40 and 60 and low is less than 40. In this cohort the distribution is shown in Figure 1.

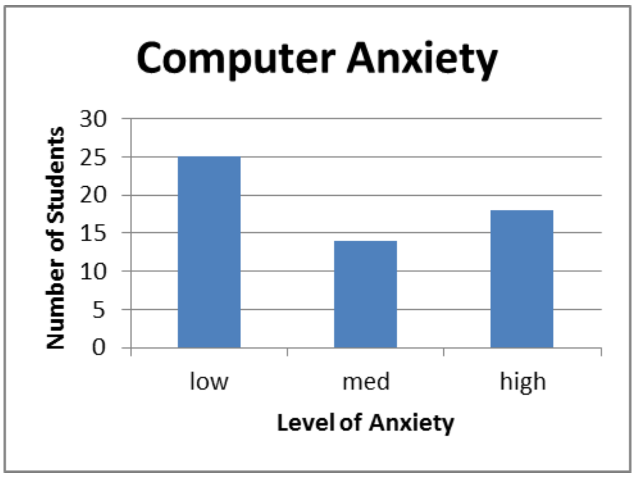


Fig 1: Computer anxiety distribution

A sizeable part of the group had low anxiety but there is still a significant number who are exhibiting high anxiety. The group has a normal distribution (Table A1)

Just under a third of the group who responded are likely to suffer from a high level of anxiety when working with computers. As a lot of the student work must be completed on line or with the use of technology such as word processors this is of concern. However as this was a voluntary exercise and the point of the research was explained it may be that a higher proportion of people who already felt anxious chose to respond.

Personality factors

We used the 5 Factors model of personality and the IPIP questionnaire (Goldberg, 1992). Each factor was scored separately and the results are shown in Table 1.

	Extraversion	*Agreeableness*	*conscientiousness*	*Emotional Stability*	*Intellect/Imagination*
Mean	55.7	67.2	56.3	50.5	61.9
Standard Deviation	16.0	16.0	15.7	15.7	12.6
Minimum	20.0	22.5	10.0	12.5	35.0
Maximum	97.5	100.0	92.5	90.0	100.0

Table 1: Results of the personality questionnaire

Emotional stability has the largest range with a minimum of 10 and a high score of 92.5. Both intellectual and agreeableness had high scores of 100 while agreeableness had the highest mean score and emotional stability had the lowest.

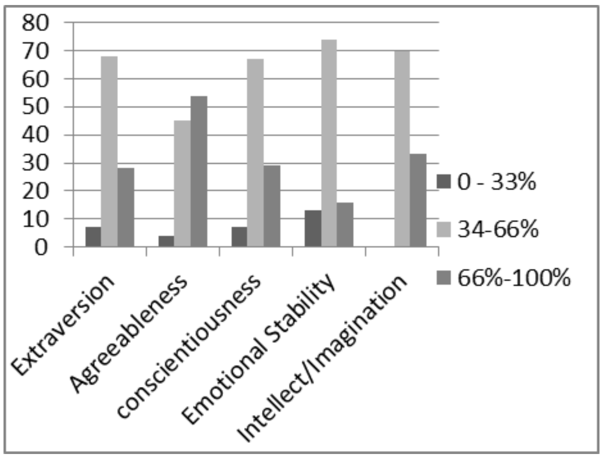


Figure 2: Bar chart of personality profiles

The general overview in Figure 2 shows that the majority of respondents were in the middle range for each trait, other than that for agreeableness which shows a higher level of high scores. It should be noted that there are no low scores for intellect and this may be because the subjects were university students. The overall profiles were normally distributed (Table A2)

Comparing our sample with the reference sample (Goldberg, 1992) using the z-test gives a z value of 14.566, which indicates statistically significant difference between the samples. This suggests that the sample was in some way different from the sample used by the other researchers. Although ethnic data was not collected for the sample other records for the sample suggest that a sizeable minority of the students are from Asia – China in particular. Different cultures can present with profiles that are not the same as Western European/ USA profiles and this can skew the data.

As the data is normally distributed it suggests that we have a representative sample of personality profiles in the group.

Combined findings

There were 28 people who completed both the computer anxiety questionnaire and the personality inventory. The different traits are compared with the computer anxiety scores but only Emotional stability (A) and Agreeableness (B) demonstrated any correlation i.e. had a value of $R^2 > 0.1$ (Figure 3).

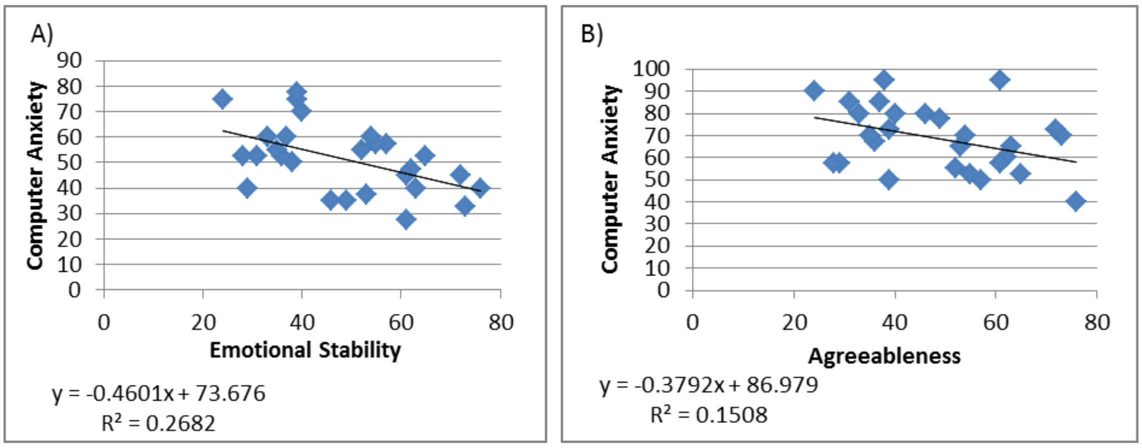


Figure 3: Graphs showing correlation between the traits of Emotional Stability (A) and Agreeableness (B) with computer anxiety

The relationship between 'technophobia' (Another name for computer anxiety) and the traits of Neuroticism (the opposite of Emotional Stability) and Openness (which maps to agreeableness) were established (Anthony et al., 2000) among a South African sample over ten years ago and these relationships have been found to be still true. Work done in New York in 2005 also found high correlation between technophobia and neuroticism (Korukonda, 2005) and a lower negative correlation with openness. However neither of these studies combined the traits to analyse the impact of the combination.

As the data set has less than 50 data points the Shapiro-Wilk test for normality was applied (Table A3) and the data was found to be normally distributed.

Using Spearman's test for correlation we show that Agreeableness and Emotional stability have the highest and significant correlation coefficients. (Table A4)

The testing for linear regression using these two traits shows that the combined model explains the level of computer anxiety better than the traits separately. The analysis also shows that both traits are significant components of the combined linear model (Table 4)

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	78.406	10.213		7.677	.000
	EmotionalStability	-.583	.193	-.518	-3.027	.006
2	(Constant)	106.843	14.380		7.430	.000
	EmotionalStability	-.592	.174	-.526	-3.401	.002
	Agreeableness	-.408	.158	-.398	-2.578	.017

a. Dependent Variable: Anxiety

Table 4 Multiple Regression

The combined linear model, according to Table 4, is:

Computer anxiety score = 106.843 – (0.592 * Emotional Stability score) – (0.408 * Agreebleness score)

Thus the model implies that, the higher the scores of emotional stability and agreeableness for a person, the lower the computer anxiety score of this person.

The summary analysis of the model (Table 5) shows the adjusted R square value is 0.379 i.e. almost 38% of the variance of the anxiety scores is explained by the emotional stability and agreeableness scores of the subjects.

Model Summary				
Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.518[a]	.268	.239	13.08639
2	.653[b]	.427	.379	11.81979

a. Predictors: (Constant), EmotionalStability

b. Predictors: (Constant), EmotionalStability, Agreeableness

Table 5 Summary analysis of the linear regression model

Our work shows that some personality traits contribute towards making a person more likely to suffer from some aspect of computer anxiety but that there may be other factors that have a considerable influence on the presence of computer anxiety in an individual. Our results show that this result holds for populations with different normal personality trait distributions, indicating that probably the link between personality traits and computer anxiety is not culture dependent.

4. Discussion

It may seem surprising that there is such a high level of computer anxiety still present in a population that has grown up surrounded by technology. The suggestion is often made that this generation are digital natives (B. M. Prensky, 2001) and should not therefore experience computer anxiety. In practice it can be seen that a sizeable minority of students are anxious about interacting with certain aspects of technology

The findings show that there is a personality profile that is more likely to result in the individual being susceptible to suffering from computer anxiety, but it does not suggest that other profiles are immune from this. Computer anxiety can present both as a transient state and a consistent trait so it is possible that the identified profiles have a trait of anxiety that manifests as computer anxiety and other profiles have moments when they are in a state of computer anxiety.

Even though the sample size was small the presence of a significant result suggests a larger size data sample is likely to confirm more clearly the correlations and the combined linear relationship that we found. The concurrence with the work of others (Anthony et al., 2000; Korukonda, 2007) from earlier years suggests that the level of computer anxiety and the factors that contribute to it are not changing over time. The personality trait distribution difference between our data and the reference data also indicates that cultural factors do not matter very much for the relationship between personality traits and computer anxiety.

This being the case perhaps suggests that the environment does not have such a large impact as suggested by Prensky (2001) and others who embrace the concept of digital native or the ‘net generation’ (Jones, Ramanau, Cross, & Healing, 2010; Kennedy, Judd, Dalgarno, & Waycott, 2010).

For the front line lecturer, knowing that people presenting with this profile may be more likely to suffer from computer anxiety might help to identify them at the beginning of the teaching year in order to pre-empt anxiety by drawing their attention specifically to a range of intervention strategies. This might support those individuals initially so that they engage with technology and then help to diminish their own anxiety by becoming more competent.

5. Conclusion and further work

Computer anxiety is a complex issue that affects a wide range of people. For the group in this study the personality traits of emotional stability and agreeableness were important in predicting the likelihood of computer anxiety being present in an individual, although it is apparent that there are other factors as well which are as yet unknown. These other factors will probably not be related to culture as our findings concur with those in other countries.

Personality profile has a role to play but this may be more important in the resolution of the anxiety than in the cause of it. More work could be done in this area.

The CARS questionnaire is a valuable instrument for identifying general computer anxiety, but more specific questions need to be asked if we are going to be able to target the interventions in an appropriate way. The next step is to develop and test an instrument that will do that.

6. References

Anthony, L. M., Clarke, M. C., & Anderson, S. J. (2000). Technophobia and personality subtypes in a sample of South African university students. *Computers in Human Behavior, 16*.

Beckers, J. J., Wicherts, J. M., & Schmidt, H. G. (2007). Computer Anxiety: “Trait” or “State”? *Computers in Human Behavior, 23*(6), 2851–2862. doi:10.1016/j.chb.2006.06.001

Bostrom, R. P., & Huber, M. (2010). End-User Training Methods : What We Know , Need to Know. *Data Base For Advances In Information Systems, 41*(4), 9–39.

Briggs Myers, I. (2000). *Introduction to Type*. Oxford: OUP.

Brosnan, M J. (1998). *Technophobia : the psychological impact of information technology*. New York: Routledge.

Brosnan, M J, & Thorpe, S. J. (2006). An evaluation of two clinically-derived treatments for technophobia. *Computers in Human Behavior, 22*, 1080–1095. doi:10.1016/j.chb.2006.02.001

Brosnan, M. J. (1998). The impact of computer anxiety and self-efficacy upon performance. *Journal of Computer Assisted Learning, 14*(3), 223–234. doi:10.1046/j.1365-2729.1998.143059.x

Cattell, H. E. P., & Schuerger, J. M. (2003). *The Essentials of 16PF Assessment*. London: John Willey and Sons.

Ceyhan, E. (2006). Computer anxiety of teacher trainees in the framework of personality variables. *Computers in Human Behavior, 22*, 207–220. doi:10.1016/j.chb.2004.07.002

Chu, P. C., & Spires, E. E. (1991). Validating the Computer Anxiety Rating Scale : Effects of Cognitive Style and Computer Courses on Computer Anxiety, (1987), 7–21.

Durndell, A., & Haag, Z. (2002). Computer self efficacy , computer anxiety , attitudes towards the Internet and reported experience with the Internet , by gender , in an East European sample. *Computers in Human Behavior, 18*, 521–535.

Elkins. (1985). Attitudes of special education personnel toward computers. *Educational Technology, 15*, 31–34.

Epstein, J., & Klinkenberg, W. D. (2001). From Eliza to Internet : a brief history of computerized assessment. *Computers in Human Behavior, 17*, 295–314.

Friedl, J., & Verčič, A. T. (2011). Media preferences of digital natives’ internal communication: A pilot study. *Public Relations Review, 37*(1), 84–86. doi:10.1016/j.pubrev.2010.12.004

Goldberg, L. R. (1992). International Personality Item Pool. *Psychological Assessment*.

Howard, G. S. (1986). *Computer Anxiety and the Use of Microcomputers in Management*. Michigan: UMI Research Press.

Huczynski, A., & Buchanan, D. (2007). *Organizational Behaviour* (6th ed., pp. 149–150). FT Prentice Hall.

Jones, C., Ramanau, R., Cross, S., & Healing, G. (2010). Net generation or Digital Natives: Is there a distinct new generation entering university? *Computers & Education*, *54*(3), 722–732. doi:10.1016/j.compedu.2009.09.022

Judd, T., & Kennedy, G. (2011). Measurement and evidence of computer-based task switching and multitasking by “Net Generation” students. *Computers & Education*, *56*(3), 625–631. doi:10.1016/j.compedu.2010.10.004

Karal, H. (2009). Assessing Pre-Service Teachers ’ Computer Phobia Levels in terms of Gender and Experience , Turkish. *Sciences-New York*, *90*(462), 71–75.

Kennedy, G., Judd, T., Dalgarno, B., & Waycott, J. (2010). Beyond natives and immigrants: exploring types of net generation students. *Journal of Computer Assisted Learning*, *26*(5), 332–343. doi:10.1111/j.1365-2729.2010.00371.x

Korobili, S., Togia, A., & Malliari, A. (2010). Computers in Human Behavior Computer anxiety and attitudes among undergraduate students in Greece. *Computers in Human Behavior*, *26*(3), 399–405. doi:10.1016/j.chb.2009.11.011

Korukonda, A. R. (2005). Personality , individual characteristics , and predisposition to technophobia : some answers , questions , and points to ponder about. *Information Sciences*, *170*, 309–328. doi:10.1016/j.ins.2004.03.007

Korukonda, A. R. (2007). Differences that do matter : A dialectic analysis of individual characteristics and personality dimensions contributing to computer anxiety. *Computer*, *23*, 1921–1942. doi:10.1016/j.chb.2006.02.003

Korukonda, A. R., & Finn, S. (2003). An investigation of framing and scaling as confounding variables in information outcomes : The case of technophobia q. *Information Sciences*, *155*, 79–88. doi:10.1016/S0020-0255(03)00153-1

Maltby, J., Day, L., & Macaskill, A. (2007). *Introduction to Personality, Individual Difference and Intelligence*. Harlow: Pearson Education Ltd.

McCrae, R. R., & Costa, P. T. (1999). A Five-Factor Theory of Personality. In L. A. Pervin & O. P. John (Eds.), *Handbook of Personality Theory and Research 2nd Edition* (2nd ed., pp. 139–153). NY: Teh Guildford Press. Retrieved from http://books.google.co.uk/books?hl=en&lr=&id=b0yalwi1HDMC&oi=fnd&pg=PA139&dq=costa+and+mccrae+big+5&ots=753DM5SsRi&sig=jQRVrTOKR0vJxo21TdLB-2OoEOk#v=onepage&q=costa and mccrae big 5&f=false

Mcilroy, D, Bunting, B., Tierney, K., & Gordon, M. (2001). The relation of gender and background experience to self-reported computing anxieties and cognitions. *Computers in Human Behavior*, *17*, 21–33.

Mcilroy, David, Sadler, C., & Boojawon, N. (2007). Computer phobia and computer self-efficacy : their association with undergraduates use of university computer facilities. *Computers in Human Behavior*, *23*, 1285–1299. doi:10.1016/j.chb.2004.12.004

Nettle, D. (2007). *Personality*. Oxford: Oxford University Press.

Oxford. (2012). Personality Definition. *Oxford Dictionaries*. Retrieved June 5, 2012, from http://oxforddictionaries.com/definition/personality

Prensky, B. M. (2001). Do They Really Think Differently ? *On the Horizon*, 1–9.

Prensky, M. (2001). Digital natives, digital immigrants Part 1. *On the horizon*, 1–6. Retrieved from http://www.emeraldinsight.com/journals.htm?articleid=1532742&amp;show=abstract

Rosen, L. D., Sears, D. C., & Weil, M. M. (1993). Treating Technophobia : A Longitudinal Evaluation of the Computerphobia Reduction Program. *Computer*, *9*.

Rosen, L. D., & Weil, M. M. (1992). Measuring technophobia, (June 1992), 1–51.

Rosen, L. D., & Weil, M. M. (1995a). Computer Availability , Computer Experience and Technophobia Among Public School Teachers. *Education*, *11*(1), 9–31.

Rosen, L. D., & Weil, M. M. (1995b). Computer Anxiety : A Cross-Cultural Comparison of University Students in Ten Countries. *Science*, *11*(1), 45–64.

Shah, M. M., Hassan, R., Embi, R., & Anxiety, C. (2011). Experiencing computer anxiety. *Business*, 1631–1645.

Shermis, M. D., & Lombard, D. (1998). Effects of Computer-Based Test Administrations on Test Anxiety and Performance. *Science*, *14*(1), 111 – 123.

Srivastava, S. (2011). Measuring the Big Five personality factors. Retrieved April 28, 2011, from http://www.uoregon.edu/~sanjay/bigfive.html

Tekinarslan, E. (2008). Computer anxiety : A cross-cultural comparative study of Dutch and Turkish university students. *Computer*, *24*, 1572–1584. doi:10.1016/j.chb.2007.05.011

Torkzadeh, G., & Van Dyke, T. P. (2002). Effects of training on Internet self-efficacy and computer user attitudes. *Computers in Human Behavior*, *18*(5), 479–494. doi:10.1016/S0747-5632(02)00010-9

Weil, M. M., & Rosen, L. D. (1995). The Psychological Impact of Technology From a Global Perspective : A Study of Technological Sophistication and Technophobia in University Students From Twenty-Three Countries. *Science*, *11*(1), 95–133.

Weil, M. M., Rosen, L. D., & Wugalter, S. E. (1990). The Etiology of Computerphobia. *Computer*, (1985), 361–379.

Wilt, J., Oehlberg, K., & Revelle, W. (2011). Anxiety in personality☆. *Personality and Individual Differences*, *50*(7), 987–993. doi:10.1016/j.paid.2010.11.014

Woszczynski, A. B., Lazar, L. D., & Walker, J. M. (n.d.). DOES TRAINING REDUCE COMPUTER ANXIETY ? *Information Systems*, (1999), 1999–2002.

Appendices

Appendix 1 Computer anxiety distribution

The distribution was normal and the mean fell within the confidence range for the reference study sample (Rosen & Weil, 1992) , shown in Table 1.

	N	Mean	Range	Standard Deviation	Skewness
Our Study	55	48.23	20-100	17.27	0.3
Reference study	2,940	41.46	20-100	14.25	1.15

Table A1: Comparison of study samples

Appendix 2 The normal distribution of the personality data

When the scores for all the traits are taken together this gives a total personality profile value. The distribution of the total personality profile values was normal according to the Kolmogorov-Smirnov test.

Tests of Normality

	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
Total personality score	.101	103	.012	.969	103	.017

a. Lilliefors Significance Correction

Table A2 Demonstrating the normal distribution of the personality trait data

Appendix 3 The normality of computer anxiety and personality distributions

Tests of Normality

	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
Anxiety	.158	27	.083	.955	27	.276
Extraversion	.151	27	.117	.963	27	.436
Agreeableness	.109	27	.200*	.972	27	.667
Conscienctiousness	.176	27	.032	.949	27	.201
EmotionalStability	.111	27	.200*	.964	27	.450
IntellectImagination	.115	27	.200*	.967	27	.527

a. Lilliefors Significance Correction

*. This is a lower bound of the true significance.

Table A3: Testing the normality of the computer anxiety and personality trait data distributions

As all the significance values are >0.05 it is confirmed that the data is normally distributed in all cases.

Appendix 4 Correlation

Correlations

			Anxiety	Extraversion	Agreeableness	Conscienctiousness	EmotionalStability	IntellectImagination
Spearman's rho	Anxiety	Correlation Coefficient	1.000	-.117	-.372	.108	-.454*	-.308
		Sig. (2-tailed)	.	.560	.056	.593	.017	.118
		N	27	27	27	27	27	27
	Extraversion	Correlation Coefficient	-.117	1.000	.151	-.453*	.050	.496**
		Sig. (2-tailed)	.560	.	.451	.018	.804	.008
		N	27	27	27	27	27	27
	Agreeableness	Correlation Coefficient	-.372	.151	1.000	.091	-.028	.201
		Sig. (2-tailed)	.056	.451	.	.653	.890	.315
		N	27	27	27	27	27	27
	Conscienctiousness	Correlation Coefficient	.108	-.453*	.091	1.000	-.120	-.105
		Sig. (2-tailed)	.593	.018	.653	.	.552	.601
		N	27	27	27	27	27	27
	EmotionalStability	Correlation Coefficient	-.454*	.050	-.028	-.120	1.000	.334
		Sig. (2-tailed)	.017	.804	.890	.552	.	.088
		N	27	27	27	27	27	27
	IntellectImagination	Correlation Coefficient	-.308	.496**	.201	-.105	.334	1.000
		Sig. (2-tailed)	.118	.008	.315	.601	.088	.
		N	27	27	27	27	27	27

*. Correlation is significant at the 0.05 level (2-tailed).

**. Correlation is significant at the 0.01 level (2-tailed).

Table A4: The results of Spearman's test for correlation between computer anxiety and personality traits

In search of practitioner perspectives on 'good code'

Gail Ollis
Postgraduate researcher
School of Design, Engineering and Computing
Bournemouth University
gollis@bournemouth.ac.uk

Keywords: POP-I.B. Barriers to programming; POP-II.A. Individual differences; POP-II.C. Working practices; POP-IV-A. Approaches to software design.

Abstract

Much of a software developer's job involves working with existing code. The comprehensibility of code therefore has a significant and ongoing effect which can continue long after it was written. Personal experience has shown that some programmers' code is frustrating and time consuming to work with, while others write code that is crystal clear. This paper sets out the basis for a definition of 'good programmer' which emphasises the powerful but invisible productivity consequences for others, rather than the more readily measurable performance of the individual. The conjectured role of personality in shaping such characteristics is also discussed.

Craftsmanship matters

The differences in performance between software developers have long been considered in terms of personal productivity. There is, for example, a pervasive idea in the software industry that individual differences of an order of magnitude exist between high and low performers. This dates back to an incidental finding by Grant and Sackman (1967), who set out to investigate the effects of direct and indirect computer access on programmer performance. In those early days of computing, the norm was for programmers to submit programs to a separate computer room to be run in their absence. Direct access to the computer was a new trend. The access method, though, proved to play only a small part, strikingly overshadowed by large individual differences.

As in many other studies, the task was one of production: the writing and debugging of a piece of code. Speed of production is important to a commercial project's success and profitability. It is also easily measured. The code produced also needs to be sufficiently correct - a characteristic which is measurable less directly, through testing and bug reports. However, these measures reflect only a fraction of the lifecycle of software systems.

A considerable part of software development calls for comprehension of someone else's code. Some of this is contemporaneous. A developer on the same project, engaged for example in integration activities or a change to common code, has the benefit of both a shared knowledge of the project's goals and the opportunity to ask questions of the original writer. The task is different for future readers who later have to understand code in order to fix, extend or reuse it. The only completely reliable information on how the program works comes from the source code; there is no guarantee that other documents, if they exist, have kept pace with the realities of the implementation.

If the code is well crafted, this task can go very smoothly. One experienced developer summed up such craftsmanship thus: "If they're brilliant, they can walk away". There is no need to be able to find the author to ask how the code works; it is clear to see. When this is not the case, the task for reader is altogether more difficult, frustrating and error-prone. It is also more time consuming. Badly written code has commercial consequences, not just aesthetic ones. Unfortunately, unlike pure production from scratch, the time that poor craftsmanship adds to the task of working with existing code cannot be isolated and measured. Its effect is apparent to developers struggling to make enough sense of things to move forward, and all but invisible to others.

It is unfortunate that the word "craftsmanship", which denotes mastery of a skill, it is also closely associated with costly physical artefacts. It does not follow that the same should apply to intellectual work, but the association is nonetheless made and even taken as read. Take Spolsky (2003), for example: "Craftsmanship is, *of course*, incredibly expensive." (my emphasis). To extend a carpentry analogy used by Spolsky, this viewpoint frames the kind of workmanship where "all the screws line up" (ibid.) as something of a luxury. It neglects the significant but intangible expense of puzzling over software that is as confusing as a floor scattered with flat-pack furniture components, a sheet of cryptic diagrams and an allen key.

What is software craftsmanship?

Since the concern here is software development work which involves working with existing code, a peer definition of craftsmanship is the most relevant. Eliciting these opinions is the current focus of the author’s research. Many aspects of other team members’ behaviour can affect a software developer's job, including such things as the approach taken to testing, build management, version control and bug reporting. These aspects will be recorded along with views about features of good and bad software craftsmanship that affect the usability of the source code. Just as there are individual differences among writers of code, so it is among readers. The criteria are unlikely to be the same for each and every one, but any areas of consensus are of particular interest.

There is already plenty of published guidance on how to produce good code (e.g. Goodliffe, 2006). It will be interesting to explore the extent to which the advice from such material coincides with the aspects that practising developers report as a help or a frustration in working with existing code. Some problems may, for example, be difficult to express in the form of a rule or instruction. Returning to the analogy of physical artefacts, an apprentice learns craft skills not from a book but through practice over a long period in order to achieve mastery.

The efficacy of rules and instructions also depends on a sufficiently deep understanding of the concept they are trying to convey. The decisions of interest here are those at the micro level: not architectural concerns, but the everyday tiny design decisions made when implementing a broader design plan. An example would be the advice to use comments which describe not "how" but "why" the code does something. Its efficacy depends on an appreciation of how to communicate a useful "why" message to an experienced reader.

Why does personality matter?

It may be helpful, first, to be clear that many factors are involved in these micro decisions. Expertise, for example, will play a part. Nonetheless, there are also differences between equally experienced programmers; years of practice can account for some of differences between groups, but not for the variation within them.

Social context is also a powerful influence: programming takes place within a multi-layered social environment including the team, office, organisation and culture. Saha (2011) gives a comprehensive overview of environmental factors which can contribute to the presence of unreadable code. As an extreme example of social reasons for incomprehensible code, Goodliffe (2006) cites the case of a Greek programmer refusing to write comments in English, the company's lingua franca.

There is, however, also a significant cognitive dimension to programming. The ability to operationalise a problem into computable steps is prerequisite. It is clear from experience of teaching first year undergraduates tackling their first ever programming that this mapping of a real-world problem into programming language syntax comes much more easily to some than to others. The need for this systemising ability (Wray, 2007) is reflected in the preponderance among software engineers of ‘Thinking’ personality types (who tend to make objective, analytical judgements) over ‘Feeling’ types (who give weight to human and personal concerns) (Capretz, 2003).

Such a skill is sufficient to produce a working program. Many different solutions can achieve the same outcome; apart from being able to run some more quickly than others, the computer is utterly indifferent to the design of the program. This is not the case for human readers. Individual differences in the problem solving and decision making approaches of the programmer have profound effects on how readily the program can be understood by others. Although Bernstein (cited in Tien, 2000) suggests that programming is "best regarded as the process of creating works of literature, which are meant to be read", these works are not always accessible to the reader.

Conclusion

A future stage of research will explore individual differences between programmers to see whether there is a correlation between specific psychological personality dimensions and the programming behaviours, good or bad, about which some consensus has been found among peers. Systemising has already been mentioned as a prerequisite for communicating the programmer’s intentions to the machine; perhaps a certain degree of empathising is helpful alongside this, in order to see things from a reader's point of view and so communicate more clearly to them. Since it is possible to create a perfectly functioning program that is not comprehensible (as illustrated by extreme examples such as obfuscated C programs; Broukhis, Cooper, & Noll, 2012), perhaps an element of conscientiousness (Costa & McCrae, 1992; cited in Halkjelsvik & Jørgensen, 2012) may also prompt programmers to cater for humans as well as machines.

Understanding approaches which tend to produce usable, ‘team friendly’ source code, or otherwise, is a necessary step in order to intervene effectively to improve standards. Just as programmers need to cater for the fellow programmers who are 'users' of their source code, designers of tools, processes and advice need to understand the perspective of their users if these interventions are to work.

Acknowledgements

Thank you to Keith Phalp and Kevin Thomas for their comments.

References

Broukhis, L., Cooper, S., & Noll, L. C. (2012). The International Obfuscated C Code Contest. Retrieved September 13, 2012, from http://www.ioccc.org/

Capretz, L. F. (2003). Personality types in software engineering. *International Journal of Human-Computer Studies, 58*(2), 207.

Goodliffe, P. (2006). *Code craft : the practice of writing excellent code*: San Francisco, Calif. : No Starch Press.

Grant, E. E., & Sackman, H. (1967). An Exploratory Investigation of Programmer Performance Under On-Line and Off-Line Conditions. *Human Factors in Electronics, IEEE Transactions on, HFE-8*(1), 33-48.

Halkjelsvik, T., & Jørgensen, M. (2012). From origami to software development: A review of studies on judgment-based predictions of performance time. *Psychological Bulletin, 138*(2), 238-271. doi: 10.1037/a0025996

Saha, A. (2011). *Origins of poor code readability*. Paper presented at the PPIG 2011, 23rd annual workshop of the Psychology of Programming Interest Group, University of York.

Spolsky, J. (2003). Craftsmanship. Retrieved September 12, 2012, from *http://www.joelonsoftware.com/articles/Craftsmanship.html*

Tien, L. (2000). Publishing software as a speech act. *Berk. Tech. LJ, 15*, 629.

Wray, S. (2007). *SQ Minus EQ can Predict Programming Aptitude*. Paper presented at the PPIG 2007, 19th annual workshop of the Psychology of Programming Interest Group, University of Joensuu, Finland.

Paper Session 3

AI and Knowledge Representation

Schema Detection and Beacon-Based Classification for Algorithm Recognition

Ahmad Taherkhani

Department of Computer Science and Engineering
Aalto University
P.O.Box 15400, FI-00076 AALTO, Finland
ahmad.taherkhani@aalto.fi

Abstract. We introduce a method for recognizing algorithms based on *programming schemas*, which are generic conceptual knowledge with details abstracted out, and *beacons*, which are key statements that suggest existence of specific structures in code. First, the method detects the schemas related to the implementation of the target algorithm and next it computes the characteristics and algorithm-specific beacons from the detected code and uses them as the learning data to construct a classification tree for recognizing new unseen instances.
We demonstrate the method and its performance for searching, heap, basic tree traversal and graph algorithms implemented in Java ($N = 222$). The results show that 94.1% of the schemas are detected correctly and the estimated accuracy of the classification measured by leave-one-out cross-validation technique is 97.3%.

Keywords: POP-II.B. algorithm recognition, detecting programming schemas, program comprehension; POP-I.C. automated assessment

1 Introduction

Programming courses require students to implement a large number of practical exercises. Several automatic assessment tools have been developed to assist teachers in assessing students' work. These tools are capable of analyzing the structure of the program and coding style, verifying that the program works correctly, assessing its run time efficiency, etc. (see, e.g., WebCat [7], CourseMarker [10], Boss [13] and Scheme-robo [20]). Two recent surveys ([1] and [11]) on the functionalities of these tools show that none of them can reliably analyze what kind of algorithms students use to achieve the required functionality and give feedback on how the algorithm is implemented. For example, if students are required to implement a specific sorting algorithm, such as Quicksort, the existing tools can verify by black-box testing that the solution sorts a sequence of integers correctly. However, using these tools, it is very clumsy and unreliable to say whether the implementation conforms to the specification (i.e., whether the algorithm is indeed the required Quicksort, or say Mergesort). This is what we are addressing in our research.

We discuss a combined method for algorithm recognition implemented in a prototype instrument called *Aari system* (an Automatic Algorithm Recognition Instrument). The method combines two different approaches: 1) the schema detection approach, where the implementation of the target algorithm in the given program is detected, and 2) the classification approach, which includes computing characteristics and algorithm-specific beacons that are used as the learning data to train a classifier that is able to classify new previously unseen implementations of algorithms. The main contribution of this paper is to show that the method we have previously presented, as discussed in the following, can be extended to further types of algorithms.

1.1 Background

In our previous work, we have introduced different methods for algorithm recognition and conducted several experiments to show the performance of the methods. We analyzed a set of different sorting algorithm implementations (Bubble sort, Insertion sort, Selection sort, Mergesort, and Quicksort) and discerned various characteristics, such as Halstead metrics, McCabe complexity and roles of variables in the algorithmic code, which can be used to recognize these algorithms. We built a manually tailored classification tree and conducted an experiment showing that the accuracy of the classification tree was 86% [28]. The next step was to apply the C4.5 algorithm [17] to select the best split from the characteristics and build an automatic classification tree that performs better. Evaluated by leave-one-out cross-validation technique, we showed that the estimated accuracy of the classification was 98.1% [26]. The data sets for these two studies ($N = 287$ and $N = 209$, respectively) were collected mainly from textbooks and the Web. We validated our method by authentic students' sorting algorithm implementations ($N = 192$) and concluded that for the aforementioned sorting algorithms, the average accuracy of the method was about 90% [27]. In addition, using the same data set, we introduced a categorization principle for student-implemented sorting algorithms and their variations [29].

We developed a new method based on programming schemas and evaluated its performance with a data set consisting of the five sorting algorithm implementations from the data sets discussed above ($N = 368$). The method detected 88.3% of the implementations correctly [25].

In this paper, we discuss a combined method where first algorithmic schemas are detected from the target program, and then the characteristics and beacons of the selected algorithmic schemas are further analyzed to build a classification tree. Our previous method for building a classification tree computed the characteristics and beacons from the whole given program. The problem with that method was that the given program might (and often do) include pieces of code irrelevant to the implementation of the target algorithm. By first detecting the piece of code in the given program that implements the target algorithm, and further process only this detected code, the combined method allows us to overcome this limitation and thus achieve a better reliability and performance. While the performance of our methods were tested by sorting algorithms in our previous experiments, the main contribution of this paper is to define schemas and beacons for a new data set consisting of the implementations of searching, heap, basic tree traversal and graph algorithms and apply the combined method to this data to evaluate its performance. The promising results show the generalizability of the proposed method.

We start by presenting related work in Section 2. Section 3 gives an overview of the method. Section 4 presents the data set and discusses the method more specifically for the algorithms of the data set. In Section 5, we introduce the classification tree generated for recognizing these algorithms. Section 6 presents the results followed by a discussion in Section 7. The paper ends in some conclusions and future work.

2 Related work

Program comprehension (PC) has been studied from both theoretical and practical points of view. Theoretical studies focus on understanding how programmers comprehend programs, what elements affect the comprehension process, what stages there are in the progress from novice to expert, etc. Different models for PC have been introduced. We will get back to some of these studies in Section 3 when discussing the theoretical background of our method.

Practical studies on PC have focused on developing techniques to facilitate the comprehension process. These techniques have been influenced by the models resulted from the theoretical studies. The characteristics that influence cognitive strategies used by programmers also influence the requirements for supporting tools [24]. By extracting the knowledge from the given program, PC tools can be applied to different problems such as teaching novices, generating documentation from code, restructuring programs and code reuse [16].

PC techniques are mainly based on stereotypical programming plans (also called schemas, idioms, etc.), which are stored in a knowledge base. Understanding the given program is carried out by analyzing the code to find pieces that match the set of plans from the knowledge base. Extracting and matching plans can be performed in *top-down*, *bottom-up* or *hybrid* manner.

Top-down approaches start by the goal of the target program and use it to find the correct plans from the knowledge base. This results in a higher probability to find the right plans from the knowledge base and thus make the searching and matching more effective. However, these approaches need the specification of the target program which are not necessarily available (see, as an example, [12]). In bottom-up approaches, the process of searching and matching is started from small plans and continued to bigger ones. Because small plans can be part of different bigger plans, this technique may become ineffective as the size of knowledge base grows (see, for example, [9]). In hybrid approaches both techniques are used (see, e.g., [16]).

The purpose of algorithm recognition is to determine what algorithm a piece of code implements. Therefore, algorithm recognition facilitates PC. Algorithm recognition can be applied in various tasks including assessing students' work (as discussed in this paper), detecting plagiarism (see, e.g., [8,18]), detecting clones in code (see, for example [2,19]) and source to source program translation via abstraction and reimplementation [30]. In [15], Metzger and Wen discuss a method for replacing algorithms with parallel algorithms that perform the same task. This type of code optimization can be applied to develop compilers for parallel processing machines.

3 Method

In this section, we discuss the combined method very briefly. For a more detailed discussion on the schema detection and classification approaches see [25] and [26], respectively.

The combined method has two main phases illustrated in Figure 1. In the first phase the schemas for the target algorithms are detected (along with the beacons necessary for detecting the schemas). As the result, the code related to the implementation of the algorithm in question is selected for analysis and other non-relevant code is not processed further. The second phase includes extracting and storing the characteristics and beacons[1], building a classification tree and evaluating the estimated accuracy of the classification. In this phase, a classifier is trained to learn how each algorithm class can be associated with specific characteristics and beacons. Thus, the implementations of the learning data are labeled by the correct type of the corresponding algorithm (this is denoted by the dashed arrow in Figure 1). It should be noted that Steps 3 and 4 in the figure are independent from each other. This means that before we build a classification tree, we can evaluate the performance of the classification. Note also that Steps 1 and 2 of the figure are executed as many times as there are instances in the data set, whereas Steps 3 and 4 only once.

Figure 2 illustrates the process of classifying a new unseen data set after the training process of Figure 1. There are two main differences between this process and the training process. First, it starts with testing that the given program works correctly (only error-free inputs will be processed). This step is not needed in the training process of Figure 1, because the data is known. Second, it gets the classification tree already constructed in the training process and ends with recognizing the previously unseen instances of the given data set. These differences are highlighted by gray rectangles in Figure 2. In this paper, we present an experiment that covers the steps illustrated by Figure 1, that is, we discuss the process of building a classification tree and evaluating the estimated accuracy of the classification using leave-one-out cross-validation. We have previously conducted an experiment that covers the steps presented in Figure 2 for sorting algorithms (see [27]). Conducting a similar experiment is out of the scope of this paper and will be reported elsewhere.

3.1 Schemas and beacons

Schemas and beacons are at the core of many program comprehension models. Soloway and Ehrlich define *plans* (which correspond to schemas in their terminology) as stereotypical action

[1] Basically, characteristics, such as software metrics, are the common features that are computed and used for all fields of algorithms, whereas beacons are the features specific to a particular field of algorithm.

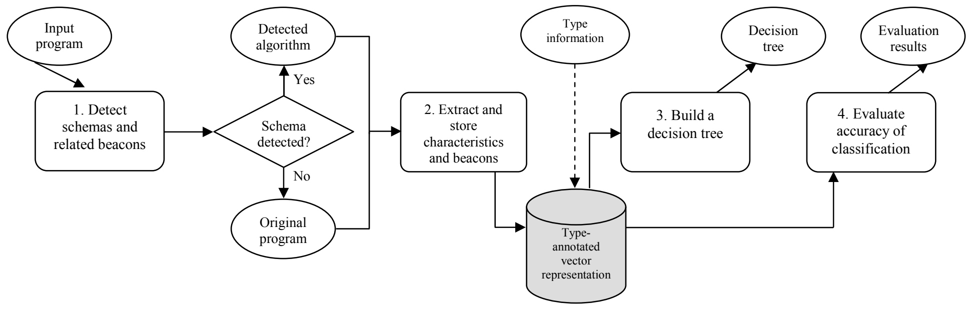

Input program
1. Detect schemas and related beacons
Schema detected?
Yes
Detected algorithm
No
Original program
2. Extract and store characteristics and beacons
Type information
Type-annotated vector representation
3. Build a decision tree
Decision tree
4. Evaluate accuracy of classification
Evaluation results

Fig. 1. The process of building a decision tree and evaluating the estimated accuracy of the classification

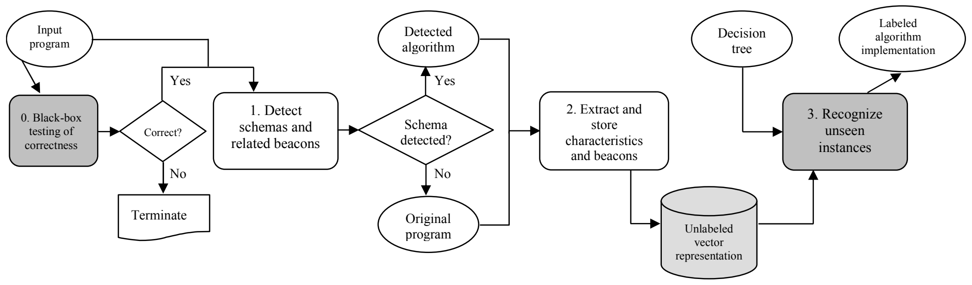

Input program
0. Black-box testing of correctness
Correct?
Yes
No
Terminate
1. Detect schemas and related beacons
Schema detected?
Yes
Detected algorithm
No
Original program
2. Extract and store characteristics and beacons
Unlabeled vector representation
Decision tree
3. Recognize unseen instances
Labeled algorithm implementation

Fig. 2. An overview of the process of recognizing previously unseen algorithm implementations

structures [23]. Détienne defines schemas as formalized knowledge structures in programs [6]. Programmers create and store schemas at different levels of abstraction, and developing schemas is what turns novices into experts. Beacons provide a link between source code and the process of verifying the hypotheses driven from the source code, helping programmers to accept or reject their hypotheses about the code. Beacons suggest the existence of a particular structure in code and experts use them to comprehend programs [4]. In [23], Soloway and Ehrlich define *critical lines* (which can be thought of as beacons) as highly informative and representative lines that are strong indications of a specific plan.

We utilize schemas and beacons to recognize algorithms automatically. The idea is to store abstracted stereotypical implementations of algorithms into a knowledge base of an automatic tool so that the tool can use them to recognize different implementations of those algorithms despite differences in implementation details. This is a similar process as what the experts do while trying to comprehend new programs. We will explain this for our data set in Section 4.

3.2 Creating characteristic and beacon vectors

We compute three types of characteristics: *numerical characteristics*, *truth value characteristics* and *structural characteristics*. The numerical characteristics include *number of operators*, *number of operands*, *number of unique operators*, *number of unique operands*, *program length* (total number of operators + total number of operands), *program vocabulary* (number of unique operators + number of unique operands), *lines of code*, *number of assignment statements*, *cyclomatic complexity (i.e., McCabe complexity [14])*, *number of variables*, *number of loops*, *number of nested loops* and *number of blocks*. Truth value characteristics consist of *recursive* (whether the target algorithm uses recursion), *tail recursive*, *using an auxiliary array* (for algorithms that use arrays) and *roles of variables* (automatically recognized roles of the variables, see [22]).

The structural characteristics help us identify language constructs and different patterns and compute algorithm-specific beacons. These include *block/loop information*, *loop counter information*, and *dependency information*.

We have implemented the method in Aari system. Aari detects schemas and computes the characteristics and beacons for programs written in Java. The input algorithm implementations are converted into characteristic and beacon vectors, which we call *technical definitions* of the implementations. These vectors are given to the C4.5 algorithm [17], which selects the characteristics that best separate the instances of the data set and builds a classification tree.

4 Experiment

We applied the method to searching, heap, basic tree traversal and graph algorithms.

4.1 The analyzed algorithms and data set

We analyzed 10 algorithms from different fields. Since analyzing a bigger set of algorithms is beyond the scope of this paper, our goal was to examine a set of well-known basic algorithms that are commonly discussed in data structures and algorithms courses and textbooks.

We collected a total of 222 algorithm implementations from various textbooks and educational web pages. Table 1 shows the number and the percentage of the implementations of the analyzed algorithms, as well as the abbreviation used for each algorithm in this paper. As indicated in the table, we analyzed the recursive version of depth first search algorithm (DFS) and the non-recursive versions of heap insertion and remove algorithms[2]. Many of the collected programs, especially those collected from the Web, in addition to the code related to the implementation of the algorithm, included non-relevant code as well, such as code related to reading in user provided data, printing the processed data and testing the implementation.

Table 1. The number and percentage of the implementations of the analyzed algorithms. The last column shows the abbreviation used for the algorithms

Algorithm	**Number (%)**	**Abbreviation**
Non-recursive BinSearch	36 (16%)	NBS
Recursive BinSearch	13 (6%)	RBS
Depth First Search (recursive)	15 (7%)	DFS
Inorder traversal	23 (10%)	InT
Preorder traversal	24 (11%)	PreT
Postorder traversal	22 (10%)	PostT
Heap insertion (non-recursive)	22 (10%)	HeapI
Heap remove (non-recursive)	21 (9%)	HeapR
Dijkstra's algorithm	23 (10%)	Dijkstra
Floyd's algorithm	23 (10%)	Floyd
Total	222	-

4.2 Schemas for the algorithms

Figure 3 illustrates the schemas for the analyzed algorithms. We examined all the implementations of the data set and determined an *implementational definition* for each algorithm. We define an implementational definition of an algorithm as the abstraction of its implementation, which reflects the functionality and structure of the algorithm. Implementational definitions do not include implementation details, such as the type of loops or variables, but only high level structural and functional features of algorithms. The schemas depicted in Figure 3 further abstract the implementational definitions of the analyzed algorithms.

[2] DFS has also a well-established non-recursive version, and heap insertion and remove algorithms have also well-established recursive versions. However, we could not gather enough samples of these versions for analysis.

Binary search (non-recursive)

```
LOOP
   MIDPOINT_SEARCH
```

Binary search (recursive)

```
MIDPOINT_SEARCH
   RECURSIVE_CALL
   RECURSIVE_CALL
```

Depth first search (recursive)

```
LOOP
   RECURSIVE_CALL
```

Preorder traversal

```
STATEMENT
RECURSIVE_CALL
RECURSIVE_CALL
```

Inorder traversal

```
RECURSIVE_CALL
STATEMENT
RECURSIVE_CALL
```

Postorder traversal

```
RECURSIVE_CALL
RECURSIVE_CALL
STATEMENT
```

Heap insert (non-recursive)

```
PARENT_INDEX_SEARCH
LOOP
   PARENT_INDEX_SEARCH
```

Heap remove (non-recursive)

```
LOOP
  LEFT_CHILD_INDEX_SEARCH
  RIGHT_CHILD_INDEX_SEARCH
```

Dijkstra's algorithm

```
LOOP
   LOOP
      DISTANCE_UPDATING
```

Floyd's algorithm

```
LOOP
   LOOP
      LOOP
         DISTANCE_UPDATING
```

Fig. 3. The schemas for the analyzed algorithms

For the algorithms of the data set that have well-established recursive and non-recursive versions, the version we analyzed is indicated in the parentheses after their name. Furthermore, indentations indicate the nesting relationship between the loops and blocks.

In the following, we elaborate on some parts of the schemas with semantic meaning and explain how they are computed.

- In the schema of binary search algorithm: $MIDPOINT\_SEARCH$ involves computing the midpoint of a sorted sequence, for example, $mid = (low + high)/2$.
- In the schemas of preorder, inorder and postorder traversal algorithms: $STATEMENT$ denotes whatever function (examining, printing, updating) that may be performed when a node of a binary tree is visited.
- In the schema of heap insertion algorithm: $PARENT\_INDEX\_SEARCH$ denotes computing the index of the parent of a given node with index i, which is $i/2$.
- In the schema of heap remove algorithm: $LEFT\_CHILD\_INDEX\_SEARCH$ for a node with index i is $2i$ and $RIGHT\_CHILD\_INDEX\_SEARCH$ correspondingly $2i + 1$. Some implementations compute the index of the right child of a node by simply incrementing the index of its left child by one, instead of computing it using the index of the node[3].
- In the schemas of Dijkstra's and Floyd's algorithms: existence of the operation denoted by $DISTANCE\_UPDATING$ (also called *relaxation* for Dijkstra's algorithm, e.g., in [5]) in code is investigated by examining whether the given implementation includes the following statements in the nested loops: **if** $v.d > u.d + w(u, v)$ **then** $v.d = u.d + w(u, v)$. That is, the process of $DISTANCE\_UPDATING$ for an edge (u, v) involves examining whether the so far found shortest path to the vertex v can be improved by going through the vertex u, and updating the shortest path to v if this is the case.

[3] If the tree root is at index 0, the parent, left child and right child of each node is located in $(i - 1)/2$, $2i + 1$ and $2i + 2$. We have considered these cases in the implementation of our schema detection method as well.

Note that the schemas of Figure 3 show abstract typical implementations of the algorithms and that slightly different implementations are also possible. For example, some implementations of non-recursive heap remove algorithm might perform *LEFT_CHILD_INDEX_SEARCH* once before the loop and at the end of the loop. As another example, some implementations of Dijkstra's algorithm might have more than one loop within the outer loop. We have not shown these details in the schemas but considered them in the implementation of Aari system.

4.3 Beacons

We analyzed the implementations of the data set to find the following set of beacons specific to the analyzed algorithms that can be used for identifying them. We automatically compute these beacons and give them, along with the computed characteristics discussed in Section 3, to the C4.5 algorithm which selects the best separating beacons and characteristics to generate a decision tree for the classification task. We will present the decision tree in Section 5.

- *MPSL*: MidPoint Search in a Loop; whether the implementation of the algorithm includes searching midpoint of an array within a loop. This mainly indicates implementations of non-recursive binary search algorithm.
- *MPBR*: MidPoint Before Recursion; whether the implementation includes searching midpoint before two recursive calls. This mainly indicates implementations of recursive binary search algorithm.
- *REIL*: REcursion In Loop; whether the implementation includes a recursive call within a loop. This mainly indicates implementations of depth first search algorithm.
- *TSRC*: Two Sequential Recursive Calls; whether the implementation includes two sequential recursive calls. This mainly indicates implementations of preorder and postorder tree traversal algorithms and separates these implementations from implementations of inorder traversal algorithm.
- *TPNI*: Two Parent Nodes Index search; whether the implementation includes searching the indexes of two parent nodes before and after a loop. This mainly indicates implementations of heap insertion algorithm.
- *LRCI*: Left and Right Child node Index search; whether the implementation includes searching the indexes of the left and right child nodes within a loop. This mainly indicates implementations of heap remove algorithm.
- *DUTWL*: Distance Update within TWo nested Loops; whether the implementation includes distance updating (i.e., relaxation) performed within two nested loops. This mainly indicates implementations of Dijkstra's algorithm.
- *DUTHL*: Distance Update within THree nested Loops; whether the implementation includes distance updating performed within three nested loops. This mainly indicates implementations of Floyd's algorithm.

5 Classification tree

Figure 4 illustrates the decision tree classifier generated by the C4.5 algorithm[4]. Using Aari, we automatically analyzed all the implementations of the data set (Table 1) and stored the computed characteristics and beacons in a database. As depicted in Figure 1, because the implementations of the data set are used as the training data and the decision tree is generated based on supervised learning, each implementation is labeled by its correct type in the database.

In the decision tree, the internal nodes, which are depicted by the white ellipses, include the tests that determine the splits. The leaves are illustrated by the gray rectangles and indicate the

[4] J48 (from Weka data mining software), which is an open source Java implementation of the C4.5 algorithm is used to construct the classification tree. URL: `http://www.cs.waikato.ac.nz/~ml/weka/`

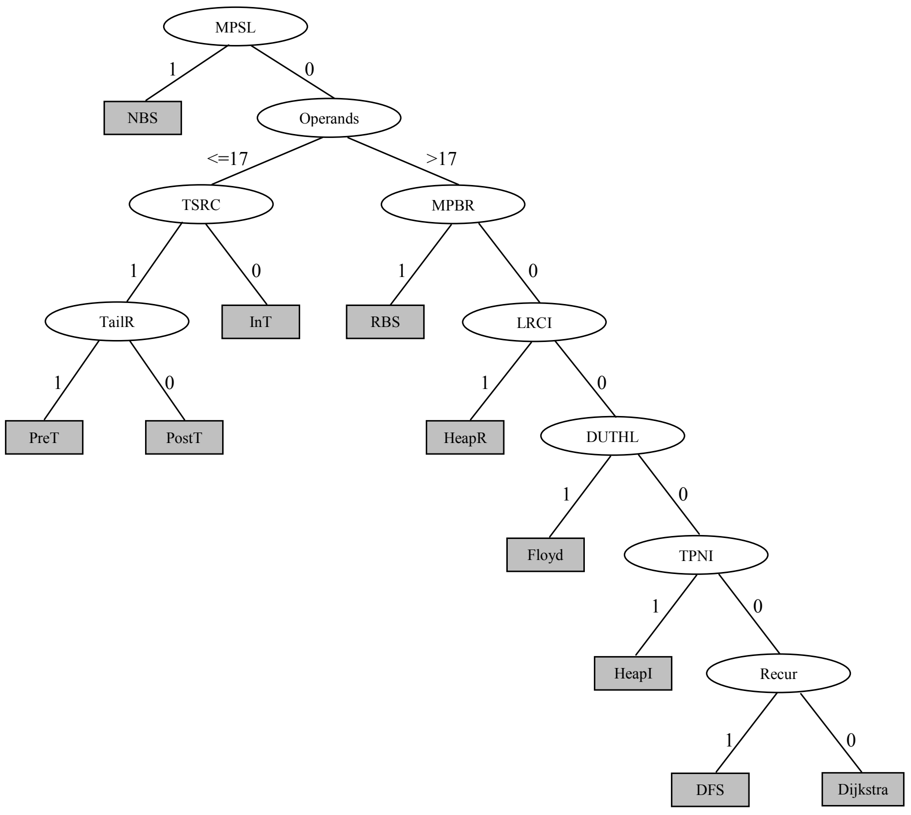


Fig. 4. The classification tree constructed by the C4.5 algorithm on the algorithm implementations presented in Table 1 (see the table for the abbreviations)

analyzed algorithms. The information of the arcs shows the outcome of the test performed in each internal node and determines which child is visited next. The decision tree has 10 leaves and 9 internal nodes (with the root included). From the beacons discussed in Section 4, the following six beacons are selected by the C4.5 algorithm to be used as the tests in the classification tree: *MPSL*, *TSRC*, *MPBR*, *LRCI*, *DUTHL* and *TPNI*. In addition, the characteristics *Recursive*, *Tail_recursive* and *number of operands* are used as the tests in the decision tree as well.

Table 2. Definition of the analyzed algorithms as rules based on the classification tree

Algorithm class	**Rule**
Non-recursive BinSearch	MPSL
Recursive BinSearch	$\neg$ MPSL $\wedge$ Operands $>$ 17 $\wedge$ MPBR
Depth first search	$\neg$ MPSL $\wedge$ Operands $>$ 17 $\wedge$ $\neg$ MPBR $\wedge$ $\neg$ LRCI $\wedge$ $\neg$ DUTHL $\wedge$ $\neg$ TPNI $\wedge$ Recur
Inorder traversal	$\neg$ MPSL $\wedge$ Operands $<=$ 17 $\wedge$ $\neg$ TSRC
Preorder traversal	$\neg$ MPSL $\wedge$ Operands $<=$ 17 $\wedge$ TSRC $\wedge$ TailR
Postorder traversal	$\neg$ MPSL $\wedge$ Operands $<=$ 17 $\wedge$ TSRC $\wedge$ $\neg$ TailR
Heap insertion	$\neg$ MPSL $\wedge$ Operands $>$ 17 $\wedge$ $\neg$ MPBR $\wedge$ $\neg$ LRCI $\wedge$ $\neg$ DUTHL $\wedge$ TPNI
Heap remove	$\neg$ MPSL $\wedge$ Operands $>$ 17 $\wedge$ $\neg$ MPBR $\wedge$ LRCI
Dijkstra's algorithm	$\neg$ MPSL $\wedge$ Operands $>$ 17 $\wedge$ $\neg$ MPBR $\wedge$ $\neg$ LRCI $\wedge$ $\neg$ DUTHL $\wedge$ $\neg$ TPNI $\wedge$ $\neg$ Recur
Floyd's algorithm	$\neg$ MPSL $\wedge$ Operands $>$ 17 $\wedge$ $\neg$ MPBR $\wedge$ $\neg$ LRCI $\wedge$ DUTHL

Based on the decision tree of Figure 4, we can describe each analyzed algorithm (i.e., each class represented as a leaf in the decision tree) as a set of rules [17]. Each set of rules presents the beacon-based technical definition of the corresponding algorithm and covers the path from the root to the leaf for that algorithm. These rules are shown in Table 2. We can, for example, express the implementations of Floyd's algorithm as those that do not have $MPSL$, have number of operand more than 17, do not have $MPBR$, do not have $LRCI$ and include $DUTHL$.

6 Results

We evaluated both the performance of the schema detection method and the estimated accuracy of the classification using the data set described in Table 1. We first present the results of the schema detection followed by the results of the estimated accuracy of the classification.

6.1 Results of schema detection

All the algorithmic schemas of the implementations of recursive and non-recursive binary search, as well as the implementations of inorder and postorder traversal algorithms are detected correctly. For DFS, preorder traversal and heap insertion implementations the accuracy of the schema detection method is more than 90%. For the rest of the implementations, the accuracy of the method is more than 80%. From all the 222 implementations, the schemas for 209 implementations are detected correctly, that is, the average accuracy of the method is 94,1%. Table 3 summarizes these results.

Table 3. The number and percent of the correctly detected algorithmic schemas

Algorithm	**Detected (%)**	**Not detected (%)**	**Total**
Non-recursive BinSearch	36 (100)	0 (0)	36
Recursive BinSearch	13 (100)	0 (0)	13
Depth first search	14 (93,3)	1 (6,7)	15
Inorder traversal	23 (100)	0 (0)	23
Preorder traversal	23 (95,8)	1 (4,2)	24
Postorder traversal	22 (100)	0 (0)	22
Heap insertion	21 (95,5)	1 (4,5)	22
Heap remove	18 (85,7)	3 (14,3)	21
Dijkstra's algorithm	19 (82,6)	4 (17,4)	23
Floyd's algorithm	20 (87,0)	3 (13,0)	23
Total	209 (94,1)	13 (5,9)	222

6.2 Results of the evaluation of the estimated classification accuracy

Cross-validation is a well-known technique for estimating the performance of a classification model. Cross-validation has different types. In k-fold cross-validation, the data set is partitioned into k subsets that include both training and validation data. The accuracy of the classification is evaluated by constructing k different classification trees using $k-1$ subsets as the training set to construct a classification tree and one subset as the validation set to evaluate the performance of the constructed tree. We evaluated the estimated accuracy of the classification using leave-one-out cross-validation technique, where a single instance of the data set is used as the validation data and the remaining instances are used as the training data. Therefore, the training set includes $N-1$ instances and the validation set a single instance. This process is repeated such that each instance of the data set is used once as the validation data. Our data set includes 222 implementations, and therefore 222 classification trees are constructed using 221 instances as the training data and one instance as the validation data for each tree.

The results of the evaluation of the estimated classification accuracy are summarized in Table 4. The column Total shows the total number of the implementations of each algorithm. The column Correct (%) shows the number and percentage of the correctly classified implementations of each algorithm and the column False (%) indicates the number and percentage of the falsely classified implementations of each algorithm. The last column shows what type the falsely classified implementations are recognized as. From 222 instances of the data set, 216 instances are classified correctly (97.3%) and 6 instances are classified falsely (2.7%).

Table 4. The estimated accuracy of the classification evaluated by leave-one-out cross-validation technique

Algorithm	**Correct (%)**	**False (%)**	**Total**	**Falsely recognized as**
Non-recursive BinSearch	35 (97,2)	1 (2,8)	36	Inorder traversal
Recursive BinSearch	13 (100)	0 (0)	13	-
Depth first search	14 (93,3)	1 (6,7)	15	Dijkstra
Inorder traversal	23 (100)	0 (0)	23	-
Preorder traversal	23 (95,8)	1 (4,2)	24	Inorder traversal
Postorder traversal	22 (100)	0 (0)	22	-
Heap insertion	21 (95,5)	1 (4,5)	22	Depth first search
Heap remove	21 (100)	0 (0)	21	-
Dijkstra's algorithm	22 (95,7)	1 (4,3)	23	Depth first search
Floyd's algorithm	22 (95,7)	1 (4,3)	23	Dijkstra
Total	216 (97,3)	6 (2,7)	222	-

7 Discussion

Programming courses require students to solve several practical exercises. Assessing students' solutions especially in large courses is a time-consuming task. A teacher can use the presented method to asses these solutions in the cases where the assignments require students to implement a specific algorithm. This allows the teacher to concentrate on the solutions that do not conform to the specification, instead of assessing all the implementations manually. The method can also be further developed to recognize variations of student-implemented algorithms and provide informative feedback to students about their solutions. This can be done by examining students' solutions and identifying the variations they implement. We have done it in the case of sorting algorithms and reported the results in [29]. Aari system can be trained by the implementations of these variations to identify previously unseen similar variations. Another application of the method in computer science education is detecting plagiarism in students' work, which can be achieved with slight modifications.

The method has potential to be applied to software engineering related tasks as well. In clone detection, as an example, the task is to locate similar pieces of code. Another example includes program translation via abstraction and reimplementation [30], which is a well-known technique for source to source translation. With appropriate further developments, our method is capable of performing these tasks for the implementations that are stored in its knowledge base. However, since these activities involve dealing with large-scale software (unlike the implementations in computer science education), the performance of the method in this context should be evaluated with empirical tests before drawing further conclusions.

As discussed in Section 2, we previously applied the schema detection and classification methods to sorting algorithms (see [25] and [26], respectively). In this paper, we have demonstrated that these methods are generalizable by applying them to the algorithms from various fields with practically the same accuracy. This suggests that we can safely claim that the methods can be extended to cover more algorithms from different fields with fairly high accuracy. The main steps of extending the method to cover other types of algorithms include analyzing those algorithms to identify the schemas that can represent them and the beacons that can strongly indicate these algorithms. Once these are defined, the next step is to develop a tool that can automatically recognize these schemas and extract the beacons (along with the characteristics discussed in Section 3) to be used by the C4.5 algorithm for constructing a suitable classification tree.

In Section 4, we introduced eight beacons for the algorithms discussed in this paper. Two of these beacons, namely *REIL* and *DUTWL* are not used by the C4.5 algorithm in constructing the decision tree of Figure 4. These beacons indicate implementations of depth first search and Dijkstra's algorithms respectively. These two algorithms are distinguished by the characteristic *Recursive* and thus their indicative beacons are left out from the tree. The C4.5 algorithm selects attributes that can discriminate between different classes of data in the best possible way, trying to keep the size of the tree as small as possible.

As in our previous studies, we used the tool developed by C. Bishop and C. G. Johnson [3] for automatically detecting roles of variables in this study. The tool, however, did not detect the roles of the variables in the implementations of the data set accurately enough. With a more accurate role recognizer, roles of variables could have played a distinguishing role in the decision tree of Figure 4 (like they did in our previous studies). For example, the *low* index in implementations of binary search (e.g., $low = middle + 1$) has a *follower* role ([21]) that could be a good beacon for identifying these implementations. We are looking for a better role detector for our future work.

8 Conclusion and future work

We have discussed a combination of two different methods for algorithm recognition and evaluated their performance. As Tables 3 and 4 show, both methods perform very accurately (94,1% and 97,3% of accuracy, respectively). In the combined method, the schema detection method first identifies the code related to the implementation of the algorithm in question. This improves the reliability of the recognition, since the characteristics and beacons are computed from the detected schemas, and not from the whole program.

In real-life programming projects, programmers often use existing standard libraries. However, in a data structures and algorithms course, students need to implement many programming assignments themselves. Aari system can help instructors to check that students have implemented the required algorithm that conforms to the specification. Before this, the correctness of the solutions can be tested by an automatic assessment tool that performs black-box testing.

We applied our methods to sorting algorithms in [25] and [26]. In this paper we have shown that the methods can be extended to cover other fields of algorithms. As a direction of future work, we will further develop the methods to deal with more algorithms and their variations.

9 Acknowledgment

The author would like to thank Lauri Malmi and Ari Korhonen for their valuable comments.

References

1. K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.
2. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.
3. C. Bishop and C. G. Johnson. Assessing roles of variables by program analysis. In *Proceedings of the 5th Baltic Sea Conference on Computing Education Research, Koli, Finland, 17–20 November*, pages 131–136. University of Joensuu, Finland, 2005.
4. R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, USA, 2009.
6. F. Détienne. Expert programming knowledge: A schema-based approach. In J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore, editors, *Psychology of Programming*, pages 205–222. Academic Press, London, 1990.
7. S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, California, USA, 26–30 October*, pages 148–155. ACM, New York, NY, USA, 2003.
8. B. S. Elenbogen and N. Seliya. Detecting outsourced student programming assignments. In *Journal of Computing Sciences in Colleges*, pages 50–57. ACM, 2007.
9. M. Harandi and J. Ning. Knowledge-based program analysis. *Software IEEE*, 7(4):74–81, 1990.
10. C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, 24–26 June*, pages 46–50. ACM, New York, NY, USA, 2002.
11. P. Ihantola, V. Karavirta, O. Seppälä, and T. Ahoniemi. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling 2010)*, 2010.
12. W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. In *Proceedings of the 7th international conference on Software engineering, Orlando, Florida, USA, 26–29 March*, pages 369–380. IEEE Press Piscataway, NJ, USA, 1984.
13. M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3):1–28, 2005.
14. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320, 1976.
15. R. Metzger and Z. Wen. *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization.* The MIT Press, 2000.
16. A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.
17. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, USA, 1993.
18. S. S. Robinson and M. L. Soffa. An instructional aid for student programs. In *Proceedings of the 11th SIGCSE technical symposium on Computer science education, Kansas City, Missouri, USA, 14–15 February*, pages 118–129. ACM, New York, NY, USA, 1980.
19. C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74:470–495, 2009.
20. R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*, pages 133–136, Canterbury, UK, 2001. ACM Press, New York.
21. J. Sajaniemei. Visualizing roles of variables to novice programmers. In *Proceedings of the 14th Annual Workshop on the Psychology of Programming Interest Group (PPIG '02), Brunel University, London, UK.*, 2002.
22. J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments, Arlington, Virginia, USA, 3–6 September*, pages 37–39. IEEE Computer Society Washington, DC, USA, 2002.
23. E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.
24. M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.
25. A. Taherkhani. Automatic algorithm recognition based on programming schemas. In *Proceedings of the 23th Annual Workshop on the Psychology of Programming Interest Group (PPIG'11), University of York, UK, 6-8 September, 2011*, 2011.
26. A. Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *The Computer Journal*, 54(11):1845–1860, 2011.
27. A. Taherkhani, A. Korhonen, and L. Malmi. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling 2012), Tahko, Finland, 15–18 November, 2012*, 10 pages, accepted.
28. A. Taherkhani, A. Korhonen, and L. Malmi. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *The Computer Journal*, 54(7):1049–1066, 2011.
29. A. Taherkhani, A. Korhonen, and L. Malmi. Categorizing variations of student-implemented sorting algorithms. *Computer Science Education*, 22(2):109–138, 2012.
30. R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, 1988.

Some Reflections on Knowledge Representation in the Semantic Web

John Kirby

Computing and Communications Research Centre
Sheffield Hallam University
John.Kirby@shu.ac.uk

Keywords: Semantic Web, Description Logics, WordNet, Knowledge Representation

Abstract

The knowledge representation technology Description Logics (DLs) has become an important component of developments around the Semantic Web. It is suggested here that in order to be really useful, the knowledge represented in DLs should in some fundamental way reflect the way the human mind organises and structures the same knowledge. There is a short historical review of some relevant background work in cognitive psychology, including WordNet. This is followed by a brief introduction to the importance of automatic classification in DLs before considering some issues around ontologies.

1. Introduction

Twenty years ago I became involved in the PEN&PAD project (Nowlan et al, 1991, Kirby and Rector, 1996) on the user centred design of clinical data entry systems based on the formal representation of medical terminology. I also participated in the GALEN project (Rector et al, 1999) which further extended the formal representation of medical terminology using a version of the knowledge representation technology Description Logics (DLs) to develop an extensive medical ontology. At that time, I believed that the use of DLs in the development of the GALEN medical ontology was an extension of the user centred approach because in some way the representation of knowledge was related to the workings of human semantic memory.

For me a key insight of the PEN&PAD project was that a fundamental aspect of the design of a system involves developing a representation of data or knowledge that intuitively corresponds to the understanding of users. For PEN&PAD this meant the development of a system of knowledge representation for symptoms, signs and diseases necessary for clinical data entry by general medical practitioners. In my experience, user interface design, even if it is firmly focused on supporting user tasks, depends to a large extent on how well the underlying representation of knowledge or data fits with the understanding of users. So an approach to the representation of knowledge that in some way corresponds to the way users think seemed to be a significant step in the direction of real user centred system design. User centred design was not just a matter of paying attention to the surface appearance in the form of good interface design but also extended to the deep structures where knowledge representation and data in some way matched or was intuitive to the user.

The proposition of the Semantic Web is based on similar assumptions of the fit between a formal knowledge representation meaning and human understanding. For example,

> *...if the interaction between person and hypertext could be so intuitive that the machine-readable information space gave an accurate representation of the state of people's <u>thoughts</u>, interactions, and work patterns...*(Berners-Lee, 1996)
>
> *In the Semantic Web "information is given well-defined meaning, better enabling computers and people to work in co-operation.* (Berners-Lee et al, 2001)

With the same emphasis on capturing meaning and formal representation, it is not surprising that over the past ten years, DLs have become increasingly prominent in the developments around the Semantic

Web leading to the development of OWL or Web Ontology Language (Patel-Schneider et al, 2004; W3C, 2009).

The literature on the Semantic Web and DLs emphasises the need for formal computer representation of "meaning", "understanding" and "knowledge" all of which is, of course, implied in the word "semantic" itself. Using formal computer representation means that Semantic Web technologies would be amenable to searching and manipulating data "in ways that are useful and meaningful to the human user" (Berners-Lee et al, 2001). Ultimately, however, the final arbiter of the adequacy of the meaning, understanding and knowledge represented has to be the human end-user. This would seem to imply that the essentially human knowledge represented in DLs should in some fairly fundamental way correspond to the way this knowledge is represented in the human mind.

The paper presents a brief historical review of background work in cognitive psychology that is relevant to the development of the DLs approach to conceptual knowledge representation. The importance of automatic classification is described before the discussion of some issues around the development of DLs ontologies.

2. Cognitive Psychology Background

3.1. Hierarchy and Inheritance - Quillian's Model

Quillian (1969) proposed that human semantic memory is organised as a hierarchy of categories or nodes each of which has a set of properties that are also inherited by subordinate categories. In Figure 1, the properties of canary are "is yellow" and "can sing" with a pointer to the category of "bird" to which canary belongs. Because canary is part of the category bird it inherits the properties "has wings", "can fly" and "has feathers" The idea that properties are inherited is also referred to as "cognitive economy".

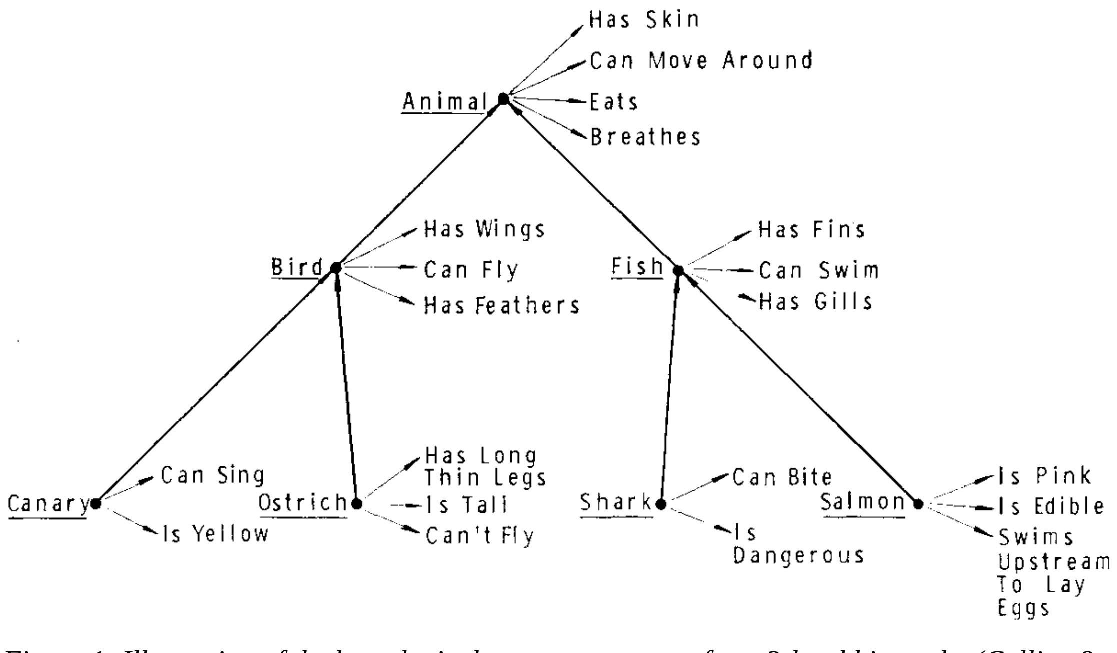


Figure 1: Illustration of the hypothetical memory structure for a 3-level hierarchy (Collins & Quillian, 1969)

Collins and Quillian (1969) tested this hypothesis in reaction time experiments in which subjects were presented with short sentences of two types. In one case subjects were asked to confirm whether or not a word was a member of another category, for example, "a canary is a bird" or "a canary is an animal". In the other case they were asked to confirm whether or not a word possessed a property either directly or inherited from a higher level category, for example, "a canary is yellow", "a canary has wings" or "a canary has skin". It was found that the reaction times of subjects increased in direct proportion to the number of levels in the hierarchy that would have to be traversed.

3.2. Experimental Counter Evidence

A number of subsequent experimental studies cast considerable doubt on both of Quillian's main propositions on semantic memory: its hierarchical organisation and the inheritance of properties.

> **Familiarity**: Conrad (1972) argued that longer reaction times were due to the lack of familiarity of subjects statements such as "a canary has skin". No difference in reaction times was found when the original experiments were repeated with controls for familiarity, suggesting that there is no evidence for the inheritance of properties.
>
> **Typicality**: Reaction times for verification of categories have shown a marked tendency for typical members of categories to have faster response times than atypical ones, for example, categorising a robin as a bird compared with an ostrich or a chicken. Rosch (1973) concluded that categorisation might be by property matching rather than being derived from the hierarchical organisation of memory.
>
> **Arbitrary Categories**: Rips et al (1973) found that subjects verified instances of mammals, such as Cat, Goat and Mouse, more rapidly as animals than they verified them as mammals which would give rise to the conclusion that mammal was the super-ordinate of animal, something that is dismissed as being nonsensical. The conclusion drawn is that the network categories are essentially arbitrary and assigned on a logical rather than empirical basis.
>
> **Fuzzy Categories:** McCloskey and Glucksbery (1978) found that some items were categorised consistently while other items were categorised inconsistently both between different subjects and by the same subject at different times. For example, tomato would be sometimes categorised as a fruit and sometimes as a vegetable. The conclusion is that natural categories have fuzzy boundaries.

These results have led many cognitive psychologies to conclude that Quillian's basic propositions of hierarchy and inheritance were essentially flawed, for example, Eysenck and Keane (2010) and Baddeley et al (2009). Collin who collaborated with Quillian later produced the alternative "Spreading Activation" model of semantic memory (Collins and Loftus, 1975). Despite being more successful in explaining experimental findings, Eysenck and Keane (2009) conclude that it is difficult to assess the adequacy of the spreading activation theory because it does not make precise predictions. For this same reason, the spreading activation theory would not appear to be amenable to the development of computerised models or knowledge representation.

3.3. The Psycho-linguistics - WordNet

However, not all workers in the field accepted the conclusions from this experimental work, in particular those who developed the "psycho-linguistic" WordNet project:

> *An alternative conclusion - the conclusion on which WordNet is based - is that the inheritance assumption is correct, but that reaction times do not measure what Collins and Quillian, and other experimentalists, assumed they did. Perhaps reaction times indicate a pragmatic rather than a semantic distance - a difference in word use, rather than a difference in word meaning.* (Miller, 1990)

Consequently, in WordNet around 100,000 English nouns[1] are organised into sets of synonymous words (synsets) that are hierarchically organised. Each synset is therefore defined in relation to its parent synset plus distinguishing features which is basically in line with Quillian's original theory of semantic memory. In addition, a synset may also include links to other synsets that capture whole-part relationships.

[1] Although WordNet now also contains verbs, adjectives and adverbs, they are beyond the scope of this paper .

It is important to note that its authors have emphasised that WordNet is a “lexical database” organised on “psycho-linguistic” principles but more recently they have also described it as an “ontology” (Miller & Fellbaum, 2007). However, these psycholinguistic principles are not incorporated in any form of automatic classification system which means that semantic inconsistencies are possible. Nevertheless, WordNet has emerged as a significant element in the developments around the Semantic Web and has been used by many workers to construct, compare and merge ontologies.

3. Description Logics

Description logics (DLs) are a family of computerised knowledge representation systems that have their roots in earlier semantic networks (Woods, 1975) and frame based approaches (Minsky, 1975). The proponents of DLs such as Nardi and Brachman (2007) emphasise the importance of the development of logical systems based on sound computational algorithms and in the process move away from any explicit reference to human semantic memory. They conceded that “owning to their more human-centred origins, the network-based systems were often considered more appealing , and more effective from a practical viewpoint than logical systems.” However, they conclude that such systems are “not fully satisfactory because of their lack of precise semantic characterization.”

2.1. Automatic Classification

Like WordNet, the knowledge represented in DLs are ontologies consisting of concepts, also referred to as classes, organised in a strict hierarchy where each concept “is-a-kind-of” its parent. Also like WordNet, concepts may have properties, referred to as roles, that are themselves organised in a subsumption hierarchy. As with object-oriented programming languages, properties of concepts are inherited, so that, for example, all of the properties of animal are inherited by bird and fish - see Figure 1.

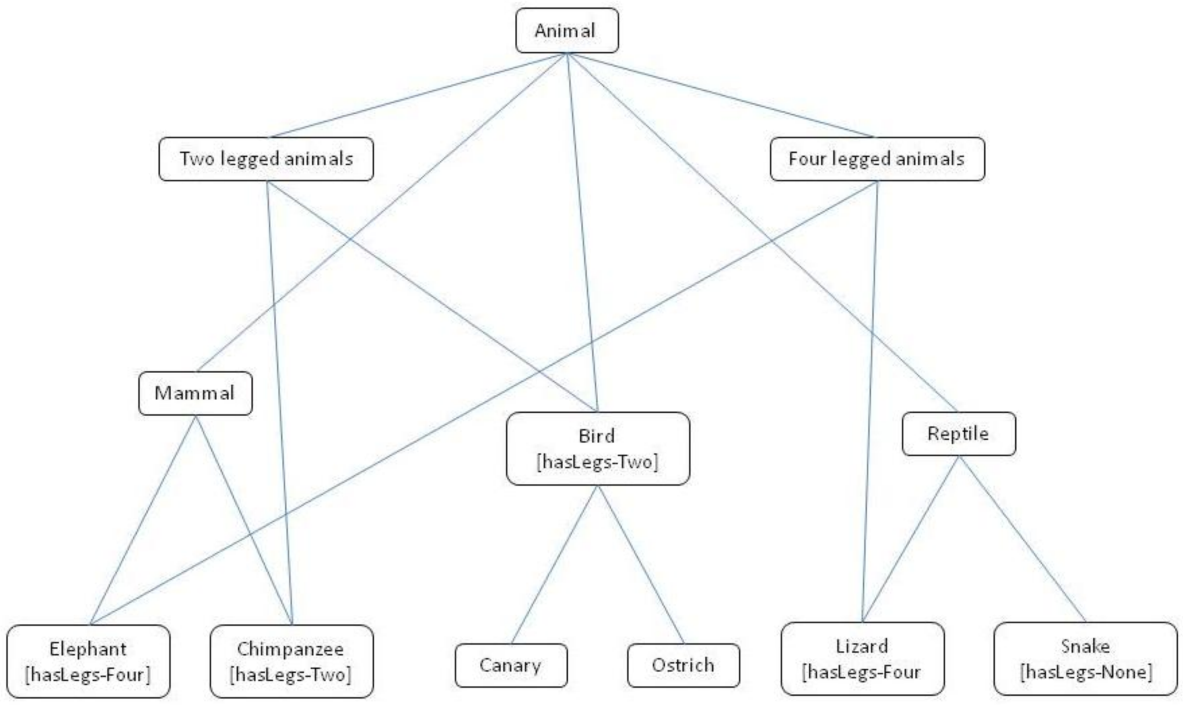


Figure 2: Hierarchy of animals with properties

The key difference with WordNet is that DLs carry out automatic classification. In addition to “atomic” concepts - such as bird and canary - DLs also allow for concept descriptions to be composed from atomic concepts and properties. These “complex” concepts are formed by creating a description that consists of a base concept and one or more property plus value pairs.

Consider the example of a role or property of “hasLegs” with possible values of Two and Four and these properties are applied to Animal concepts as shown in Figure 2. The complex concepts of “Two legged animal” - Animal: hasLegs-Two - and “Four legged animal” - Animal: hasLegs-Four - may then be constructed. When these complex concepts are created in DLs they are automatically inserted into the correct position hierarchy, that is, they are automatically classified. Figure 2 shows that the children of Two legged animal include Chimpanzee and Bird, and by inheritance Canary and Ostrich. In DLs, automatic classification is used both in the construction of DLs ontologies to ensure that they are logically and semantically consistent and for searching and reasoning using DLs.

3.2. Ontologies

The usefulness of DLs depends on the creation of comprehensive ontologies. In his review of progress on the Semantic Web, Horrocks (2007) cites examples of the development of specialist ontologies for biology, medicine, geography, geology, astronomy, agriculture and defence. However, Shadbolt et al (2006) appear to accept that the development of comprehensive and usable ontologies has been relatively slow and go on to suggest that too much effort may have been expended on specialist, or “deep”, ontologies rather than on “shallow” ontologies representing more everyday concepts such as customer, account number and overdraft.

3.2.1. Empirical Findings Revisited

Berners-Lee et al (2001) assert that the Semantic Web will “improve the accuracy of web searches [by using] precise concepts instead of ... ambiguous keywords”. However, I am not convinced that formally and logically correct ontologies will necessarily enable the development of systems providing improved user experience. In spite of being rejected by the developers of WordNet, and essentially ignored by advocates of DLs, there may still be issues arising from the early experimental studies mentioned earlier.

It seems likely that most of us have a mental definition of a “typical bird” that has wings and feathers and, in particular, can fly. Ostriches and penguins are not “typical birds” because they do not fall into that definition of because neither flies which may mean that we put them into one or more different categories of “atypical birds”. However, when asked most people know that “technically” ostriches and penguins are types of birds albeit atypical ones. An alternative explanation of the typicality effect reported by Rosch (1973) might be that the categories used by human beings are not necessarily always the scientifically or logically correct ones assumed in the experiments.

In arguing that hierarchical categories seem to be arbitrary, Rips et al (1973) cite the seemingly anomalous finding that instances of mammals such as horse or elephant were categorised as Animal faster than as Mammal. They go on to reject as “most implausible” the idea that people would categorise such instances of mammal being immediate subordinates of Animal. However, anecdotal evidence suggests that many people do indeed consider such instances of mammals to be direct sub-ordinates of the term Animal. This includes my crossword puzzle dictionary in which the overwhelming majority of entries in the Animal section are mammals (Bailie, 1998). If asked directly, it seems likely that most people would be able to confirm the scientific and logically correct position that such mammal instances were technically mammals and that not all animals are mammals. This may suggest that many of us can simultaneously use different classification hierarchies, in this case an everyday one in which cats, goats and mice are animals and another more logical and scientific one in which they are mammals.

The finding of McCloskey and Glucksbery (1978) that tomato would be sometimes categorised as a fruit and sometimes as a vegetable may also be explained by proposing different hierarchies or at least branches of hierarchy. In the strictly scientific sense tomato is a fruit whilst in relation to food it is a vegetable because it is used in savoury sauces, salads etc rather than in desserts where we would normally think of consuming more typical fruit such as apples, pears or mangos. This would not suggest that natural categories have fuzzy boundaries but that human beings are capable of classifying the same thing in a number of different, but equally valid, ways.

These tentative alternative explanations of experimental findings support the idea of a hierarchically organised semantic memory with the inheritance of properties as proposed by Quillian (1969) and supported by Miller (1990). However, it is suggested that human semantic memory might consist of several possibly contradictory or inconsistent hierarchies involving the same concepts even within the same individual. It is not clear what impact this would have for creating DLs ontologies that more intuitively correspond to the way human beings represent conceptual knowledge.

3.2.2. Developing Ontologies

Shadbolt et al (2006) believe that the Semantic Web needs ontologies that are “developed, managed, and endorsed by committed practice communities”. However, the development and management of ontologies is a time consuming and labour intensive process. Because ontologies are hand-crafted, the hierarchical structure and properties reflect the knowledge, understanding and even the values and prejudices of their authors. In addition, ontologies may be created for specific purposes in the same subject area which may result in mean important differences that may be difficult to reconcile. It may therefore be difficult to develop an ontology in each subject area that is endorsed by specialists in that area.

Furthermore, there is no guarantee that endorsement of an ontology by specialists in a subject area will provide the basis for producing useful and usable systems easily. For example, in the early days of the PEN&PAD project the clinical data entry user interface had been driven directly from the underlying medical ontology. The later version of PEN&PAD that I was involved in with used the more general and re-usable GALEN ontology that had been developed and endorsed by a number of physicians and surgeon. However, this broader ontology contained much that was not relevant for clinical data entry. For example, the fact that arteries and veins were modelled as hollow tubes was of little use to general practitioners entering data about symptoms, signs and diseases of the circulatory system. The solution was to develop an additional processing layer between the ontology and the user interface using pragmatic knowledge of what aspects of the underlying ontology were clinically relevant.

In general, the development of an ontology for a specific purpose in a particular subject area is likely to meet the requirements of that purpose but is unlikely to meet other requirements as in the above PEN&PAD example. Instead of creating a multitude of potentially contradictory single purpose ontologies in the same subject area, it would seem desirable to build general and re-usable ontologies. However, to gain acceptance of correctness across a potentially diverse community of practice such ontologies are likely to become abstract and divorced from any particular purpose. In order to address this issue within the GALEN project, Rector et al (2001) described the development of a layered architecture.

5. Conclusions

DLs and WordNet are major components in developments around the Semantic Web. Despite their differences, both approaches are firmly based on the idea that human knowledge is represented in a hierarchical fashion with the inheritance of properties. This basic proposition seems to lead to the development of simple hierarchies and more extensive ontologies that are “correct” in some strictly logical or accepted scientific sense. It is speculated here that such representations of knowledge may not correspond to the way human beings organise these concepts for everyday purposes.

7. References

Baddeley A, Eysenck MW & Anderson MC (2009) Memory. The Psychology Press, Hove & New York.

Bailie J (1988) Pocket Crossword Dictionary. Hamlyn, London.

Berners-Lee T (1996) The World Wide Web: Past, Present and Future. http://www.w3.org/People/Berners-Lee/1996/ppf.html

Berners-Lee T, Hendler J and Lassila O (2001) The Semantic Web. Scientific American, 284(5): 35–43, May 2001.

Collins AM & Loftus FL (1975) A Spreading-Activation Theory of Semantic Processing. Psychological Review Vol. 82, No. 6, 407-428

Collins AM & Quillian MR (1969) Retrieval time from semantic memory. Journal of Verbal Learning and Verbal Behaviour, 8, 240-247.

Conrad C (1972) Cognitive economy in semantic memory. Journal of Experimental Psychology, 92, 149-154.

Eysenck MW & Keane MT (2010) Cognitive Psychology: a student handbook - 6th edition. The Psychology Press, Hove & New York.

Horrocks I (2007) Semantic Web: The Story So Far. Proceedings of the 2007 International Cross-Disciplinary Conference on Web Accessibility (W4A2007), Banff, Canada, May 7–8, 225. 120–125. ACM Press, NY, USA.

Kirby J and Rector AL (1996) The PEN&PAD data entry system: from prototype to practical system. In: Cimino J, editor, AMIA Full Symposium. Washington DC: Hanley and Belfus, 709-13.

Kintsch W (1980) Semantic Memory: A tutorial. In: Nickerson RS (Ed), Attention and Performance VIII. Hillsdale, NJ. Lawrence Erlbaum Associates Inc. 595-620.

McCloskey ME & Glucksberg S (1978) Natural categories: Well defined or fuzzy sets. Memory and Cognition, 6, 462-472.

Miller GA (1990) Nouns in WordNet: A Lexical Inheritance System. International Journal of Lexicography, Vol. 3 No. 4, 245-264.

Minsky M (1975) A framework for representing knowledge. In: Winston PH (Ed), The psychology of computer vision. McGraw Hill, New York. 211-277.

Nardi D and Brachman RJ (2007) An Introduction to Description Logics. In Baader F, Calvanese D, McGuinness D, Nardi D and Patel-Schneider PF (Eds) The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press.

Nowlan WA, Rector AL, Kay S, Horan B and Wilson A (1991) A patient care workstation based on a user centred design and a formal theory of medical terminology: PEN&PAD and the SMK formalism. In Clayton P, editor, Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care. SCAMC-91. Washington, DC. McGraw-Hill, 855-7.

Patel-Schneider PF, Hayes P and Horrocks I (2004) OWL Web Ontology Language semantics and abstract syntax. W3C Recommendation, 10 February 2004. Available at http://www.w3.org/TR/owl-semantics/.

Quillian MR (1969) The teachable language comprehender: A simulation program and theory of language. Communication of the ACM, 12, 459-476.

Rector AL, Zanstra PE, Solomon WD, Rogers JE, Baud R, Ceusters W, Claassen W, Kirby J, Rodrigues J-M, Mori AR, van der Haring EJ, and Wagner J. (1999) Reconciling Users' Needs and Formal Requirements: Issues in Developing a Reusable Ontology for Medicine. IEEE Transactions on Information Technology in Biomedicine, Vol 2, No 4, 229-242.

Rips LJ, Shoben EJ & Smith EE (1973) Semantic Distance and the Verification of Semantic Relations. Journal of Verbal Learning and Verbal Behaviour, 12, 1-20.

Shadbolt N, Hall W & Berners-Lee T (2006) The Semantic Web Revisited. IEEE Intelligent Systems, May/June 2006, 96-101.

W3C (2009) OWL 2 Web Ontology Language, Document Overview, W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-overview/

Woods WA (1975) What's in a link: Foundations for semantic networks. Reprinted in: Brachman RJ and Levesque H.J (Eds), Readings in Knowledge Representation. Morgan Kaufmann Publishers, San Francisco, California, 1985. 217–241.

Paper Session 4

Expertise

Thrashing, Tolerating and Compromising in Software Development

Tamara Lopez[1], Marian Petre[1], and Bashar Nuseibeh[1],[2]

[1] Centre for Research in Computing, The Open University
t.lopez, m.petre, b.nuseibeh@open.ac.uk
[2] Lero - The Irish Software Engineering Research Centre, Ireland

Abstract. Software engineering research into error commonly examines how developers pass judgement: to isolate faults, establish their causes and remove them. By contrast this research examines how developers experience and learn from things that go wrong. This paper presents an analysis of retrospective accounts of software development gathered from a single organisation. The report includes findings of how work is conducted in this organisation, and three themes that have emerged in analysis are discussed: thrashing, tolerating and compromising. Finally, limitations and implications for future research are given.

1 Introduction

Software rarely works as intended when it is first written. Things go wrong, and developers are commonly understood to form theories and strategies to deal with them [4]. This research began with a desire to examine in more detail how developers reason while making software, and in particular, how they reason when things go wrong.

Software engineering research, by contrast, is commonly concerned with *passing judgement*, assessing why a piece of software has failed and who is to blame [11] , or why a piece of software is flawed and how to prevent such flaws in the future[7]. This kind of research commonly examines errors in the context of bugs, elements of software *as written* that produce undesirable, unexpected and unintended deviation in behaviour[1]. It tends not to examine errors from the perspective of individual developers, nor to consider how things that go wrong along the way are fixed, thus contributing to better software and better developers.

In the rest of this paper, we report a study designed to address these gaps.

2 Related Work

Understanding the personal strategies and theories that developers have for why things go wrong and how to deal with them is an old, but under-examined concern in software engineering research[4]. It is particularly evident in root-cause analysis, a method used to improve software engineering process.

Root-cause studies identify the kinds of faults that predominate in a system, and determine how process can be altered so as to prevented their occurrence in the future. They draw upon data from bug and modification reports [9, 10, 6], but also make use of in-process questionnaires [2] and retrospectively administered surveys [8]. Data are analysed and classified into taxonomies that identify the root-causes for errors. The classified set of data forms the basis for additional examination of particular code features such as complexity [13], interface defects [9, 10] or more generally, environmental factors that influence software dependability [2]. The findings similarly address familiar software engineering themes. Complexity is found both to correlate to error frequency [13], and not to [2]. Application programming interfaces are found to have particularly high frequencies of errors associated with them [9, 10] while these and other causes are evaluated in terms of the cost associated with finding and fixing faults [13, 2, 6].

Over the years, these studies have consistently made two suggestions for additional research. The first is that data about errors should be collected from the entire development cycle, not just at points of testing and integration[4][7], and should not be collected too long an interval of time after events have passed[8]. The second is that studies should be made that examine the causes of "human erring"[4, p.331], including factors such as problems of understanding[4], inexperience[10], lack of information[8], and skill mismatch[6].

Unfortunately, though these papers offer clear ideas about what to examine, they do not offer many suggestions for how to go about doing it. They do, however, suggest likely challenges. To get around the fact that time erodes knowledge about errors, Perry suggests that programmers be asked to classify their errors as a part of closing modification and bug reports [7], a technique found not to work particularly well by Leszak et al.[6]. Organisational access is noted to be difficult to attain when it requires sharing information about mistakes[7], and management can seriously constrain study design, in extreme cases resulting in retrospectively gathered, anonymous self-reports[8].

3 Talking to Developers: Critical Decision Method

We wish to address a long-held concern in software engineering research about how developers experience and learn from things that go wrong in their work. Methodologically, we seek to address the gaps identified by the root-cause analyses: to gather evidence about error from the full development cycle, with particular emphasis paid to those areas that have been associated with "human erring": problem understanding, experience, information and skill.

To explore both gaps, we adapted the critical decision method, as described in *Working Minds: A Practitioner's Guide to Cognitive Task Analysis*[3]. The critical decision method was developed to study how decisions are made in real world settings. In addition to illuminating how people think on the job, the larger framework of cognitive task analysis assists researchers in understanding expertise in individual domains, by revealing the differences between how experts and novices approach and manage their work.

This method has been used before to study expertise in software development. One notable study was conducted at Bell Labs to produce a training course in expert debugging skill[14, 12, 5]. It began with the premise that software development is a domain in which there is a clear difference between the way experts and novices perform their work and where experts become so only after years of experience on the job. Data was drawn from sixteen critical decision method interviews with experts, and two interviews with typical (i.e., not expert) developers. Findings were corroborated and enlarged through a focus group and via surveys distributed to developers throughout the company.

The study found that expert debuggers, like experts in other domains, think extensively about problems before taking action. This manifested in waiting longer to employ debugging tools, and in finding information about what to try next rather than jumping into "poorly directed" but hopeful activities. Study participants suggested that less experienced developers, by contrast, were reported to "thrash" around, trying to find solutions.

The authors also reported differences in how experts and novices handled "close reading" of code to establish what it does, and the past history of the data it handles. Both novices and experts were found to read comments, but novices were less critical of what comments say, tending to take them at face value. By contrast, experts treated the comments as evidence of the number of hands present in a piece of software, and the conditions under which authors were working. Finally, the authors found that experts read code as a last resort, preferring instead to seek help from other developers with detailed knowledge of the software.

Though this study collected detailed information about technical aspects of bug fixing, feedback given by participants led the analysts to focus their efforts on explicating the social aspects of debugging. It did not examine expertise in the context of other kinds of development activity, and reported findings based on the views of acknowledged experts in a single organisation. The findings are thus compelling but leave room for more detailed examination of development activities other than debugging, and the inclusion of non-expert perspectives.

3.1 Study Execution

For our study, seven individuals were interviewed over the course of four weeks. Participants perform a range of software development tasks in an established UK digital humanities centre, described in 2008 by the Council on Library and Information Resources as an environment "where new media and technologies are used for humanities-based research, teaching, and intellectual engagement and experimentation"[15, p. 4]. Each informant was asked to recount an incident in which they played a discrete role, in audio-recorded sessions that lasted from between forty-five and seventy-five minutes.

Interviews were conducted by a single person, but otherwise followed the basic procedures for conducting a critical decision method interview. These entail examining a single incident in four semi-structured "sweeps". In the first sweep, the informant and the researcher identified an incident, broadly defined as one having taken place in the previous two weeks and in which the informant was a key decision maker. In the second, a timeline was established to note critical decision points. In the third, deepening probes were used to develop comprehensive and detailed understanding about the incident. Though researchers often selectively use probes at this stage to examine one or two cognitive phenomena, this study made opportunistic use of a range of probes, with an aim to identify in analysis those which are most effective for learning about things that go wrong. Finally, the informant was asked to consider hypothetical alternatives to decisions taken.

Each interview concluded with questions about the informant's educational and professional background. Informants were asked to give the researcher copies of artefacts mentioned in the discussion.

A Note About the Workspace All of the interviews took place at developers' desks; five of the seven developers were located in the same open plan office. Desks were located in close proximity to one another, and employees were aware that interviews were being conducted. The sixth developer was located in a different open plan office, and the final interview was held in the participant's private office.

The choice to conduct these interviews *in situ* was deliberate. It was felt that conducting them in the developer's own environment would allow for better access to physical and digital artefacts that came up in conversation. Given the focus on problems and challenges, it was also hoped that holding these discussions in the open would signal to informants that the topic was not being pursued in order to assign blame, but rather in a spirit of inquiry. Informants gave no indication that the choice of venue made them uncomfortable, though it was noted in several cases that informants displayed discretion in referring to colleagues located in the same office, either by lowering their voices or by referring to them simply as "my colleague".

Information on computer screens, paper diagrams and a poster on the wall were used to initiate discussion in three cases. In addition, informants shared source code with the interviewer, explained the output of stack traces and demonstrated debugging tools, prototypes and the software being built. Several developers appeared to remember with their fingers, orally recounting details while at the same time accessing files and websites and conducting internet searches similar to those they had used in solving problems.

3.2 Data Analysis

Data analysis began at the point of collection. Terms were checked with informants and information previously given was stated back for clarification and correction. In several instances, restating information also resulted in the addition of omitted details. Immediately following each interview, notes taken during the interview were annotated and expanded. In addition, reflection was made to describe impressions and details of the major topics raised in the interview, and to evaluate application of the method.

In addition to the in-interview corroboration, informants were sent follow-up email messages seeking additional materials mentioned in-interview, and inviting them to provide additional comment. Informants have also been sent draft copies of reports featuring information pertaining to their case.

A near-verbatim transcription was created of each interview. Each transcript was read and annotated in an inductive, iterative process to identify themes in the data. The analysis of individual texts was supplemented by the development of matrices and diagrams to explore points across cases.

3.3 Validity

Given its exploratory aims and its focus on a single organisation and method of data collection, the results of this study alone cannot make strong claims of validity. In addition, it must be stated that the primary researcher had prior understanding about the organisational culture in which the informants work. However, this researcher had no direct knowledge of any of the projects discussed.

4 Results

This section presents results of the cross-case analysis of six interviews. The seventh interview did not result in the identification of a clear incident, and is not included in this report. This interview did yield information about how work happens in this organisation, which has been used in analysis for comparison with other cases. The views of individual developers are presented using pseudonyms.

4.1 Participants

Given the focus on understanding more about things that go wrong, it was felt that having access to participants with a range of experience would be useful. To that end, this study did not strive to identify and interview acknowledged experts. Seven people were interviewed, six men, and one woman. The youngest participant was in her late twenties, and the oldest was in his sixties. The rest of the participants were in their thirties and forties.

Developers old and new to the organisation were interviewed, with one having less than a year at the organisation, and one over ten years. Two participants had computing degrees, one had a computing postgraduate degree, one had a computing applications postgraduate degree, and one had a postgraduate computing diploma. Three had industry computing experience in the web media, financial, education and GIS sectors. Two had post-graduate or research degrees in the social sciences and humanities. There were also humanities computing specialists, with one informant having at least two decades experience in digital humanities work, and a second having a decade and a half. These informants had both worked in multiple organisations on digital humanities projects, while for the remainder, this was their first position in a digital humanities centre.

4.2 Projects

Informants described work performed for three projects. James described refactoring a piece of software for a tool he is building that facilitates note-taking and annotation (Project A). Valentin described a project for which he was the sole application developer, tasked with creating both an editorial tool and a web edition for displaying a critical edition of texts (Project B). The remaining four participants, Joaquim, Marisa, Evan and Richard described performing different tasks for a single project to support detailed annotation and display of medieval handwriting (Project C).

The latter two projects (B and C) follow the same general model: tools are created for use by domain specialists to manage and create data related to physical, often historical materials. These data are in turn presented to the public via other pieces of software that are also developed by the centre. Public facing outputs take the form of web editions of texts and web reference tools. In some cases, projects also produce print monographs.

This model is common in this centre, and as a consequence, development staff routinely produce software for multiple user groups with different functional requirements, and different usability thresholds. Domain specialists are full partners in the project. They closely interact with developers and analysts and are prepared to work with tools that require complicated installation procedures or which have a less than finished feel. Their priority is to have a piece of software finished enough so that they can advance their research. By contrast, readers of public facing web editions and reference tools, themselves also typically domain specialists, have an expectation that the tools they use will be finished to a very high standard. Developing tools for the public, and doing so in new and innovative ways, is a high priority for the centre, but the requirements for these tools emerge slowly, sometimes over a period of years as the specialists work with original materials and develop their understanding about what they mean.

The data produced and managed in these projects are different from commercial data: they are less structured, orientated around natural language and approximate. One developer characterised them in this way:

> "So a good example are dates. If you say the date of this manuscript is around 1113 well it could be this date or it could be that date. Or even worse somebody is saying it is that date, somebody is saying it is that date, somebody is saying it is that date. In the commercial world it is just a single precise date to the millisecond. Here you want many dates by different people and you want all the opinions shown on your website and preserved. So the interpretation is very important."

4.3 How work is done

Developers in this centre tend to work alone, even when assigned to tasks for the same project. A single person may be assigned to work on all deliverables, or different people may be assigned to different areas of the software. It is common for developers to work on the same project at different times. Developers know the others who are working on their projects, and report that they attend meetings at which other developers are present, however each works in reference to the overarching project team. Participants identified having an area of technical expertise such as in application or interface development or in data modelling. Despite this, several recounted the need to learn new skills to meet requirements for projects that emerged over time. For example, one application programmer described learning and implementing client-side technologies, while another developer who was proficient in XML data modelling described a need to learn relational data modelling.

Developers also take the initiative for prioritising and organising their work. This can involve adopting new working practices, as in the case of one developer who described introducing a new working style on his project as "agile-like," with rapid iterations and frequent meetings with project partners. Another described how his responsibilities at the organisation are growing, and how in his present project he extended the original task to do more on his own initiative, more or less as "the accepted order of things". Ad-hoc technical teams are important, with one participant describing "finding" his way on a project with the help of an "amazing" colleague who offered technical advice and guidance about how to manage relationships with partners. Another described "luck" in finding the solution to his issue as due in part to the technical expertise of a colleague who was not working on the project. A third described looking to a trusted colleague for help before relying on internet fora and other technical documentation.

4.4 The Problems

Though an aim of this study was to examine things that go wrong in contexts other than in bug reporting and bug fixing, bugs did feature in some of the stories. The starting point for discussion with Joaquim was a bug that had been reported to him by a project partner. Valentin, despite describing his issue as “not necessarily a bug, it’s an improvement” recounted that his issue nevertheless manifested as a bug four times in the course of a year and a half, in different pieces of software that were being used and built. The first manifestation brought the issue to his attention, two later occurrences were “expected,” and one was a surprise, helping him to realise how “widespread” the issue was, and causing him to re-prioritise and plan for solving it. James reported finding and fixing small ”secret” bugs[4] as a part of his work to refactor the storage layer for his software.

Three participants described issues directly connected to writing software. Joaquim described an issue in implementing an interaction model for annotating digital images in a browser. Valentin described problems in rendering special characters from historical texts in web browsers. Evan described the personal challenges he faced in getting a local copy of an application framework up and running. These included resolving version dependencies in installed software and correcting filesystem path information in a configuration file.

In three other cases, the issues described revealed how external factors influence software development in this organisation. Richard described developing a data model for a project that would take into account two legacy models, work that involved close interaction with a project partner to identify and fully specify concepts. His issues primarily arose from pressures on his time, the need to “get something working” for this project, while still meeting the demands of other projects. James described a breakthrough in his thinking about how to re-architect a piece of software. His story detailed not the thing going wrong, but its flip side, how longstanding “motivators for change” that came from the larger research community in which he participates were suddenly resolved. Marisa described issues she faced in meeting complicated, ambitious requirements for a user interface. Her incident included technical aspects, but also involved learning how to recognise her own limits and to manage the expectations of her project partners.

4.5 Time

Every participant reported a working pattern of “fits and starts”, the need to pick up a task and set it down as required to meet the demands of multiple projects. Perhaps as a result of this practice, the issues described by participants included relevant details that were at times temporally distant from one another. In the stories recounted by Richard and James, details were given that had occurred at least a year prior to the interview. Evan and Marisa reported events with two clear intervals, one occurring in the weeks leading up to the December holiday break, and the other in the weeks following the break. Valentin reported events that occurred a year and a half prior to the interview, and the timeline for his interview included several distinct decision points, moments when the issue manifested in a way which changed priorities and actions. Evan’s timeline was much more compressed, comprising the events of a single day which had occurred in the week prior to the interview. He set out the major episodes at the beginning of the interview, and these remained relatively stable for the rest of the interview.

In some cases, timelines were more difficult to establish, perhaps because work related to the issue was ongoing at the point at which the stories were collected. Valentin, with whom a timeline was quickly established, was coming to the end of his development on the project he discussed. Evan was at the very beginning of his work on the project, and did not have a long history of experience to relate. By contrast, Joaquim, James, Richard and Marisa were interviewed in the midst of longstanding work on a project, and seemed to have more difficulty in establishing a sequence of linked events. A timeline did emerge, but it was established in analysis, and wasn’t particularly useful in structuring the interview protocol. In these cases, the timeline was also not as precise, and possibly not as accurate.

As might be expected, finer detail was collected about events that occurred close to the point of interview. For example, Joaquim was able to report in some detail his interactions with a debugger and IDE for work related to the incident that had occurred the day previous, but did not report in such detail the process he used to research technologies at the start of his work for the project some four months prior. Likewise Evan was able to recall in detail his experiences of a few days previous, but did not provide as much detail about troubleshooting a related incident some two months prior.

5 Discussion

Three themes have emerged in analysis of the cases of Joaquim, Valentin and Evan, the informants who reported incidents directly connected to writing software. This section identifies and defines these themes, and presents a report of each.

Thrashing is identified in the Bell Labs study as poorly directed, ineffective problem solving. There is some suggestion that novices and experts thrash, but novices fail to realise they are doing it in good time, and fail to break out of it. Experts, by contrast, realise when they are thrashing, and seek help from colleagues with more experience. In these interviews, Evan describes a day in which he spent time thrashing while trying to set up a local copy of a web application framework.

Tolerating - Valentin describes how he tolerated an error for over a year, implementing temporary solutions along the way. He reports this activity as strategic, a behaviour that is consistent with descriptions of expert debuggers in the Bell Labs study.

Compromising - These interviews suggest that developers settle at times for sub-optimal solutions in order to move work along. This is exemplified in Joaquim's story, in which a bug fix results in a working solution, but one with which he is intuitively dissatisfied.

5.1 Thrashing: At that point I made a cup of tea

> Evan described a day in which things went wrong while setting up a local copy of an open source web application framework, a task necessary to complete in order that the "real work" could begin. Though relatively new to the framework and to the language it is implemented in, he had done the task before for a different project a couple of months prior. That time the process had not been smooth, and he had not written anything down, and his goal now was to cement the approach that had been followed. The task began well. He was able to locate and install all of the required software modules, and to get the web server to start without reporting any errors. However images weren't loading and web-pages looked funny when loaded in the web browser. After "poking around in the dark" for an hour or so, and becoming increasingly annoyed and confused, he was able to narrow the issue down to one of configuration, and to copy the necessary setting from the previous project's files. He was relieved, until he loaded a page that he expected to work and a different error message appeared. At that point, he stepped away from the computer and made a cup of tea.

The story recounted by Evan is unique among the interviews in the rich perspective it provides about things that go wrong as they occur. His is a vivid and sometimes harrowing account of "thrashing," described in the Bell Labs study as an ineffective process of going over and over a problem. He is aware that his approach was unsystematic, flawed and risky, calling it a process of attrition, and noting at one point an awareness that "if I plugged the dam somewhere it was going to burst somewhere else". At other points he described his approach as "basically experimenting" and "hacking around." He is also aware that he is a novice. In fact it is because he is "wary of screwing things up" that he has undertaken this task in the first place.

External factors may have contributed to his difficulty. He was working on a home computer, tunneling into a virtual machine hosted on his machine in the office. He became confused in switching between environments, turned around about what he had done, "what I'd changed and what hadn't changed."

He faced two obstacles. The first was one of conguration and is noted in the tutorials and documentation of the open source framework to be tricky, suggesting that other developers have encountered similar difficulty. The second involved a dependency between modules of the framework which could only be resolved by downgrading one of the two to an earlier version. Both of these issues had been encountered in an earlier installation of a local environment for a different project nearly two months prior, before a holiday break.

Thrashing is described as a negative novice behaviour in the Bell Labs study. Novices were found to fail to recognise that thrashing is happening, and to be unable to break out of it. The reports suggested that experts might thrash, but that they were able to attend more quickly to emotional cues that they were doing it, and to seek help sooner from colleagues with greater expertise.

By contrast, in Evan's story, thrashing is not only "annoying and confusing" it is also useful, because it forces him to take a closer look at the software he is using and building. As he puts it:

> "You know this is quite informative 'cause obviously you would get something and it would work out of the box and you don't really think about it again, so even though this was an annoyance, it was quite useful to actually have to look into those relationships."

Evan was also able to recognize when he had reached the limits of his knowledge and experience. His description of the strategy he employed at this stage is insightful:

> "I'd spent long enough messing with the configuration files. I realised either it wasn't there or I'd broken it completely. Let's get it back to how it was - you know I think you take a step back and you think okay it should be working the way it is so let's move on to the next thing and try and understand."

In the end, Evan gets everything working, but his confidence in the solution is not high. He is not seeing any error messages or funny behaviour which suggest to him that everything is working now, "touch wood". He considers the day to have been a "personal failure," but he is more confident when describing how his knowledge of the application framework has grown:

> "I'm comfortable with creating that environment, I'm comfortable with getting up and running and also I'm much more aware of creating something that's got a bit of longevity."

5.2 Tolerating: I wouldn't say "cropped up". I expected to see the error.

> Valentin describes an issue that surfaced as a bug several times over the course of nearly two years, in tools used by the developer and in different areas of the software being developed. The issue was related to the use of Unicode, which presented particular complexities when introduced to the domain. Though Valentin has years of commercial software development experience and considers himself to be well-versed in this standard, a solution was not immediately apparent. He explains that the first occurrence of the issue in the organisation's documentation wiki prepared him for later manifestations. These he managed by "setting aside the complexity" of the problem and by making assumptions about users' environments. At two stages, he implemented temporary solutions, a strategy that allowed him to concentrate on fulfilling more important requirements for the project, and to analyse the problem "in the background". In the end, a permanent solution was found with the help of a colleague who is more skilled in client-side development.

Valentin's story is striking in its clarity. Though this particular situation is new to him, he is an experienced developer, and this is reflected in the way he describes organising his work:

> "I didn't want to be in the situation where I'm approaching deadline, a phase where we have to do a demonstration or release this on the live website and I have to find a solution in just a very limited time for a problem I've never encountered before. So I'd rather prepare the thinking and explore things in different directions to be sure that I will be ready for that."

The findings of the Bell Labs study relate specifically to expert debugging behaviour, however they resonate at points with comments made by Valentin. That study found that expert debuggers think extensively about problems before taking action. For Valentin, this takes the form of partial, temporary solutions. As with expert debuggers, Valentin provides evidence that he uses the time to find information about what to try next, he plans and prepares before taking action.

The story recounted by Valentin, like the Bell Labs reports, also includes evidence of sophisticated interpersonal skills in interacting with domain experts. He used feedback from domain experts to prioritise and plan when the issue appeared in an unforeseen area of the software. This occurrence made him realize that the problem was more widespread than he had thought, and also clarified for him its importance to the domain experts. He notes that he felt pressure at this point to identify a strategy for addressing the issue:

> "I had to say something, to tell them that I have a strategy, not necessarily a solution, but a strategy."

Valentin is confident and pleased with the ultimate solution, describing it as "very clean" and "well established". However, he realises that if he had not had the help of a colleague, he may have had to accept an inferior alternative. He is also keenly aware of how his own limits contributed to the issue. Though he had a superficial awareness of the solution that was adopted, he admits that his knowledge was lacking and that he "hadn't done his job" at keeping up with user interface development.

5.3 Compromising: I'm just not that happy with it yet

> Joaquim described fixing a recently reported bug in a tool he is building to support detailed editorial work on medieval manuscripts. Recreating the flawed behaviour was tricky because the conditions under which it occurred were not accurately reported, and triggering the bug required performing an unplanned-for action multiple times. However the fix itself was trivial, involving altering basic conditional behaviour in a single function. Now he has a produced a solution which is meeting requirements. He is not satisfied, however, explaining that he is not "still not very happy with it yet," and that he is not sure how well it is working. He is aware that his understanding of how the open source library he is working with is still developing. He is cautious, describing the library he has found as "not the right way, a better way". This caution extends into the software, causing Joaquim to hold back on promoting the function where the bug was located into the general API he is designing.

The details of Joaquim's story are less emotionally vivid than those given by Evan. He, like Valentin, has spent a number of years developing software in commercial and academic environments, and of the three developers, has spent the most time in this organisation. However his account included less evidence of overt strategy than that of Valentin, perhaps because his story was collected in the midst of his work on the project. The details he recounts suggest that at times even experienced developers make simple mistakes and settle for solutions with an intuitive sense, rather than a rational understanding that they are flawed.

Though most strongly exemplified in Joaquim's story, evidence for this theme was also provided by Evan and Valentin. It also surfaced in accounts given by James, Richard and Marisa. These developers did not recount incidents so directly connected with writing software but, like Joaquim, were interviewed in the midst of longstanding work on a project.

Compromise was related in terms of an intuitive sense of how well software is functioning, a sense which at times contradicts cues given by the software. Joaquim has achieved a working solution in his design, but he is dissatisfied and still feels that something isn't right, that it may not be working well enough. Evan has no information that things aren't working, but is still wary. Everything is working now "touch wood".

Closely related to this, informants conveyed that they had settled for a sub-optimal solution in aesthetic terms. Evan has gotten his application framework to run, but is aware that it is "pretty dirty". Joaquim described working with event handling in web-browsers first as "not nice," and later explained a particular event handling scenario as a "bit of a disaster". James describes his refactoring work as "good enough", but notes that he would have liked it to be a "a little prettier".

Compromising is informed by past events, and shapes future expectation. Evan expects that he will have similar kinds of problems when he promotes his software to a different environment. Valentin reported that he expected to see occurrences of the rendering error based on past occurrences. As he described it, "this [the first occurrence] prepared me for that". Richard doesn't think he has achieved the perfect data model, but isn't sure what exactly might be wrong. Even though his project is behind schedule, he feels that they have "moved too quickly" and as a result are going to have to settle for a less than "ideal" model.

Valentin suggests that intuition can dissuade developers from compromising, describing an unused alternative solution as "very ugly," something to be used only as a "last resort". Similarly, Marisa recounts nearly giving up, before trying again to find a solution for a respected project partner:

> "I didn't want to go and say... It's impossible. Trying to find something that was maybe not everything that they wanted but better...You don't give up."

6 Limitations

Stories were collected from a single organisation, and may not represent software development in different sectors, or in organisations with different work practices. As these stories were gathered retrospectively, it is possible that details were forgotten or distorted[3].

Indeed, the accounts given yielded rich evidence for thrashing tolerating and compromising, but cannot fully explain them. Was Evan's story of thrashing solely the result of his lack of experience? Was Valentin's story an unambiguous example of an expert programmer at work? Joaquim's story suggests that there may be more to both stories. Of the three, his is the only story collected while he was in the midst of a long development arc. He is the developer with the most experience in the organisation and yet the bug he fixed was trivial, and might have been interpreted as a novice error. These facts and the way he describes settling *for now* suggest that the perspective developers hold when stories are collected is important. Gathered early in a development or learning process, errors may lend themselves to novice explanations. Gathered completely after the fact, decisions related to things that go wrong may be reported as strategic. However when they are gathered in the midst of work, a murky middle area emerges that deserves closer examination.

All of the themes suggest that errors influence and inform development work, however the theme of compromising in particular indicates that developers use intuition to identify and manage flaws in the software they are building. This sense is described in terms of doubt, confidence, and satisfaction. However, it is not clear to what extent or in what ways they use this sense while work is happening.

Retrospective elicitation also cannot explain how individual perspective might obscure details of how thrashing, tolerating and compromising co-occur in a single development process. Joaquim's original implementation took less than an hour. The bug may have resulted in part from an earlier, hasty decision to get it working. Similarly, though Valentin reports having consciously implemented partial, pragmatic solutions, this strategy crystallised in response to interactions with project partners and managers in which the priority of the issue was emphasised to him. This suggests that even though the story was recounted as strategic, there may have been moments in which he settled for rather than tolerated interim solutions. Neither Valentin nor Joaquim report periods of thrashing like those reported by Evan. However, Joaquim does report a moment of confusion about the source of the bug and Valentin reports being surprised by a manifestation of the issue in an area of the software he had forgotten about.

7 Conclusions

The critical decision method was used in this study to elicit rich accounts of issues faced in one kind of software development. The analysis examined things that go wrong beyond bug fixing, and emphasised the perspective of the developer toward problems. This analytical perspective has yielded three insights. First, it provides a detailed look at what happens when a developer thrashes, and suggests that thrashing may hold value for the developers who engage in it. It includes evidence of expert behaviour that is consistent with related studies, but extends the applicability of such findings to development activities other than bug fixing. Finally, by considering development activity more generally, a nuanced view of how flaws are intuitively understood and managed over time emerges. This view might be explored in future research by examining how a developer's perspective toward particular choices changes in the course of work on a project.

8 Acknowledgements

We thank the software developers who shared their experiences with us. This research is supported in part by ERC grant AdvG 291652-ASAP (Nuseibeh) and SFI grant 10/CE/I1855 (Nuseibeh).

References

1. Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 91–120. Springer Boston, 2004.
2. Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, 1984.
3. B. Crandall, G.A. Klein, and R.R. Hoffman. *Working minds: A practitioner's guide to cognitive task analysis.* The MIT Press, 2006.
4. A. Endres. An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Reliable Software*, pages 327–336. ACM, 1975.
5. Jared T. Freeman, Thomas R. Riedl, Julian S. Weitzenfeld, Gary A. Klein, and John D. Musa. Instruction for software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 271–282, London, UK, UK, 1991. Springer-Verlag.
6. M. Leszak, D.E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *The Journal of Systems & Software*, 61(3):173–187, 2002.
7. D. Perry. Where do most software flaws come from? In A. Oram and G. Wilson, editors, *Making Software: What Really Works, and Why We Believe It*, pages 453–494. O'Reilly Media, Inc., 2010.
8. D. Perry and C. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 48–67. Springer-Verlag, 1993.

9. D.E. Perry and W.M. Evangelist. An empirical study of software interface faults. pages 32–38, 1985.
10. Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface faults — an update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, volume II, pages 113–126, January 1987.
11. Brian Randell. On failures and faults. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 18–39. Springer Berlin / Heidelberg, 2003.
12. Thomas R. Riedl, Julian S. Weitzenfeld, Jared T. Freeman, Gary A. Klein, and John D. Musa. What we have learned about software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 261–270, London, UK, UK, 1991. Springer-Verlag.
13. N.F. Schneidewind and H.-M. Hoffmann. An experiment in software error data collection and analysis. *Software Engineering, IEEE Transactions on*, SE-5(3):276 – 286, May 1979.
14. Julian S. Weitzenfeld, Thomas R. Riedl, Jared T. Freeman, Gary A. Klein, and John D. Musa. Knowledge elicitation for software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 283–296, London, UK, UK, 1991. Springer-Verlag.
15. D. Zorich. *A survey of digital humanities centers in the united states.* Council on Library and Information Resources, 2008.

Conducting Field Studies in Software Engineering: An Experience Report

Rebecca Yates

Lero - the Irish Software Engineering Research Centre
University of Limerick, Ireland
rebecca.yates@lero.ie

Abstract. Involving practitioners in software engineering research is crucial for relevant, applicable results, but finding willing participants can be difficult. This experience report describes some of the issues that can block or mar field studies of software engineering, and suggests tactics to avoid or mitigate these problems.

1 Introduction

Field studies of software developers at work are important for generating externally valid software engineering research results. Unfortunately, persuading software practitioners to participate in research projects can be difficult, and conducting the studies themselves has many potential pitfalls.

This paper reports on the experiences from a study of software engineering onboarding (i.e. their period of familiarisation with the company, the development team and the unfamiliar code and tools) (Yates 2011). This project collected recordings of onboarding sessions using a camcorder and screen capture software, interviews with the new developers, and questionnaires to capture the participants' background as developers. Out of around 70 approaches to a variety of companies, eight were willing to participate in some way.

The lessons learned from setting up and conducting the study are discussed here in the hope that this information will be of use to others designing similar field studies. Please note that the interpretations of events are only my opinion.

2 Finding participants

Having designed a field study, the first stage is to find willing participants. In this section, I discuss methods for reaching potential participants, understanding their concerns about participation, mitigating those concerns, and common responses from contacts.

2.1 Contacting developers

Personal contacts The most successful way to find participants was through personal contacts in industry, academia and other groups. The majority of the field studies were set up via friends and colleagues. As with job-hunting, spreading the word about the search can lead to useful introductions, which in turn can lead to field study opportunities.

University-friendly companies Some companies contact favoured universities as part of their recruitment drives. Companies who recruit in this way are more likely to be positive about academia in general and the institution in particular, so this is a good source of potential participants. As my research required software teams that were taking on a new member, I responded to recruitment emails sent to the university's computer science department. This strategy was occasionally successful.

Social networking Social networks (such as Twitter, LinkedIn and mailing lists) can be used to broadcast requests for participation. A colleague reported some success with this technique for a low-involvement request (completing an online survey), but it did not generate any responses to my more involved proposals.

Careers fairs Careers fairs provide a opportunity to meet face-to-face with company representatives. However, the representatives are typically either HR staff (who may pass on details of the research but are unlikely to be enthusiastic themselves) or junior developers (who may be enthusiastic but have little influence within their organisation). I had hoped that careers fairs would allow me to contact companies preparing to take on new hires, but this strategy did not prove a fruitful source of participants. If this approach is used, it is important to be clear that one is not actually looking for a position with the company, or the conversation can become very confusing.

Participant recommendations Initial participants can be asked to recommend others who might be willing to participate. I did not try this technique, but based on the comments I had after the data collection, I believe that it would be fruitful with participants who found the experience positive. In the case of this observational study, participants often commented that the involvement was much less onerous than expected, required very little of their time, and generated some valuable introspection; with these assurances, the recommended contacts might be more inclined to participate.

2.2 Common concerns

Software developers and their managers are likely to be enthusiastic about research projects that address problems they are living with, or opportunities to "show academia what the real world is like". However, across all organisations, two major concerns prevent developers becoming involved with field studies: time pressures and security concerns. Addressing these issues upfront may turn an automatic 'no' into a constructive negotiation.

Time pressures While managers and developers may be broadly enthusiastic about a research project, often their major concern is that involvement will take up time (both during the actual data collection, and the overhead of arranging permissions, NDAs and so on). With many software development projects operating under tight deadlines, or already running late (see Perlow (1999)), any extra distractions may be considered unacceptable.

It is important, therefore, to be very clear about how much time the research will take up on top of the practitioners' everyday work. Project managers will require time estimates for surveys and interviews, and a list of actions including administration (such as NDAs and visitor passes) which will also take time to complete. If the bulk of the data collection is fly-on-the-wall observation of developers' normal activities, and therefore does not take time away from work, emphasising this may help to persuade contacts that participation presents little risk to their schedules.

Intellectual property Protecting intellectual property is sometimes a concern for potential participants (albeit not usually the primary concern). A non-disclosure agreement (NDA) is often enough to allow the research to proceed. Large corporations typically have dedicated legal teams and standard procedures for creating NDAs, while very small companies may not require NDAs at all (though ethically, the data and the participants must still be protected to the same degree). Medium-sized companies may struggle with this because participants see the need for IP protection but lack the resources or procedures to set this up. In all cases, NDAs may take longer than expected to set up, which may impact the research schedule.

IP is less of a concern when software is not the company's main line of business. It may be more comfortable for (for example) a materials supplier to allow access to its in-house stock management system, than for a software development company to permit published research on its flagship software product.

Project-specific issues The details of the project may introduce extra constraints on the search for participants. My research project focussed on onboarding, so I sought out software teams that were planning to introduce a new developer to their team in the near future. Many potential participants were enthusiastic about the project but unable to help because no new recruits were joining. This was sometimes even the case when the company was actively recruiting, because finding the right person could take many months.

2.3 Persuasion

A contact who is in a position to approve the research will have a set of questions about the requirements and implications of getting involved; these questions are often common to all organisations. However, if this person is not the initial point of contact, it can be difficult to communicate these details through a chain of command. Misunderstandings may lead to difficult situations where managers agree to participate, only to discover that they have actually been asked for something very different.

To mitigate this problem, the purpose and requirements of the research can be provided to the organisation on a single-page PDF (see appendix for a sample document used in my research). This provides a quick and accurate summary of the proposed research, which can be passed through an organisation from the initial contact to the correct person. The content of the information sheet evolved in response to frequently asked questions and occasional feedback.

This experience suggested that the most important points to be clear about were:

- The purpose of the research.
- The actions required from the participants and others in the company.
- The timescale of participation (in particular, any time taken away from work).
- The ways in which the collected data would be used and protected.

2.4 Responses

Common responses include:

That sounds interesting, but no. This is very common.

Sorry, we don't participate in research at Company X. This may simply mean that this contact is not interested in the overhead of organising participation. It may still be possible to conduct a study at this company via another, more enthusiastic contact.

An initially positive response, and then nothing. This may indicate that the company is too busy to participate, but occasionally a polite reminder will allow the conversation to move forward.

No response at all. This is the most common outcome. Occasionally, out of the blue they will reply months later (in one case to let me know that the NDA was set up and they were ready to help).

We would like to help but we are not in a position to do so. My study had the additional requirement that participating companies had new developers joining their teams. It is occasionally worth contacting them again after several months to see if the situation has changed.

Yes, we'd like to help, let's set up a meeting... It does happen.

Given the number of contacts (including multiple contacts for large companies) I found it useful to maintain records of contacts, particularly to highlight those that gave a positive response but had not arranged to participate. A spreadsheet function can show the number of days since the last communication, to ensure that timely reminders are sent and potential participants are not lost.

3 Technical issues on-site

Having negotiated access to a company, it is important to make the most of the opportunity and avoid wasting the participants' time. Most of the potential pitfalls can be avoided through good preparation.

Ambient noise Ambient noise can pose a problem for audiovisual recording. Open plan offices can be unexpectedly loud, and meeting rooms can have air-conditioning that nobody knows how to control. Even areas that seem quiet at first may be used at breaktimes for coffee, loud conversations and games of pool and pingpong. Depending on the nature of the field study, this may be unavoidable, but if the study does not require the participants' usual working environment, the participants may be able to book a quiet room for the study.

Some microphones are directional and can be aimed to capture more of the desired sound. Getting to know the audio equipment's capabilities beforehand will allow better recordings to be made. Depending on the purpose of the recordings, ambient noise can be frustrating, but is rarely a major problem.

Software installation and use As companies grow larger, they are more likely to hand control of their IT systems to dedicated teams. This can be an issue if the research requires the installation of any software (for example, screen capture or task recording). The IT department may have prevented the installation of any new software on participants' machines.

Occasionally, participants already have suitable software on their machines (for example, screen capture is currently available as part of QuickTime on Snow Leopard). If not, participants can be asked to preinstall the software, allowing them time to negotiate permission with their IT departments if required.

An alternative approach is to bring standalone ("portable") copies of the required software on a USB key (these do not require installation). In this case, the participants' OS may not be known in advance, so it is worth bringing different versions of the software to cover all eventualities.

It is very important to minimise the risk of negative effects of participation. Providing software is a risk because of the possibility of introducing viruses or other malware. Make very, very sure that any software you ask them to install or use is safe.

Running extra software on the participant's computer may cause it to slow down, impacting their interactions with it. In the case of screen recording software, the performance impact can be minimised by reducing the video quality and preventing unwanted audio recording.

Power sockets Power sockets can be a surprising issue. In many offices, every available socket is in use, or the only available socket is on the other side of a walkway. The activity to be recorded may take place away from any power sockets, and participants may move from place to place (e.g. to access whiteboards or demonstrate hardware). Again, familiarity with the recording equipment will prove valuable - for example, knowing how a camcorder will behave if unplugged during recording. It can also be a good strategy to bring an extension cable.

Data size Depending on the nature and length of data capture, data files may become very large. This creates several issues.

Data files can be kept small in the first place by choosing appropriate recording quality settings. Some screen capture software defaults to very high quality settings, and also records audio which may not be required.

During long sessions, the recording device may fill up before the participants have finished. Knowing the quality settings and the hardware specifications, it is possible to calculate in advance how much storage will be required (or how much of a session can be recorded at once). It may be possible to take advantage of coffee breaks to swap storage devices (e.g. spare memory cards) or transfer data files into storage, but this can be a lengthy process.

Data storage can be a particular problem if recording software is in use on the participant's computer and they do not have enough free space available. In this case, one option is to save the file directly to an external drive (this also avoids lengthy data transfer times after the recording session). An unexpected issue in this study was the 4GB file size limit on older Windows machines; files larger than this become corrupted and unusable.

Questionnaires As part of my study, I asked participants to complete an online questionnaire about their background as developers. Despite testing the questions with colleagues beforehand, some of the questions were difficult for participants to answer accurately. Examples include the question "how long have you worked at this company?". This apparently straightforward numeric question was difficult to answer for the developer who worked for the company as a contractor, then worked elsewhere, and then returned to the company as an employee. A free text box often allows more intelligent answers.

In another scenario, the questionnaire had to be hastily adapted because a non-developer became involved in the recordings and the questions were worded with the assumption that all participants would be developers.

4 Ethical issues of collaboration

Sharing data It is important to be clear with the participants and the company about who the recordings or other data will be shared with. Typically, recordings should not go beyond the participants and the research group, but occasionally other employees in the same company may ask to see or hear them. This could be an issue if the participants, believing the data collection to be private, have been discussing other employees or office politics, or making jokes that could be misinterpreted. Sharing the recordings could have negative consequences for the participants; this must be avoided.

Participant consent Information does not always flow as freely through corporations as one might hope. It is entirely possible to set up a field study with a manager, and arrive to discover that the participants themselves have not been informed or consulted about the planned research. This presents the ethical issue of informed consent; participants may then feel pressured into participating.

This scenario could be avoided by requiring potential participants to complete an online survey or other task before the recording session. This would ensure that they were aware of the field study and able to discuss any concerns beforehand.

Unexpected participants During a field study, it is fairly common for extra employees to interact with those who have agreed to participate in the research. A pre-prepared explanation and spare participant consent sheets provides the option for them to join the study, if they wish, without major disruption to the session.

5 Thanking participants

It is often recommended to "give something back" to the companies involved in the form of presentations about the research. So far, not a single company has taken me up on this (probably because of the time pressures mentioned earlier).

As thank-you gifts, physical Amazon gift vouchers were appreciated, but electronically sent vouchers were often mistaken for spam. A thank-you card and a box of chocolates or biscuits were appreciated in office environments.

6 Related discussion

Further discussions on ethical issues can be found in the special issue of Empirical Software Engineering focusing on ethics, edited by Singer & Vinson (2000), and data collection techniques for field studies are examined by Lethbridge, Sim & Singer (2005).

Experimentation in the field does not always go to plan. Exton, Avram, Buckley & LeGear (2007) provide a discussion of technical and social issues that affected an experimental setup.

7 Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 04/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie). The author thanks Jim Buckley and Norah Power for their comments and advice.

References

Exton, C., Avram, G., Buckley, J. & LeGear, A. (2007), An experiential report on the limitations of experimentation as a means of empirically investigating software practitioners, *in* 'Psychology of Programming Interest Group', pp. 173–184.

Lethbridge, T., Sim, S. & Singer, J. (2005), 'Studying software engineers: Data collection techniques for software field studies', *Empirical Software Engineering* **10**(3), 311–341.

Perlow, L. (1999), 'The time famine: Toward a sociology of work time.', *Administrative Science Quarterly* **44**(1), 57–59.

Singer, J. & Vinson, N. (2000), 'Ethics and empirical studies of software engineering', *Empirical Software Engineering* **5**, 89–91. 10.1023/A:1009859121816.

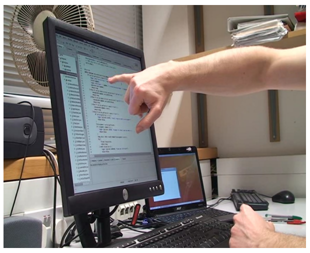
Yates, R. (2011), Expert explanations of software, *in* 'Psychology of Programming Work in Progress'.

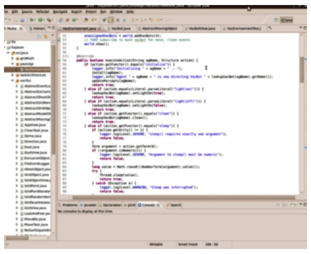
THE IRISH SOFTWARE ENGINEERING RESEARCH CENTRE

Research Proposal

The Issue

- On joining a company, or moving to a new project, a software developer is faced with unfamiliar code and has difficulty contributing. Ramping up to full productivity can take up to six months.
- In research at Microsoft, 56% of developers reported that understanding someone else's code was a serious problem.
- Expert software developers are able to mentor new team members – but the experts aren't always available, and mentoring can be time-consuming.



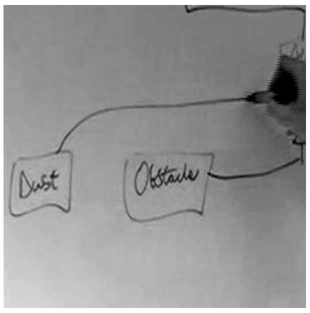


```
        unassignedVacBots = world.addVacBots(4);
        // TODO subscribe to each vacbot for move, clean events
        world.show();
    }

    @Override
    public boolean executeAction(String agName, Structure action) {
        if (action.getFunctor().equals("initialise")) {
            logger.info("Initialising " + agName + " ...");
            install(agName);
            logger.info("Agent " + agName + " is now directing VacBot " + lookupVacBot(agName).getName());
            updatePercepts(agName);
            return true;
        } else if (action.equals(Literal.parseLiteral("light(on)"))) {
            lookupVacBot(agName).setLightOn(true);
            return true;
        } else if (action.equals(Literal.parseLiteral("light(off)"))) {
            lookupVacBot(agName).setLightOn(false);
            return true;
        } else if (action.getFunctor().equals("clean")) {
            lookupVacBot(agName).clean();
            return true;
        } else if (action.getFunctor().equals("sleep")) {
            if (action.getArity() != 1) {
                logger.log(Level.SEVERE, "sleep() requires exactly one argument");
                return false;
            }
            Term argument = action.getTerm(0);
            if (!argument.isNumeric()) {
                logger.log(Level.SEVERE, "Argument to sleep() must be numeric");
                return false;
            }
            long value = Math.round(((NumberTerm)argument).solve());
            try {
                Thread.sleep(value);
                return true;
            } catch (Exception e) {
                logger.log(Level.WARNING, "Sleep was interrupted");
                return false;
```

No consoles to display at this time.

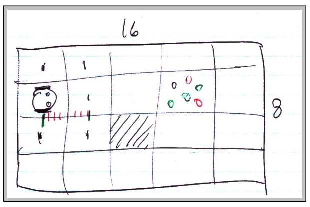
Research Proposal

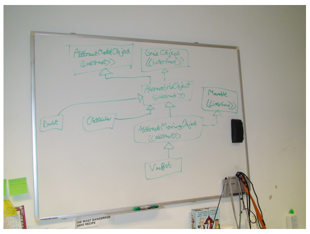
- To address this issue, I'm investigating how experts explain their code to new team members. Through video recordings, screen capture, code analysis and interviews, I'm seeking to understand what information about a codebase is most useful in this situation.
- This is a new approach to the problem. To ensure valid and useful results, it's important to involve industrial partners.
- More information about the data collection can be found on the reverse.

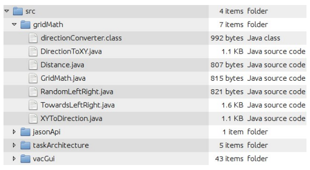


Outcomes

- This kind of research can answer questions such as:
  - Which parts of the code are important for a new team member to know?
  - How are diagrams used to convey information?
  - How useful was the explanation to the newcomer?
  - How could the ramp-up be improved?
- I hope that this research will lead to recommendations for mentoring techniques or tool support for new team members, reducing ramp-up time and increasing developer productivity.







```
▾ src                               4 items    folder
  ▾ gridMath                        7 items    folder
      directionConverter.class      992 bytes  Java class
      DirectionToXY.java            1.1 KB     Java source code
      Distance.java                 807 bytes  Java source code
      GridMath.java                 815 bytes  Java source code
      RandomLeftRight.java          821 bytes  Java source code
      TowardsLeftRight.java         1.6 KB     Java source code
      XYToDirection.java            1.1 KB     Java source code
  ▸ jasonApi                        1 item     folder
  ▸ taskArchitecture                5 items    folder
  ▸ vacGui                          43 items   folder
```

Images from the pilot study at the University of Limerick

If you have any further questions, or would like to participate, please contact me: (rebecca.yates@lero.ie). Thank you for your time.

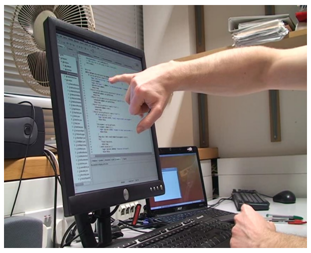
Rebecca Yates is a PhD student at the University of Limerick, Ireland, supervised by Jim Buckley and Norah Power. Prior to this, she worked as a software engineer.

THE IRISH SOFTWARE ENGINEERING RESEARCH CENTRE

What's involved?

Thank you for considering helping with this research into expert explanations of software. This page details the steps typically required for participation.

Beforehand:



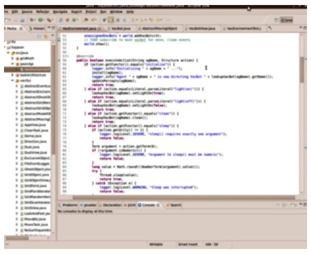
We arrange a time for the data collection session.

I am most interested in a newcomer's first introduction to the codebase. The walkthrough typically takes between one hour and half a day. Newcomers may be new hires, or existing employees moving to a different project within the company.

We set up an NDA if required.

This can be supplied by your company or created by the University of Limerick.

On the day:

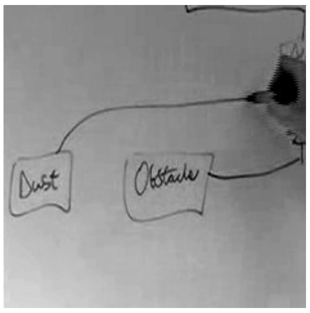


The participants read and sign the Participants Rights document.

This document covers their rights regarding anonymity, their right to pause or abandon the recording for any reason, and describes how the collected materials will be used.

We install screencapture software on their machine.

e.g. the freely available CamStudio (unless a suitable program is already installed). This is because the text on screen doesn't show up clearly on the camcorder.



I place the camcorder and start recording.

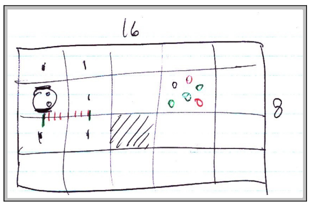
This is a small camcorder on a desktop tripod, and is pointed at the screen. It's unobtrusive, and can be moved easily to follow the conversation to whiteboards or other locations.

The participants work as usual, with the expert explaining the code to the newcomer.

I move the camcorder if required, and don't interfere with the work in progress.

After the session, I take a photo of any diagrams that were used.

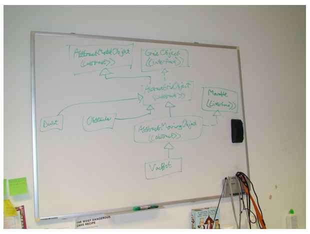
Like the screencapture, this just ensures a clear copy in case it's not readable on the video.



I analyse the source code under discussion.

This is for the quantitative part of the research, comparing the path of the explanation to the code metrics. All copied materials are stored encrypted using ecryptfs and are not released; if code cannot be removed, on-site access would suffice.

Follow up:

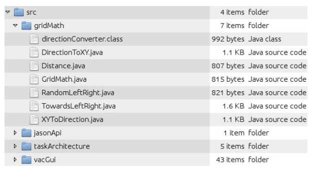


The participants complete an online questionnaire.

This is a short (~10 minutes) questionnaire on their background as software developers.

The newcomer gives a short interview.

This takes place around three weeks later, and asks how useful the explanation was. This can be completed in person or by phone and should take less than half an hour.



```
src                          4 items    folder
  gridMath                   7 items    folder
    directionConverter.class 992 bytes  Java class
    DirectionToXY.java       1.1 KB     Java source code
    Distance.java            807 bytes  Java source code
    GridMath.java            815 bytes  Java source code
    RandomLeftRight.java     821 bytes  Java source code
    TowardsLeftRight.java    1.6 KB     Java source code
    XYToDirection.java       1.1 KB     Java source code
  jasonApi                   1 item     folder
  taskArchitecture           5 items    folder
  vacGui                     43 items   folder
```

Optionally, I can provide a report or presentation on this research.

To respect the privacy of the participants, this is a report on the research in general, and does not name any individuals or companies. As detailed on the previous page, this report would focus on parts of the code base that are important to new team members, how diagrams can be used effectively, the usefulness of different types of information and general improvements to the ramp up process.

If you have any further questions, or would like to participate, please contact me: (rebecca.yates@lero.ie). Thank you for your time.

Images from the pilot study at the University of Limerick

Paper Session 5

Learning to Program

Comments on the papers from the session on Learning Programming

Benedict du Boulay
Human Centred Technology Research Group
University of Sussex
b.du-boulay@sussex.ac.uk

The four papers in the session on Learning Programming cover differing areas of the topic, though they are united in that they report work that is very much in progress, and in two cases explicitly concentrate on a future research agenda (Bednarik and Orlov, Major et al.) rather than on reporting results. Two of the papers are concerned with novices learning their first programming language (Bornat et al., Major et al.), one looks at novices learning their second programming language (Mselle) and one focuses on differences between novices and experts in terms of their gaze patterns (Bednarik and Orlov). Two of the papers are essentially concerned with pedagogy (Mselle, Major et al.), one with code understanding (Bornat et al.) and one with perception (Bednarik and Orlov).

Mselle argues for the value of a paper and pencil notation, MTL, that can be used to describe the sequence of memory states of a notional machine as it executes small programs. The notation is not generated automatically but is drawn by hand, either by teachers of programming or by learners getting to grips with understanding the way a language that they are learning works. Mselle reports an experiment in which undergraduates used the notation to help them learn their second programming language but did not go to lectures. The end of course results for these students were compared with those who went to lectures and used the usual text book. There were a number of flaws in the experiment, as Mselle himself points out, but the results provide some support for the value of MTL. For me the big question centres around the best way that such a notation should be used: alone or in pairs or groups? In comprehension or in debugging tasks? Instead of lectures and lab classes or as addition to them? Reasoning from code to MTL diagrams, as in the paper, or from MTL diagrams to code? There are obvious links to the work of Bornat, Dehnadi and Barton in that MTL could provide a notation for expressing (possibly incorrect) beliefs about sequence and assignment, see below.

Bornat, Dehnadi and Barton explore two issues[1]. First is the cumulative nature of first year university computer science curricula and the way that success or failure with the customary initial concepts of sequence and assignment has consequences for getting to grips with later and more complex coding concepts. The second is the hypothesis that a prerequisite for learning to program is an ability to see a computer program as a small machine that executes its instructions in a consistent way, even if that ability is based on an incorrect version of that consistency. They describe a test that goes some way towards distinguishing those who think consistently about sequence and assignment from those who do not and looked at the consequences for each kind of student on future success.

[1] I need to declare an interest here, as I am currently working with Bornat and Dehnadi on these issues.

Their work has received criticism, but the basic notions about the consequences of the cumulative nature of learning to program, and the necessity to be able to understand how code is actually executed are clearly important. While they are not directly concerned with pedagogy but with understanding, Mselle's MTL notation might well provide a tool to assess understanding as well as to elucidate it.

The paper by Major, Kyriacou and Brereton sets out a plan to test the hypotheses that learning to program through the use of robot simulator is an effective method and improves learners' perceptions of programming. There is a long history to the idea, going back to the early 1970s, that program code can be externalised for pedagogical purposes though driving either a real robot (such as the Logo Turtle) or a simulated one (such as Karel the robot for Pascal) or characters in a game (such as in Alice) and much has been written on the issue covering both the cognitive and the motivational. While the specific robot simulator that they have in mind may be up to the minute, the big questions remain as they were before. Will students be more engaged and willing to exert effort to drive this simulator to do what they intend than the other problem-solving environments that have been developed? Will the basic concepts of programming -- sequence, assignment, loops, conditions, procedure calls and so on -- be learnt more easily and accurately? Will the three main systems be easy for learners to disentangle – the robot simulator and its behaviour, the programs and their execution that drive the robot, and the underlying computer providing file storage, editing and other tools in support of the first two.

Bednarik and Orlov are concerned with understanding more about the way that human perception works, particularly in the case of complex, multi-window displays. Various techniques have been used in the past, including eye tracking as well as blurring out parts of the image, to get an idea of where programmers look while using complicated graphical interfaces and how important peripheral vision is in such tasks. Bednarik and Orlov set out a research agenda exploring how eye tracking and peripheral vision interact to enable the programmer both to see in detail what they are paying attention to, as well as to understand enough of the visual context to gain supportive information from it. This might have practical applications in that a system might be able to save time and power by not computing parts of the interface that the programmer will not (or cannot) pay attention to. However the main outcome is more likely to be in increasing our understanding how people conduct problem-solving using complex graphical interfaces. Just as in the work on Air Traffic Controllers, Ships' Bridge Commanders, and London Underground System Controllers, simply adding more information does not always lead to better performance and fewer errors. Oftentimes it's redundancy of information and (off screen) fail safe systems that win the day. There ought to be a productive coupling of these two authors' work on perception with Green and others' work on the cognitive dimensions of notation.

Teaching Novices Programming Using a Robot Simulator: Case Study Protocol

Louis Major Theocharis Kyriacou Pearl Brereton

School of Computing and Mathematics
Keele University, UK
l.major@keele.ac.uk t.kyriacou@keele.ac.uk o.p.brereton@keele.ac.uk

Keywords: POP-I.B. programmer education, POP-V.B. research methodologies, POP-VI.E. Computer science education research

Abstract

This protocol provides details of a case study design that will investigate the use of simulated robots as introductory programming teaching tools. This research is motivated by the results of a Systematic Literature Review which indicated that such work would be valuable. The protocol will help to ensure that a reliable, transparent and rigorous study is performed. Furthermore, potential problems have been considered and accounted for in advance of its implementation. The protocol may also act as a point of reference for other researchers interested in performing a case study.

1. Introduction

In this paper a protocol for a planned case study is presented. Case studies are empirical strategies for research which involve an investigation of a phenomenon using several sources of evidence. This case study will investigate the use of simulated robots as introductory programming teaching tools. The research has been influenced by the results of a Systematic Literature Review (SLR) which indicated that such work would be valuable. A range of participants will be involved in the case study including novice programmers and trainee high school teachers. Data collected during the study will be used to evaluate the effectiveness of a robot simulator, and associated workshop, which have been developed to support the learning of introductory programming. This research aims to contribute to knowledge by addressing the findings of the SLR. Moreover, this is the first case study to examine the implementation of a robot simulator in such a context.

The remainder of this paper is organised as follows. In Section Two information relating to the background of the research project is presented. Section Three provides information about the design of the case study. Section Four offers an overview of data that will be collected while Section Five provides details of how this will be analysed and interpreted. In Section Six measures which have been taken to ensure the validity of the case study, in addition to a consideration of potential limitations, are outlined. This is followed by a summary in Section Seven.

2. Background

Learning to program a computer is a difficult task for novices (Kelleher and Pausch 2005). Various efforts have been made by educators to overcome such a problem by implementing active learning environments (McGill 2012). This has included using robots as teaching tools (Fagin 2003, Lauwers et al 2009, Martin and Hughes 2011, McWhorter and O'Connor 2009). The work that is presented is motivated by the results of a SLR which investigated the use of robots in such a manner (Major et al 2011a, Major et al 2012). In total, 36 papers were accepted in the SLR. Of these: 25 examined the effectiveness of physical robots, seven the effectiveness of simulated robots and four the use of physical and simulated robots together. 26 papers (75%) report robots to be effective when used to teach introductory programming. However, the potential to further investigate the use of robots remained, particularly in regards to simulated robots. This is because the quality and rigour of the seven papers related to simulated robots was judged to be inadequate as: four offer a ‘lessons learned’ account, or description of an approach, and provide no empirical data (Becker 2001, Buck and Stucki 2001, Enderle 2008, Ladd and Harcourt 2005); one describes the results derived from interviews as being non-generalisable as only four novices were involved (Borge et al 2004); one specifies the use of a questionnaire but presents no quantitative data (Lemone and Ching 1996); one describes the implementation of pilot lessons but does not undertake detailed analysis (Sartatzemi 2005).

As a result of the SLR, and after reviewing educational software guidelines (Squires and Preece 1999, ANSI Standards 2001, Beale and Sharples 2002), a robot simulator and associated workshop material have been developed. The simulator is modelled after a small real world robot called the Mark III[1]. The robot has two actuators and several input devices. The workshop covers the fundamental programming constructs identified in the ACM/IEEE Computer Science Curriculum Joint Task Force Report (ACM/IEEE 2008). This includes: basic syntax and semantics of a higher-level language; variables, types, expressions and assignment; simple I/O; conditional and iterative control structures; methods and parameter passing; structured decomposition. In addition, the workshop also includes an introduction to arrays. The ACM/IEEE recommend a minimum coverage time of nine hours for the fundamental constructs they identify. With the introduction of arrays the workshop will last around 10 hours in total. Java has been selected as the programming language that will be taught as this is used on Computer Science courses at Keele University as it is currently one of the most widely adopted programming languages (TIOBE 2012). The simulator and workshop were piloted with a number of novice programmers in order to validate the procedures and instruments. Other research has also taken place including the hosting of introductory programming sessions involving 23 pre-service and seven in-service ICT/Computer Science teachers. These sessions were used to evaluate an early version of the robot simulator and to determine participant's attitudes towards the teaching of programming. Some of this research is described in Major et al (Major et al 2011b).

This case study will form part of a PhD project. The case study methodology is being used as it is highly flexible and suitable for complicated studies involving multiple human participants. Case studies are strategies for research which involve an empirical investigation of a phenomenon using several sources of evidence (Robson 2002). One strength of case studies is that they are able to provide a deeper understanding than controlled experiments (Runeson et al 2012) whilst remaining capable of achieving scientific objectives (Lee 1989). The development of a protocol helps to ensure reliable, transparent and targeted research which considers potential problems in advance (Yin 2009).

3. Design

In this section information relevant to the case study design is outlined including the aim of the study, propositions, workshop structure, participants, data sources, cases, procedures and roles.

3.1 Aim

This is an exploratory case study as it aims to seek new insights (Runeson et al 2012). Moreover, it is also considered to be a positivist case study as past evidence has been examined (as detailed in the SLR), a range of variables will be measured, propositions will be tested and inferences will be drawn from samples to stated populations (Klein and Myers 1999). This protocol is based on one described by Brereton (Brereton et al 2008). The aim of this study is to determine whether a robot simulator is effective for supporting the learning of introductory programming by using such a tool in a specially designed workshop. The following research question will be asked:

Is a robot simulator an effective tool for supporting the learning of introductory programming?

3.2 Propositions

Four propositions have been developed as a result of the case study aim:

P1 A robot simulator is an effective tool for supporting the learning of introductory programming

P2 A robot simulator improves novice's perceptions of programming

P3 A robot simulator offers a more effective introduction to basic programming concepts when compared to other teaching methods

P4 A robot simulator improves trainee ICT/Computer Science teacher's confidence in their ability to teach introductory programming

[1] http://www.junun.org/MarkIII/

3.3 Data Sources

Several data sources will be used during the case study. The research question and propositions will be addressed as follows:

- By using **questionnaires** to determine participants programming knowledge, in addition to their attitudes towards programming, both before and after the workshop.
- By administering (and later scoring) **programming tests** both during and after the workshop, which have been constructively aligned with the learning objectives of the workshop, in order to determine programming progress.
- By maintaining a **log of events** that occur during the workshop session.
- By **interviewing** three current teachers, who have been involved in the planning of the workshop sessions, in order to determine their thoughts on the effectiveness of the simulator.

3.4 Workshop Structure, Participants and Cases

Two separate workshop sessions have been scheduled which will involve students with limited or no programming experience aged between 16 and 18 years old. Each workshop will last two days (5 hours per day) and will involve 10 and 11 students respectively. The two sets of students are studying at different Further Education (FE) institutions. Two programming tests will be completed by students in order to gauge progress and a pre- and post-workshop questionnaire will be administered. At the end of the workshop students will also complete four programming tasks which are designed to draw together the concepts that they have encountered.

22 trainee ICT/Computer Science teachers will also take part in a workshop that will replicate the FE students workshop discussed above. Two separate two-day workshops will again be held and this cohort of trainees will be split. Whereas the students have had limited exposure to programming these trainee teachers have all encountered programming in some capacity before. Therefore, in addition to undertaking the assessment tasks designed for student participants, trainee’s confidence in their ability to teach programming (in addition to their attitude towards the subject) will be investigated. This will be done by administering a pre- and post-workshop questionnaire. The trainees will also be asked to compare their previous programming learning experience to the one using the robot simulator.

In addition, a further introductory programming workshop has been scheduled. This will involve a cohort of 22 Year 9 High School pupils (aged around 14 years old) who are about to embark on a Computing GCSE qualification and have little or no programming experience. As this group of participants are still enrolled at High School it is not possible to run the 10 hour version of the workshop previously discussed (due to pupils having other time commitments). Instead, a shortened version of the full workshop (lasting five hours) will be delivered. During this modified version of the workshop pre- and post-workshop questionnaires will be used to collect data.

In order to address the research question this study will be a multiple-case case study:

- *Case 1* will be the novice student programmers who are aged 16-18 years old and are currently in FE. The experiences of 21 students will be considered as part of this case. Case 1 participants will take part in the full two-day workshop.
- *Case 2* will be the trainee ICT/Computer Science High School teachers who all have some programming experience. The experiences of 22 trainees will be considered as part of this case. Case 2 participants will take part in the full two-day workshop.
- *Case 3* will be the novice student programmers who are aged around 14 years old and are currently in High School. The experiences of 22 students will be considered as part of this case. Case 3 participants will participate in a shortened one-day version of the main workshop.

3.5 Case Study Procedure and Roles

LM will deliver the introductory programming workshops and will be the case study leader. PB and TK reviewed an early version of the case study protocol and will continue to be consulted as the study progresses providing research support and advice. The case study protocol has also been evaluated by an independent expert (Barbara Kitchenham of Keele University).

3.6 Ethical Considerations and Participant Code Numbers

Data collected from participants will be stored securely in accordance with the Data Protection Act 1998. Real names, raw, analysed and demographic data will not be associated with any participant. Any identifying features arising from the interviews will be removed during transcription. All data will be exclusive to members of the research team. All participants will receive an information leaflet and informed and written consent forms will be received from all participants. All participants will be given a unique code number. This will be written by participants on all data collection instruments. Keele University's Research Ethics Panel has approved the use of the robot simulator, and workshop, for research purposes.

4. Data Collection

Details of data that will be collected during the case study is provided in this section.

4.1 Case 1: Novice Programmers (FE Students)

Pre-Workshop Questionnaire

A paper based questionnaire will be completed by novices before the workshop in order to determine their past programming experience and attitude towards the subject. An overview of the content of this questionnaire is presented in Table 1.

Novice's Past Programming Experience	**Novice's Attitude to Programming**	**Misc.**
Have novices previously programmed	Should programming be taught in schools	Gender
What languages have novices used (if any)	Would novices consider learning to program	
Why novices previously programmed	Problems while learning to program	
Was previous experience challenging	Stereotypes associated with programming	
Enjoyment of previous experience		

Table 1 – Pre-Workshop Questionnaire Overview (Case 1 Participants - Novices)

In-Workshop Programming Exercises

Two paper based programming exercises will be completed during the workshop in order to monitor programming progress. These will draw on concepts that students have encountered.

In-Workshop Researcher Log

The lead researcher (LM) will keep a personal log of incidents or issues that occur during the workshop session according to pre-determined criteria.

Post-Workshop Questionnaire

A second paper based questionnaire will be completed by novices after the workshop in order to gauge their thoughts on the workshop experience. In addition, novices will again be asked about their attitude towards programming. Table 2 provides an overview of this questionnaire.

Novice's Workshop Programming Experience	**Novice's Attitude to Programming**	**Misc.**
Enjoyment of session	Programming plans going forward	Gender
Difficulty of session	Has simulator changed perceptions	
Thoughts on effectiveness of simulator	Has simulator dispelled any stereotypes	
Most/least liked aspects of simulator (up to three)	Would novices consider learning to program	
Thoughts on effectiveness of workshop	Should programming be taught in schools	
Comparison of previous programming learning experience (if any) against the workshop		

Table 2 – Post-Workshop Questionnaire Overview (Case 1 Participants - Novices)

Post-Workshop Programming Exercises

Novices will complete four programming challenges that have been constructively aligned with the learning objectives of the workshop session and draw on the concepts encountered. At least 30 minutes will be required for this. These exercises will determine whether *deep learning* has taken place. Case (Case 2008) describes deep learning as when students aim towards understanding whereas *surface learning* is where students aim to simply reproduce material in a test or exam without actually understanding it. Code will be collected and graded according to a three point scoring system:

A) Participant's code shows evidence of deep learning as knowledge gained during the workshop has been used to critically solve a new problem. At least 80% of code is correct.

B) Participant's code shows some evidence of deep learning as the new problem has been attempted and successfully solved in part. Between 50% and 80% of code is correct.

C) Participant's code shows no or little evidence of deep learning as no or little attempt has been made to solve the problem. The participant may have not differentiated between general principles and examples. The participant may have simply tried to repeat information from memory or has merely copied previous code without trying to adapt it to solve the new problem. Less than 50% of the code is correct.

4.2 Case 2: Trainee ICT/Computer Science Teachers

For Case 2 participants (the trainee ICT/Computer Science teachers) different pre- and post-workshop questionnaires will be used to collect data. The procedures for the in-workshop programming exercises, in-workshop researcher log and post-workshop programming exercises, however, remain the same as those described for Case 1 novice programmers.

Pre-Workshop Questionnaire

A paper based questionnaire will be completed by the trainees before the workshop in order to determine their past programming experience and attitude towards the subject. An overview of the content of this questionnaire is presented in Table 3.

Trainee's Past Programming Experience	**Trainee's Attitude to Programming**	**Misc.**
Have trainees previously programmed	Should programming be taught in schools	Gender
What languages have trainees used	Confidence teaching programming in school	
Why trainees previously programmed	Perceived difficulty teaching programming	
Enjoyment of previous experience/Identification of concepts previously used		
Was previous experience challenging		

Table 3 – Pre-Workshop Questionnaire Overview (Case 2 Participants – Trainee Teachers)

Post-Workshop Questionnaire

A second questionnaire will be completed by trainees after the workshop in order to gauge participant's thoughts on their workshop experience. In addition, trainees will also be asked about their attitudes towards programming. Table 4 provides an overview of this questionnaire.

Trainee's Workshop Programming Experience	**Trainee's Attitude to Programming**	**Misc.**
Enjoyment of session	Consider using simulator in own lessons	Gender
Difficulty of session	Confidence teaching programming in school	
Thoughts on effectiveness of simulator	Perceived difficulty teaching programming	
Most/least liked aspects of simulator (up to three)		
Thoughts on effectiveness of workshop		
Comparison of previous programming learning experience against the workshop		

Table 4 – Post-Workshop Questionnaire Overview (Case 2 Participants – Trainee Teachers)

4.3 Case 3: Novice Programmers (High School Students)

For Case 3 participants, the same pre- and post-workshop questionnaires will be used as for Case 1. The in-workshop researcher log will also be completed. Due to this workshop being shorter in time, however, the programming tests will not be implemented.

4.4 Additional Data Source: Teacher Interviews

Teacher's thoughts on the robot simulator as a means of introducing programming concepts to novices will also be collected. Themes will include the suitability and effectiveness of the robot simulator as an introductory programming teaching resource. Semi-structured interviews will be used during this process and will be recorded with the consent of the interviewee for later transcription. By the time of the interviews all three teachers will have seen the robot simulator, the workshop sessions and will have discussed with their students about the workshop experience.

5. Analysis

As outlined in Section 4, several sources of data are to be collected during the case study. This will enable the triangulation of collected data which will strengthen the findings of the case study due to it allowing for converging lines of enquiry and corroboration. Triangulation involves taking multiple measures of a studied object and is relevant for qualitative, quantitative and mixed method studies (Runeson et al 2012). Triangulation also helps to address the potential problem of construct validity (discussed in Section 6). An electronic case study database will be used to organise and document collected data. This will be made available to secondary investigators and will help to ensure the transparency of the case study process. A chain of evidence will also be established. Details of how collected data will be analysed during the case study is presented as follows:

- Table 5 provides details of the analysis strategy for data collected from Case 1 (FE Novice Programmers) participants
- Table 6 provides details of the analysis strategy for data collected from Case 2 (Trainee Teacher) participants
- Table 7 provides details of the analysis strategy for data collected from Case 3 (High School Novice Programmers) participants

In regards to the teacher interviews each interview will be transcribed before being thematically analysed for commonalities.

Data Source	Description
Pre-Workshop Questionnaire	Qualitative and quantitative analysis
In-Workshop Programming Exercises	Examination/comparison of participants programming knowledge during the workshop
In-Workshop Researcher Log	Notable events discussed. Common trends identified
Post-Workshop Questionnaire	Comparison with pre-workshop questionnaire results in addition to further analysis
Post-Workshop Programming Exercise	Analysis of participant's programming progress, and evidence of deep learning, by grading participant's code according to a pre-determined three point scale

Table 5 – Analysis Strategy for Case 1 Data (FE Novice Programmers)

Data Source	Description
Pre-Workshop Questionnaire	Qualitative and quantitative analysis. Comparison with previously collected data reported in Major et al 2011b
In-Workshop Programming Exercises	For those without substantial Java programming experience analysis of how knowledge progressed during the workshop by examining, comparing and scoring responses. Separate analysis of data collected from participants with considerable Java experience
In-Workshop Researcher Log	Notable events discussed. Common trends identified
Post-Workshop Questionnaire	Comparison with pre-workshop questionnaire results in addition to further analysis. Comparison with collected data reported in Major et al 2011b
Post-Workshop Programming Exercise	For those without substantial Java programming experience analysis of programming progress by grading code according to a pre-determined three point scale. Separate analysis of data collected from participants with considerable Java experience

Table 6 – Analysis Strategy for Case 2 Data (Trainee Teachers)

Data Source	Description
Pre-Workshop Questionnaire	Qualitative and quantitative analysis
In-Workshop Researcher Log	Notable events discussed. Common trends identified
Post-Workshop Questionnaire	Comparison with pre-workshop questionnaire results in addition to further analysis

Table 7 – Analysis Strategy for Case 3 Data (High School Novice Programmers)

5.2 Rival Explanations

It is also intended that a further analytical strategy, examining rival explanations, will be adopted and embedded in the data collection and data analysis stages. Examining rival explanations involves engaging in a systematic search for alternative themes, divergent patterns and rival explanations (Patton 2001). Reporting that a case study sought out, considered and did not find evidence to support a number of plausible rival explanations enhances the credibility of a case study and helps to counter the suggestion that the results are shaped by any predispositions or biases. Yin (Yin 2009) lists many types of potential rivals while Rosnow and Rosenthal (Rosnow and Rosenthal 1997) discuss factors that can impact upon the result of experiments involving human subjects. Several of these issues are considered as potentially relevant to the case study and will be addressed as follows:

Rival Explanation	**Description**	**How to Address**
Null Hypothesis	Observations are the result of chance circumstance only	Workshops to be replicated. Multiple sources of evidence to be used to support findings
Novelty of the Simulator	The novelty of the simulator encourages participants to say they have learnt more than they actually have (i.e. participants confuse interest in the learning mechanism with actual learning)	Is addressed by the scoring process which distinguishes between deep and surface learning on the post-workshop programming exercises
Experimenter Expectation Effect	The scoring of the programming tests is influenced by the experimenter’s expectation that the simulator is an effective learning mechanism	Will be addressed by adhering to the marking schedule documented in Section 4.1. A random sample of this data (programming exercises completed by 12 participants) will also be marked by a second member of the research team (TK) according to this schedule. Disagreements in the scoring of these tests will be resolved by consulting the other research team member (PB) and by grading the exercises collected from all participants collectively
“Good Subject Effect”	When participants mark their subjective opinions strongly in favour of the simulator in order to aid the research project and not because it helped them to learn programming	Is addressed in several ways: 1) By the scoring process used to grade participants programming progress 2) By asking participants to identify up to three things they like/dislike about the simulator – participants are likely to be more truthful when identifying positive and negatives than simply answering a question on whether the simulator helped them to learn 3) If far more positives than negatives are reported this would corroborate positive answers to the questionnaire questions related to the effectiveness of the workshop and robot simulator
Implementation Rival	The implementation process (e.g. the nature of the workshop sessions), not the robot simulator, accounts for the results	Is addressed by asking participants to rate (on a five point scale) the effectiveness of the simulator in addition to the effectiveness of the workshop in general. If substantially more participants rate the workshop as effective, and the simulator as ineffective, then the nature of the workshop itself may account for the results of the study

Table 8 – Consideration of Potential Rival Explanations

5.3 Interpretation

During the analysis stage data will be used to address the four propositions as follows:

***P1**. A robot simulator is an effective tool for supporting the learning of introductory programming*

This proposition will be supported if:

- 75% of novices are awarded either an A or B on the post-workshop programming exercise
- The average score of novices on the in-workshop programming tests is greater than 50%
- 50% of novices rate the robot simulator as an effective introductory programming learning tool on the post-workshop questionnaire
- All of the teachers interviewed believe the robot simulator is an effective tool for supporting the learning of introductory programming

***P2**. A robot simulator improves novice's perceptions of programming*

This proposition will be supported if:

- A comparison between novice's pre- and post-workshop questionnaire data shows a positive improvement in regards to participant's perceptions of programming
- All of the teachers interviewed believe the robot simulator helps to improve novice's perceptions of programming

***P3**. A robot simulator offers a more effective introduction to basic programming concepts when compared to other teaching methods*

This proposition will be supported if:

- 50% of trainees who have previously been taught programming believe their previous introductory programming learning experience to be less effective than the one using the robot simulator
- 50% of novices who have previously been taught programming believe their previous introductory programming learning experience to be less effective than the one using the robot simulator
- All of the teachers interviewed believe the robot simulator offers a more effective introduction to basic programming concepts when compared to other teaching methods

***P4**. A robot simulator improves trainee ICT/Computer Science teacher's confidence in their ability to teach introductory programming*

This proposition will be supported if:

- A comparison between trainee's pre- and post-workshop questionnaire data shows a positive improvement in trainee's confidence in their ability to teach introductory programming

The in-workshop researcher logs will be used to ensure that any significant incidents or issues which occur during the study, and could impact upon its findings, are documented according to pre-determined criteria. The logs will not be used to directly address any of the case study propositions.

6. Plan Validity and Study Limitations

In this section measures which have been taken to ensure the validity of the case study, in addition to a consideration of potential limitations of the study, are presented.

6.1 Plan Validity

In order to ensure the rigour and reliability of the case study several measures have been taken. Firstly, as documented in Appendix A, this protocol has been designed after considering Per Runeson and Martin Höst's case study design checklist (Runneson and Höst 2008).

Secondly, as suggested by Yin (Yin 2009) in order to ensure construct validity, multiple sources of evidence (pre- and post-workshop questionnaires, in and post-workshop programming tests, in-workshop researcher log and teacher interviews) and the establishment of a chain of evidence (plans to make available a database for secondary researchers, the final report to refer heavily to collected evidence and the protocol procedures to be followed and deviations documented) are to be used.

In regards to internal validity, a pre-identification of potential rival explanations (see Section 5.2) coupled with the adoption of a data collection and data analysis strategy which actively investigates these rivals helps to ensure internal validity is established.

As Case 1 and Case 3 participants are aged between 14 and 18 years old, and do not all come from the same educational institution, it is believed that the results of the study will be generalisable to a similar demographic of novice programmers. Moreover, despite all Case 2 participants being enrolled on a Teacher Training Course (PGCE) at Keele University the programming backgrounds of participants are significantly varied. As such it is considered that the results of the study will be generalisable to a similar demographic of trainee ICT/Computer Science teachers. As the case study protocol has undergone expert review, in addition to peer review, the risk of unidentified threats to the validity of the study are considered to have been minimised.

6.2 Study Limitations

Aside from the programming exercises, other instruments that will be used during the case study will collect data that is self-reported (i.e. cannot be independently verified and what participants say in interviews and questionnaires has to be taken at face value). This may lead to sources of bias such as selective memory and exaggeration. It is intended that the use of open and closed questions (to avoiding ‘leading’ participants) and reinforcing the anonymous nature of the study will help to reduce the potential impact of self-reported bias.

Another possible limitation of the case study is that the interviewees and student participants will be self-selected. Indeed, in Major (Major 2012) it is described how some potential learner participants chose not to be involved in a study after they were approached. There is a risk that a similar occurrence during the case study could result in some data being excluded from the final report. By inviting a broad selection of participants to take part in the research (which in total will number over 65), however, it is predicted that this risk has been minimised.

7. Summary

In this paper a protocol which provides details of a case study that will investigate the use of a robot simulator as an introductory programming teaching tool has been presented. Such research is being undertaken as a Systematic Literature Review indicated that this work would be valuable. The development of a case study protocol in advance of the main study will help to ensure that reliable, transparent, targeted and rigorous work is performed. Furthermore, potential problems which may affect the study have been considered and accounted for in advance of its implementation. This protocol provides background information, details of the planned study design, information about the strategies for data collection and data analysis in addition to a consideration of factors which could affect the validity of the study. This protocol may also act as a point of reference for other researchers interested in performing a case study.

References

ACM/IEEE Interim Review Task Force, 2008. Computer Science Curriculum. ACM/IEEE.

ANSI Standards Committee on Dental Informatics, 2001. Working Group Educational Software Systems Guidelines for the Design of Educational Software.

Beale, R. and Sharples, M., 2002. Design Guide for Developers of Educational Software. British Educational Communications and Technology Agency, 1. pp. 1-29.

Becker, B.W., 2001. Teaching CS1 With Karel the Robot in Java. In SIGCSE '01: Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education, pp. 50–54. ACM.

Borge, R., Fjuk, A. and Groven, A.K., 2004. Using Karel J Collaboratively to Facilitate Object-Oriented Learning. In ICALT 2004, pp. 580–584.

Brereton P., Kitchenham B., Budgen D. and Li Z., 2008. Using a Protocol Template for Case Study Planning. Proceedings of EASE 2008, BCS-eWIC.

Buck, D. and Stucki, D.J., 2001. JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum. In Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education, New York, NY, USA, pp. 16–20. ACM.

Case, J., 2008. Education theories on learning: an informal guide for the engineering education scholar. Higher Education Academy Engineering Subject Centre, Loughborough University. Accessed Online: http://hdl.handle.net/2134/9730 (2nd May, 2012).

Enderle, S., 2008. Grape: Graphical Robot Programming for Beginners. In Research and Education in Robotics EUROBOT, 33, pp. 180–192. Springer Berlin Heidelberg.

Fagin, B., 2003. Measuring the Effectiveness of Robots in Teaching Computer Science. In 34th SIGCSE Technical Symposium on Science Education, pp. 307–311. ACM.

Kelleher, C. and Pausch, R., 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. ACM Computer Survey, 37 (2), pp. 83–137.

Klein, H. K. and Myers, M. D., 1999. A set of principles for conducting and evaluating interpretive field studies in information systems. MIS Quarterly, 23(1):67.

Ladd, B. and Harcourt, E., 2005. Student Competitions and Bots in an Introductory Programming Course. J. Comput. Small Coll., 20 (5), pp. 274-284.

Lauwers, T., Nourbakhsh, I. and Hamner, E., 2009. CSbots: Design and Deployment of a Robot Designed For the CS1 Classroom. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education, New York, NY, USA, pp. 428-432. ACM.

Lee, A. S., 1989. A scientific methodology for MIS case studies. MIS Quarterly, 13(1):33.

Lemone, K.A. and Ching, W., 1996. Easing into C++: Experiences with RoBOTL. SIGCSE Bull., 28 (4), pp. 45-49.

Major, L., 2012. An evaluation of the Advanced Diploma from the Perspective of Staff and Learners. Research in Post-Compulsory Education. Volume 17, Issue 1, March 2012.

Major, L., Kyriacou, T. and Brereton, O.P., 2011a. Systematic Literature Review: Teaching Novices Programming Using Robots. In 15th International Conference on Evaluation and Assessment in Software Engineering (EASE 2011), Durham University, UK, 11 - 12 April 2011. IET. pp. 21-30.

Major, L., Kyriacou, T. and Brereton, O.P., 2011b. Experiences of Prospective High School Teachers Using a Programming Teaching Tool. In Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11), Koli National Park, Finland, 17-20 November 2011. ACM, New York, NY, USA. pp. 126-131.

Major, L., Kyriacou, T. and Brereton, O.P., 2012. Systematic Literature Review: Teaching Novices Programming Using Robots. In IET Software (To Appear).

Martin, C. and Hughes, J., 2011. Robot dance: edutainment of engaging learning. Proceedings of the 23rd Psychology of Programming Interest Group, PPIG (2011), York, UK, 6-8 September.

McGill, M. M., 2012. Learning to program with personal robots: Influences on student motivation. ACM Trans. Comput. Educ. 12, 1, Article 4 (March 2012).

McWhorter, W.I. and O'Connor, B.C., 2009. Do LEGO Mindstorms Motivate Students in CS1? In SIGCSE '09, pp. 438-442. ACM.

Patton, M., 2001. Qualitative Research and Evaluation Methods (2nd Edition). Thousand Oaks, CA: Sage Publications. p. 553.

Robson, C., 2002. Real World Research (2nd Edition). Blackwell.

Rosnow, R. and Rosenthal., R., 1997. People studying people - Artefacts and Ethics in Behavioural Research. W.H. Freeman and Co, New York.

Runeson, P. and Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Softw. Engg. 14 (2), pp. 131-164.

Runeson, P., Höst, M., Rainer, A. and Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples (1st Edition). Wiley.

Sartatzemi, M., Xinogalos, S. and Dagdilelis, V., 2003. An Environment for Teaching Object-Oriented Programming: objectKarel. In Proceedings of ICALT 2003, pp. 342-343.

Squires, D. and Preece, J., 1999. Predicting quality in educational software: Evaluating for learning, usability and the synergy between them. Interacting with Computers, 11 (5), pp. 467-483.

TIOBE Programming Community Index for April 2012. Accessed Online: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (24th August, 2012).

Yin, R.K., 2000. Rival explanations as an alternative to reforms as experiments. In L. Bickman (Ed.) Validity and social experimentation: Donald Campbell's legacy. Thousand Oaks, CA : Sage.

Yin, R. K., 2009. Case Study Research: Design and Methods (4th Edition). Sage.

Appendix A: Case Study Design Checklist

Item	Checklist Question	Comments
1	What is the case and its units of analysis?	See Section 3 'Design'
2	Are clear objectives, preliminary research questions, hypotheses defined in advance?	One main research question (see Section 3.1) and several propositions (see Section 3.2) have been outlined
3	Is the theoretical basis - relation to existing literature or other cases - defined?	Results of a previously completed SLR (Major et al 2011a, Major et al 2012) provide the basis for this study
4	Are the authors' intentions with the research made clear?	The purpose of the study is to determine whether a robot simulator is an effective tool for supporting the learning of introductory programming (see Section 3.1 'Aim')
5	Is the case adequately defined (size, domain, process, subjects…)?	See Section 3 'Design'
6	Is a cause–effect relation under study? Is it possible to distinguish the cause from other factors using the proposed design?	See Section 5.2 'Rival Explanations' and Section 6.1 'Plan Validity'
7	Does the design involve data from multiple sources (data triangulation), using multiple methods (method triangulation)?	The case study design involves collecting multiple forms of data using multiple data collection methods (as detailed in Section 4 'Data Collection'). Collected data will be triangulated as outlined in Section 5 'Analysis'
8	Is there a rationale behind the selection of subjects, roles, artefacts, viewpoints, etc.?	Yes. This is described throughout the protocol document
9	Is the specified case relevant to validly address the research questions	Expert and peer review of the protocol, use of multiple sources of evidence and the establishment of a chain of evidence help to overcome potential issues with construct validity
10	Is the integrity of individuals/organisations taken into account?	This factor is recognised in Section 3.6 'Ethical Considerations and Participant Code Numbers'

Observing Mental Models in Novice Programmers

Richard Bornat[1], Saeed Dehnadi[1] and David Barton[2]

R.Bornat@mdx.ac.uk, S.Dehnadi@mdx.ac.uk, bartond@trinityhigh.net
[1]School of Science and Technology, Middlesex University, London, UK;
[2]Redditch Trinity High School, Redditch, UK

Abstract. A test which partitions subjects into those who appear to use an algorithmic model of program execution and those who do not has been automated. Experiments have been conducted in a UK school with a year cohort of students aged 13-14 and in a Mexican university with a cohort of novice computer scientists. In the school about a third of subjects appeared to use an algorithmic model, which we find surprisingly many; in the university there were around half in the same category, which is in line with university results in the UK. Operation of the online test and the analysis tool is described. Interviews with subjects in the school revealed some ways in which the algorithmic classification may be expanded. End-of-course results are not yet available for the test subjects, so statistical associations have not yet been explored.

1 Background

Dehnadi (2006, 2009) observed that some novices confronted by simple programming exercises give rational but incorrect answers. Their answers are algorithmically plausible though not always orthodox: for example, they might assign the value of a variable from left to right rather than right to left, or move a value in an assignment rather than copy it. He devised a test made up of questions about assignment and sequence programs, delivered to novice programmers without giving an explanation of the questions; those who consistently gave algorithmically-plausible results were significantly more likely to pass the end-of-course examinations than those who did not. Dehnadi et al. (2009) summarise the results of his experiments, applied to a large number of students in a wide range of universities in the UK, showing that consistent use of an algorithmic model is not simply the result of background programming experience, and that by contrast such experience on its own has little or no effect on success in the end-of-course examination; it also gives references to and discussion of previous related work.

Robins (2010) took note of Dehnadi's results, and hypothesised that learning to program is difficult because courses present a sequence of topics, each strongly supporting the next. If a student fails to understand one topic, then the next becomes far more difficult. In statistical simulations he showed that quite weak topic-on-topic dependencies produce strongly bimodal course results, a ubiquitous effect in first courses in programming which is otherwise hard to explain. It seems likely that Dehnadi's test quantifies a cognitive obstacle which trips up many students early on in their first programming course. We hypothesise (at this stage without experimental confirmation) that many novices find it difficult at first to understand that machines act utterly formally, without considering the consequences and without intention.

Ford and Venema (2010) administered Dehnadi's test to a course cohort *after* the course examination. Only 50% of those who *passed* the course could answer the multiple-assignment questions using the correct programming-language model. This opened up a new way of using the test, as a measure of successful learning in supposed practioners.

We want to administer Dehnadi's test more widely, in schools as well as in universities, and to administer it in non-English-speaking countries. To do so we decided to construct an automated version.

2 An online test

Dehnadi's test was paper-based, which made it difficult and expensive to administer – all the test cohort had to be gathered, examination-style, in the same room at the same time – and expensive to assess – he had to read every test script and apply a fairly intricate algorithm to come to a judgement of 'consistency' or 'inconsistency'.

We had hoped to be able to integrate the test into one of the widely-used ILEs (Interactive Learning Environments) such as Blackboard or Moodle, so that teachers could easily administer it as part of the normal course activity, could receive data on performances of the individuals and groups, and could easily correlate the test results with course results. We would also avoid difficult questions about data protection by operating entirely inside the ILE firewall. But the question format of the ILEs we looked at were unhelpful, and none of them were able to implement the subtleties of the assessment algorithm (see section 2.1).

```
int a = 10; int b = 20; a = b;
int a = 10; int b = 20; b = a;
int big = 10; int small = 20; big = small;

int a = 10; int b = 20; a = b; b = a;
int a = 10; int b = 20; b = a; a = b;

int a = 10; int b = 20; int c = 30; a = b; b = c;

int a = 5; int b = 3; int c = 7; a = c; b = a; c = b;
int a = 5; int b = 3; int c = 7; c = b; b = a; a = c;
int a = 5; int b = 3; int c = 7; c = b; a = c; b = a;
int a = 5; int b = 3; int c = 7; b = a; c = b; a = c;
int a = 5; int b = 3; int c = 7; b = a; a = c; c = b;
int a = 5; int b = 3; int c = 7; a = c; c = b; b = a;
```

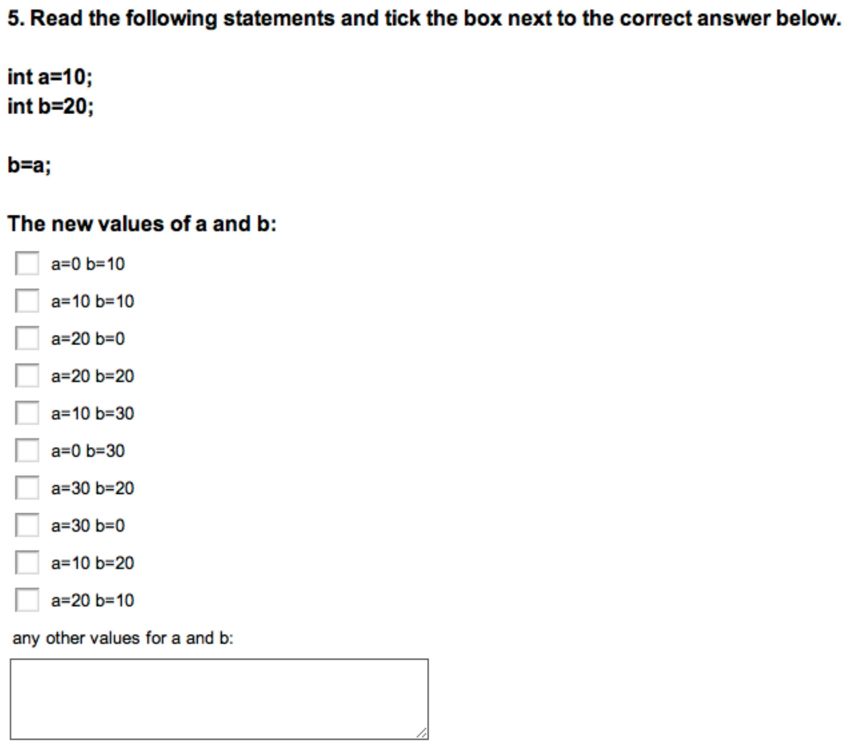
Fig. 1. Dehnadi's test questions

We had already developed a program which could generate the paper version of Dehnadi's test from a textual description such figure 1, using a formal description of each of his models to generate the answers and producing LaTeX code for the question and answer sheets. We took the output of the program and transcribed it into SurveyMonkey (SurveyMonkey, 2012). Each question was coded as a multiple choice; subjects could tick as many responses as they wished; there was a text box to enter alternative answers. A sample question is shown in figure 2. There were some difficulties in the transcription, so to improve accuracy we modified the generator program to produce also a text file that could be pasted, piece by piece into SurveyMonkey to produce an exact version of the test. Question answers are presented in a randomised order by the SurveyMonkey mechanism, in order to avoid the questionnaire bias which might arise if the same model appeared in the same answer-position in each question.

2.1 Analysing the output

SurveyMonkey can generate a CSV (comma-separated values) output of survey responses, each line of the output corresponding to one session with a particular subject. Our generator program was modified to analyse this output, applying a version of Dehnadi's original assessment algorithm (Dehnadi, 2006, 2009). Mental models are made up of an assignment model (one of M1-M11, each describing the action of a single assignment statement) and a sequence model (one of S1-S3, each describing the action of a number of assignments written one after the other). Each mental model determines an answer to each question, which can be a tick in a single box or in several boxes. In all but the first three single-assignment questions there are many responses – sets of ticks – which are ambiguous in their interpretation.

To resolve this ambiguity Dehnadi had used a marksheet (figure 3) with a column for each assignment model. An ambiguous answer marked all the columns whose assignment model generated that answer. Marks were notionally in pencil. Then marks in the column with the most ticks were notionally inked, and 'consistency' in answering the test was judged as follows:



5. Read the following statements and tick the box next to the correct answer below.

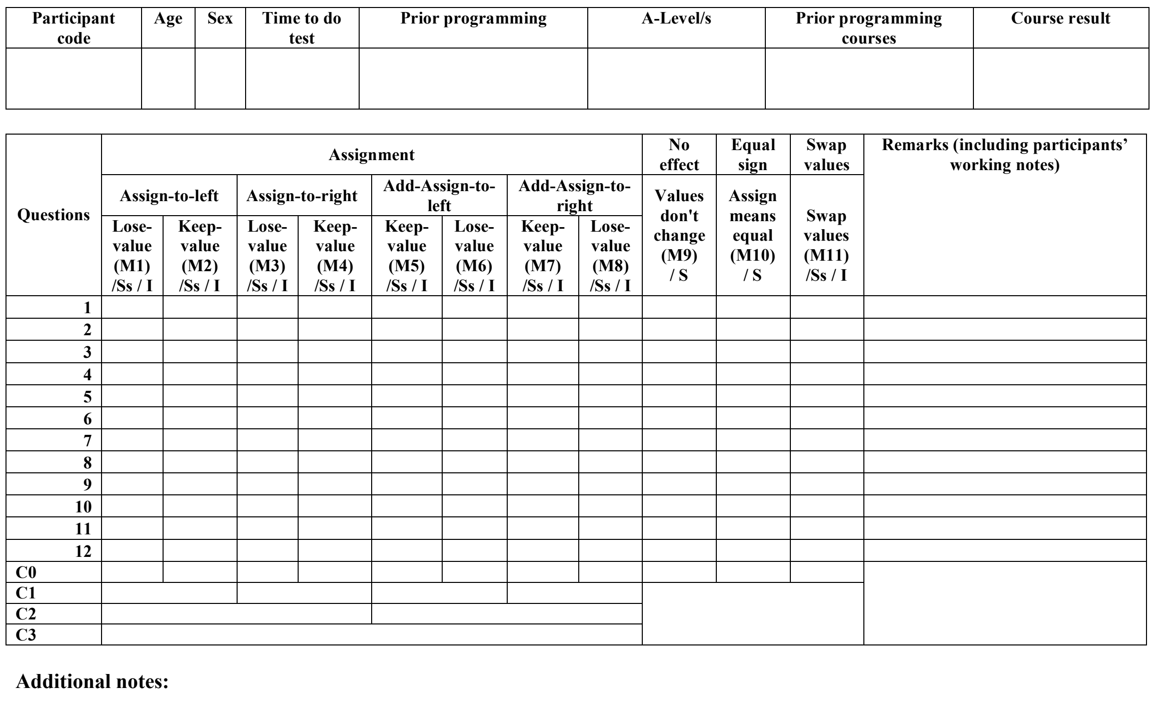
```
int a=10;
int b=20;

b=a;
```

The new values of a and b:

- [ ] a=0 b=10
- [ ] a=10 b=10
- [ ] a=20 b=0
- [ ] a=20 b=20
- [ ] a=10 b=30
- [ ] a=0 b=30
- [ ] a=30 b=20
- [ ] a=30 b=0
- [ ] a=10 b=20
- [ ] a=20 b=10

any other values for a and b:

Fig. 2. One of Dehnadi's test questions in SurveyMonkey



Participant code	Age	Sex	Time to do test	Prior programming	A-Level/s	Prior programming courses	Course result

Questions	Assignment								No effect	Equal sign	Swap values	Remarks (including participants' working notes)
	Assign-to-left		Assign-to-right		Add-Assign-to-left		Add-Assign-to-right		Values don't change (M9) / S	Assign means equal (M10) / S	Swap values (M11) /Ss / I	
	Lose-value (M1) /Ss / I	Keep-value (M2) /Ss / I	Lose-value (M3) /Ss / I	Keep-value (M4) /Ss / I	Keep-value (M5) /Ss / I	Lose-value (M6) /Ss / I	Keep-value (M7) /Ss / I	Lose-value (M8) /Ss / I				
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
C0												
C1												
C2												
C3												

Additional notes:

Fig. 3. Dehnadi's marksheet

1. A response with six inked marks in the same column for Q1-Q6 (single and double assignment) was judged consistent.
2. Otherwise, a response with 8 or more inked marks in the same column was judged consistent.
3. Otherwise, a response with fewer than 8 marked rows (two-thirds of questions) was judged blank.
4. Otherwise, the response was judged inconsistent.

Note that the assessment ignored the subject's use of sequence models, because of the difficulty of analysing the test results on paper. Note also that 'consistency' is an abbreviation for 'consistent use of a recognised rational mental model': Dehnadi was not judging a psychometric attribute of the subject, but rather a particular characteristic of their test performance.

The analysis tool uses a similar algorithm, but takes account of the use of sequence models to refine its judgements. Each subject's answer to each question is a set of ticks (write-in answers are converted to ticks of imaginary boxes with that answer). The first three questions don't require a sequence model, so answers to those questions are interpreted ambiguously as using any one of the sequence models. Rather than using 'consistent' and 'inconsistent', which can be misinterpreted, it makes judgements 'Algorithmic' and 'Unrecognised'.

Each question-response – a set of ticks – corresponds to a set of mental models, use of any of which will generate that response. The tool looks for the mental model which appears most often in the responses over the whole test (by analogy with Dehnadi's 'inking' step). If a single model is used in each of the first six questions, the response is judged 'Algorithmic (first 6)'; otherwise a single model used eight times gives 'Algorithmic overall'. This is a harsher assessment scheme than Dehnadi's because, for example, a subject who uses M2+S1 in four questions and M2+S2 in four others would not be judged Algorithmic, though Dehnadi would have labelled them 'consistent': so there's a lesser judgement 'Possibly algorithmic', assessed by considering only the assignment model and ignoring the sequence models used. There is also a judgement 'No change' applied to those who simply ticked the values from the original state (Dehnadi would have judged them 'consistent' using model M9).

3 A first experiment

We applied the test to 126 school students in an 'academy'[1] at the end of year 9 (ages 13-14). These students had been exposed to programming with MIT Scratch (MIT, 2007) and to ICT tools such as Microsoft Office (Microsoft, 2012). The analysis of their responses is shown in table 1.

Table 1. School experiment

Algorithmic		Possibly algorithmic		Unrecognised	No change	Blank
overall	first 6	overall	first 6			
42	2	4	0	70	5	3

Although these students were not complete novices, they had not been exposed to any notion of assignment, hardly at all to sequence, and not at all to formal programming notation. In fact most judged Algorithmic used the sequence model S3, in which assignments are executed in parallel. They were a complete year cohort, not a group self-selected for their interest in programming. In undergraduate novice computer scientists we have typically seen 50% or more judged algorithmic; we were surprised to find that as many as a third of these school students appeared to use an algorithmic model in 8 out of 12 questions, and that almost none had answered fewer than 8 questions.

[1] A state-funded school with a comprehensive – non-selected – intake and a state-approved aspirational agenda for its pupils.

Because the test was marked by a program, we could analyse the responses in minutes and review the judgements immediately. The school allowed us to interview some subjects on the afternoon of the day they took the test, some selected by them and some selected by us. The school's selection included both Algorithmic and Unrecognised subjects, in each group some who their teachers had expected to be placed there and some surprises. We picked out some more Algorithmic and Unrecognised individuals, and in the time we had left we tried to select at random from the students in the classroom. Overall we interviewed about fifteen subjects, out of 40 who had taken the test that morning.

We found that every interviewed student judged Algorithmic, prompted only by the question "what did you think was happening?", reported use of the mechanism identified by the analysis tool. Three of those judged Unrecognised reported something new to us: two seemed to have switched models mid-test, and one seemed to be using two models at the same time, reporting both answers. Some other Unrecognised students answered that question with a shrug, and we hadn't enough time to probe their thinking.

The analysis tool was refined to try to pick up the sequential and concurrent model users. Observation of the data showed that some subjects had ticked all or almost all the answer boxes in each question, and a judgement 'Ticked everything' was applied to them: we feel that those responses are a kind of protest, more Blank than Unrecognised. It was also possible to see that some students appeared to be algorithmic in 6 out of the last 9 questions (double and triple assignments). The refined analysis is shown in table 2. The drop in 'Algorithmic overall' is due to a decision to demote consistent use of the Equality model to 'Possibly algorithmic'; other changes are due to introduction of new judgements.

Table 2. School experiment (refined analysis)

Algorithmic					Possibly algorithmic			Unrecognised	No	Ticked	Blank
overall	first 6	last 9	sequential	concurrent	overall	first 6	last 9		change	everything	
39	2	7	7	1	5	0	2	54	5	3	3

Dehnadi's marksheet assessment would have reported 48 'consistent'; the tool reports 46 in the corresponding columns 'Algorithmic overall', 'Algorithmic (first 6)', 'Possibly algorithmic (overall)' and 'Possibly algorithmic (first 6)'. It has spotted 6 rejections where the marksheet algorithm would have had 3, though a human marker would surely have noted the 'Ticked everything' responses at least informally as rejections.[2] It has demoted 3 'consistent' responses to 'No change' and recognised two 'inconsistent' responses as the same thing.

Some of the new judgements may or may not correlate with success in the course examination. The 'Algorithmic (sequential)' category is small but may prove interesting; the 'Algorithmic (concurrent)' category may be an artefact of looking too hard. So may the 'first 6' and 'last 9' judgements. 'last 9' is especially problematic in this experiment: inspection of the tool's output showed that one way to produce it was to tick the same answer in each of the last six questions, an aspect of questionnaire bias that we hadn't previously noted. The generation/analysis tool has already been modified to check tests for that particular bias so that in future experiments we don't provoke such opportunistic behaviour.

4 A second experiment

LimeSurvey (Limesurvey, 2012) is an open-source tool which provides a very similar mechanism to SurveyMonkey but also allows direct input of survey descriptions. Although the copy-and-

[2] Dehnadi didn't see of that kind of response in his experiments. It may be that school students are more rebellious than undergraduates and college students, or it may be a consequence of online administration of the test. At the time of administration the analysis tool didn't generate this judgement, so we weren't able to interview any of the corresponding subjects.

Table 3. Mexican experiment (refined analysis)

Algorithmic					Possibly algorithmic			Unrecognised	No change	Ticked everything	Blank
overall	first 6	last 9	sequential	concurrent	overall	first 6	last 9				
44	2	7	3	1	3	0	3	20	1	0	8

paste mechanism of SurveyMonkey allows accurate transcription of test questions and answers, it is somewhat tedious, and SurveyMonkey requires payment in order to support a test with as many questions as Dehnadi's together with administrative questions such as 'what is your name', auxiliary questions such as 'have you programmed before', and so on. We modified the generator tool to generate an XML file describing the test to LimeSurvey (still using the original test as in figure 1, because we hadn't at that time recognised the 'last 9' questionnaire bias). Edgar Cambranes-Martinez of the University of Sussex translated its English texts into Spanish. This Spanish version was tried out on 92 students from a university novice computer science cohort in Mexico.

Analysis of the results is in table 3. This result is like Dehnadi's results (Dehnadi, 2009) in UK universities: 49, or just over half, would have been judged 'consistent' by him (Algorithmic or Possibly algorithmic, overall or first 6). The 'Algorithmic (sequential)' category is present, as in the first experiment, but proportionally a little smaller in this case (3/93 rather than 7/126). Most of the other categories are populated too, but we note that there were no 'Ticked everything' protests.

5 The benefits of interviewing

The statistical association of 'consistent'/'inconsistent' judgements in Dehnadi's test with success/failure in the course examination is highly significant. But viewed as a predictor of success or failure in the course examination the test doesn't do so well (Bornat et al., 2008). Less than 20% of his 'consistent' subjects fail, which makes 'consistency' look like quite a good predictor of success, but over 40% of 'inconsistent' subjects pass, which makes 'inconsistency' not such a good predictor of failure (see (Dehnadi et al., 2009), table 9). Those 'false negatives' are clearly very interesting.

It was always clear that Dehnadi's experiments were incomplete without interviews with both kinds of subjects. We have so far carried out only a very few very unstructured interviews under difficult conditions, but we immediately saw that there were some previously unrecognised ways of answering the test with an algorithmic mindset. So far those categories which we have recognised and can pick out from the data with an analysis tool cover few subjects, but we expect that more careful interviewing of more subjects judged Unrecognised will refine our analysis still further.

In the school experiment, almost all of the subjects placed in one of the algorithmic categories were those expected to do well by their teachers, but our tool also picked out some others. From the 40 that were tested during our visit the tool spotted four of these surprises: each one on interview reported use of the model the tool had identified. The teachers were delighted because they had an opportunity to congratulate students who didn't otherwise get much encouragement, and the students were gratified.

In at least one case an interview revealed a novel description of an algorithmic model. It was unfortunately impossible to explore that description further at the time. Future experiments will certainly try to explore subjects' descriptions in depth.

One surprise in our interviews was that at least one student, although judged Unrecognised, is keen to learn programming, but only if it has nothing to do with ICT. We were able to assure him that it doesn't, but we don't yet know if he will be successful.

6 Conclusions and further work

Now that we have an online test and an automatic analysis tool, it is much easier to conduct an experiment and to analyse its result. The analysis tool has already shown us some ways in which the test can be improved (removal of one kind of questionnaire bias) and can recognise some new groups who seem to be responding algorithmically. It remains to be seen how closely its judgements align with course success.

The most striking result from the two experiments we have conducted with the new tools is that interviewing subjects soon after test administration is very illuminating. We expected to find something, but not so much, so soon and so easily. We intend to do very much more interviewing in future experiments.

Interviews can do much more than tell us if our tools are getting the 'right' result. We want to know why subjects who aren't recognised as algorithmic by the tools answer as they do, because we are interested in understanding the obstacle to learning which the test has begun to quantify. We haven't so far been able to conduct interviews which probe in that way: certainly, short interviews with apparently un-algorithmic school students were unproductive.

7 Acknowledgements

We are grateful to the staff of Trinity High School, Redditch, for allowing us to test their students, and to the students for their good-humoured cooperation. We are grateful to Edgar Cambranes-Martinez for assistance in translating the test into Spanish, and for administering the test in Mexico.

References

Richard Bornat, Saeed Dehnadi, and Simon. Mental models, consistency and programming aptitude. In Simon and Margaret Hamilton, editors, *Tenth Australasian Computing Education Conference (ACE 2008)*, volume 78 of *CRPIT*, pages 53–62, Wollongong, NSW, Australia, 2008. ACS. URL http://crpit.com/confpapers/CRPITV78Bornat.pdf.

Saeed Dehnadi. Testing Programming Aptitude. In P. Romero, J. Good, E. Acosta Chaparro, and S. Bryant, editors, *Proceedings of the PPIG 18th Annual Workshop*, pages 22–37, 2006. URL http://www.ppig.org/papers/18th-dehnadi.pdf.

Saeed Dehnadi. *A Cognitive Study of Learning to Program in Introductory Programming Courses.* PhD thesis, Middlesex University, 2009.

Saeed Dehnadi, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. In *PPIG '09: Proceedings of the 21st Annual Workshop of the Psychology of Programming Interest Group*, 2009. URL http://www.ppig.org/papers/21st-dehnadi.pdf.

Marilyn Ford and Sven Venema. Assessing the Success of an Introductory Programming Course. *Journal of Information Technology Education*, 9:133–145, 2010.

Limesurvey. Website, 2012. URL http://www.limesurvey.org.

Microsoft. Office website, 2012. URL http://office.microsoft.com.

MIT. Scratch, 2007. URL http://scratch.mit.edu/.

Anthony Robins. Learning edge momentum: A new account of outcomes. *Computer Science Education*, 20(1): 37–71, 2010. doi: 10.1080/08993401003612167.

SurveyMonkey. Website, 2012. URL http://www.surveymonkey.com.

Investigating the role of programmers' peripheral vision: a gaze-contingent tool and an experiment proposal

Roman Bednarik
University of Eastern Finland
roman.bednarik@uef.fi

Paul A. Orlov
St.Petersburg State Polytechnical University
paul.a.orlov@gmail.com

Keywords: eye tracking, programming, perceptual span, experiment proposal

Abstract

Previous research of visual attention in programming has shown differences in how expert and novice programmers attend to the available information. What has not been yet sufficiently investigated is the degree with which the information is sampled by the peripheral vision during programming. Such issues have been investigated by a contingent-window paradigm in other domains and we have thus developed a tool allowing such studies in programming. In this paper, we introduce the tool and a proposal for an experiment we plan to conduct.

1. Visual attention in programming

Computer programmers typically develop software using highly dynamic, integrated development environments (IDE). These tools often present information in adjacent windows that provide multiple representations about the software. One line of psychology of programming (PoP) research employs so called contingent window paradigm to investigate how programmers coordinate the representations during comprehension and debugging (Romero et al., 2002).

Other studies have employed eye-tracking as a proxy to visual attention during programming. Since Crosby and Stelovsky's 1990 seminal paper (Crosby and Stelovsky, 1990), a body of knowledge about attention in programming has been growing, indicating that expertise in programming is characterized by distinct patterns of visual attention (Bednarik, 2012). For example, expert programmers employ a wider range of strategies during debugging with multiple representations (Bednarik, 2012) that develop over time (Bednarik and Tukiainen, 2008), they attend to beacons (meaningful areas of code) (Crosby and Stelovsky, 1990), and attend to output more often than novices during debugging (Hejmady and Narayanan, 2012).

Beyond single user studies, recent developments in dual eye tracking methodology allowed expansion of eye tracking programming studies to multiple user situations (Jerman et al., 2012). Jermann and Nüssli (2012) conducted a study in which pairs of programmers collaboratively comprehended source code while their visual attention was captured using an eye-tracker. They found that programmers' attention is often coupled and this coupling increases during code selection episodes accompanied by speech. Bednarik et al. (2011) transferred the point of gaze of an expert programmer to the display of novice programmers while explaining an algorithm. They showed that the attention of the novice programmers was more similar when the gaze of the expert was shown, but that had no effect on the comprehension outcome.

An area deserving a deeper investigation in PoP is the role of visual attention beyond the fixation point. In other domains, such as in reading, the perceptual span - the breadth of attention surrounding the point of gaze - plays a significant role during stimuli processing. The size of perceptual span directly influences the spread of covert attention during scanning of stimuli, and thus mediates performance. In search, for instance, the span varies depending on the difficulty of the distractor.

In programming, we hypothesize, wider perceptual span provides (expert) programmers faster and more effective navigation in source code and better information extraction in coordination of various representations. Further, we conjecture, compound and more difficult expressions have effect on perceptual span and the difficulty again interacts with expertise.

2. Perceptual span and Gaze contingent moving window paradigm

For most of the awaken time human's vision fixates objects. Fixations, the relatively stable movements of the eyes when perception is turned on, typically last about 300ms. Saccades, on the other hand, are rapid ballistic movements lasting no more than tens of milliseconds during which perception is virtually shut down. While fixations are required for perception to happen, there is a question about the size of the perceptual area around the center of fixation within which detailed information can still be sampled without an additional eye movement. Dodge suggested calling this area a fixation zone (Dodge, 1907).

In his research of the perceptual span in reading, Rayner discovered that the size of perceptual span is not constant but varies as a function of text difficulty. The size of the span decreases when text is difficult to read (Inhoff et al., 1989; Rayner, 1986). Rayner also suggested that the visual field can be divided into three regions: foveal, parafoveal and peripheral. Although acuity is very good in the fovea (the central 2° of vision), it is not nearly as good in the parafovea (which extends out to 5° on either side of fixation), and it is even poorer in the periphery (the region beyond the parafovea) (Rayner, 1998). Studies by Gippenreiter (1964) shown that participants recognize objects positioned 10-15° from the center of fixation. The perception of the peripheral signal occurred simultaneously with the preparation of the saccade, when the eye was still on the fixation point (Gippenreiter, 1964).

Several methods have been developed in past to evaluate the breadth of attention, many of which fall under the category of moving window paradigm. One of the most prominent techniques to investigate the role of perceptual span under this paradigm is *gaze-contingent multiresolutional display* (GCMRD) (Reingold, Loschky et al., 2003). Such systems combine eye-tracking and a stimulus so that the center of fixation is determined by an eye-tracker and the current fixation areas on the stimulus are rendered in sharp focus while the rest of the stimuli is blurred. The properties of the focused area can be contingent on the characteristics of the fixation, for example, longer fixations cause the focused area to be enlarged.

There are numerous applications for GCMRD systems, such as in resource-demanding graphics environments to allow for efficient 3D rendering and modeling, and in video and image compression methods. The following lists some of the main issues that arise when constructing GCRMD systems (Reingold, Loschky, et al., 2003):

≅ Can we construct just undetectable GCMRDs that maximize savings in processing and bandwidth while eliminating perception and performance costs?

≅ What are the perception and performance costs associated with removing above-threshold peripheral resolution in detectably degraded GCMRDs?

≅ What is the optimal resolution drop-off function that should be used in guiding the construction of GCMRDs?

≅ What are the perception and performance costs and benefits associated with employing continuous vs. discrete resolution drop-off functions in still vs. full-motion displays?

The range of applications of GCMRD extends beyond restricting views and bandwidth savings. For instance, intelligent interfaces can make use of gaze-contingent zooming, where the available information about the object increases with the proximity of attention to the object. One example of measuring attention in similar systems builds on face-detection algorithms. The closer a person is to the monitor, the higher level of attention and the higher level of detail are employed. The amount of information depends on the distance of the human face of the monitor (Harrison, Dey, 2008).

3. Gaze-contingent moving window in PoP

In programming research, the gaze-contingent window paradigm has previously been employed in a series of studies of Romero et al.. As an alternative to gaze-contingency, the authors employed the Restricted Focus Viewer (RFV) (Jansen et al., 2003). RFV emulates gaze-contingency by mouse-contingency in such a way that a user controls the position of the only focused area on the screen using the computer mouse. The movements of the mouse are recorded and are supposed to be the estimates of the point of gaze in time. In a replication study Bednarik and Tukiainen (2007) however demonstrated that in programming such assumption does not hold, and in addition, the restriction interferes with natural strategies.

One explanation of the findings is that programmers do need to use the peripheral information for some purposes, and when the periphery is masked, the performance decreases because peripheral information is not available directly. Thus, a question arises concerning the role of periphery and size of perceptual span in programming. Bednarik and Tukiainen reported that when using RFV, programmers glanced towards the blurred areas and that the contingency had slightly different effects depending on expertise.

Their study does not provide an answer why the programmers perform in this way. A hypothesis to investigate is whether expert programmers are better able to extract parafoveal and peripheral information without moving the eyes and when the cost of accessing the information increases because of blurring, they need to perform extra eye-movements.

There are numerous questions spanning from the notion of role of the perceptual span that need to be investigated to provide a qualified answer. What is the size of perceptual span of a programmer? Is there a relationship between expertise and size of perceptual span? How do changes to the size of the visible area influence behavioral patterns of programmers? Would limiting the perceptual span for novice programmers have detrimental or positive effects on performance?

These questions determine the objectives of the research planned here. Answers to the above questions will not only help in understanding the role of attention in programming, but could also will be useful when asking what effects expertise in programing has on visual attention skills.

In the rest of this paper, we present a study proposal to investigate the role of perceptual span in programming and a tool allowing GCMRD in programming.

4. Objectives and Methods of the planned research

The primary goal of the study is to determine the role of visual attention span in programming. We aim to conduct studies and test hypotheses about the influence of the perceptual span area on the behavioural patterns of programmers. To achieve this goal it is necessary to set up a number of experiments.

We put forward the following working hypothesis:

≅ The pattern of gaze of programmers when working with a restricted window IDE would be different to the pattern of gaze without blurring, dependent on the expertise of the programmer and task at hand.

We aim to answer the hypothesis by running a series of studies in which participants of distinct expertise would be engaged in various tasks such as comprehension and debugging. We will systematically modify the size of the contingent window. The dependent measures of the studies would include debugging and comprehension performance, task difficulty index, and eye-tracking measures.

The following section describes the design of the gaze contingency tool developed for the planned studies.

5. Gaze contingent IDE

To test the hypotheses it is necessary to first create an appropriate research framework. To our knowledge, existing solutions such as RFV do not allow unrestricted stimuli presentation and thus we decided to develop an extension with such functionality by ourselves.

We began developing the platform adhering to the following requirements:

≅ The system has to be able to serve as a standard software development tool in Java programming language.

≅ The source code in the editor panel will be blurred except for an area whose center is defined by a direction of programmer gaze.

≅ The transition from focused to blurred areas should be gradually smoothed.

≅ All movements of the focused area should be logged along with all source code elements that were at the point of fixation.

To satisfy the first requirement, we selected NetBeans as the experimental environment. Implementation of the source editor window blurring was carried out with the help UI library SWING. We have overridden the render method so the whole image on the screen was blurred. It was decided that the method of implementation of the blur will be based on a combination of pre-prepared maps of the gradient and the current image. This allows modifying the gradient map in a graphics editor or using programming. Changing the map scale is possible too, as well as affine mapping that allows using any 2D image as a gradient map. A screenshot of the application is shown in Figure 1.

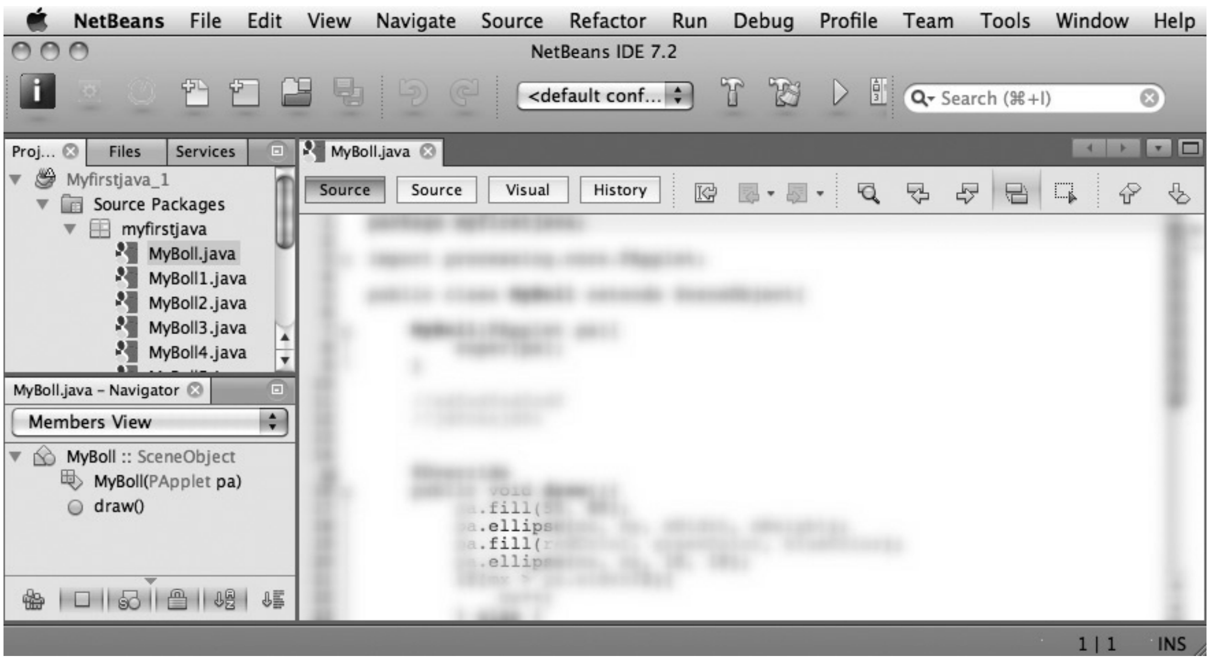


Figure 1. Screenshot of the experimental application.

The figure shows that the source editor window is blurred (and washed and line numbers and vertical scrolling, and color tips for scrolling). The sharp zone is shown only at the fixation point.

Developing of the logging mechanism for source code elements that have hit the point of fixation was carried out also by using the SWING library. The position on the screen indicates the corresponding substring in the source code. After that, when we find the substring, we use a Java parser to detect the type of substring. Such information can further be used for modifying the contingency, for example, only certain elements or types of stings could be rendered in full, responding immediately to the user's input. As a result, the output log files are similar the following example (Table 1):

Dot	Expression	Kind	Time	File
385	noFill	METHOD	1348066278437	MyBall.java
385	noFill	METHOD	1348066278437	MyBall.java

…	…	…	…	…
238	points	FIELD	1348066278721	MyBall.java
433	strokeWeight	METHOD	1348066279859	MyBall.java
621	Math	CLASS	1348066280619	MyBall.java
…	…	…	…	…
627	sin	METHOD	1348066281232	MyBall.java
633	Math.sin((float)	NONE	1348066281421	MyBall.java

Table 1. – log file example.

In this fragment, the "Dot" field corresponds to the caret location in the text file of source code, "Expression" is the substring in user's focus. The "Kind" column contains the type of substring which is obtained from JavaSource parser, "Time" is a precise timestamp of the fixation and "File" is the name of the opened source file. We will employ the logs for analysis of the visual attention patterns during navigation in the source code. A demonstration video can be found online at http://vimeo.com/49874161.

5. Future gaze-contingent IDEs

The phenomena of perceptual span and peripheral vision are not only of theoretical interests. They may be incorporated into the solutions of important practical HCI problems such as design of graphical user interfaces in general and software development interfaces in particular. Designers of interactive user interfaces wish to display information to the users so that it is attended, processed and used in the right time. Information about what information is processed and when thus contributes to better designs of interactive systems.

We wish to expand the debate about visual attention in programming by discussing the role of perceptual span and peripheral vision. We presented a gaze-contingent extension to a commonly used programming tool and an outline of studies to investigate its effects in programming tasks. Further steps include refining the experimental design and executing the study.

5. References

Bednarik, R.(2012) Expertise-dependent Visual Attention Strategies Develop Over Time During Debugging with Multiple Code Representations. International Journal of Human - Computer Studies 70, pp. 143-155.

Bednarik, R., Tukiainen, M.(2008) Temporal eye-tracking data: Evolution of debugging strategies with multiple representations. In Proc. ETRA Symposium, 99-102

Crosby, M. E., Stelovsky, J. (1990) How do we read algorithms? A case study. IEEE Computer 23, 1, 24–35.

Dodge, R.(1907) On experimental study of visual fixation.-"Psychol.Monogr.", v.35, p.95.

Gippenreiter J.B. (1978) Movements of the human eye. Monography, Moscow, pub. Moscow University, 256p.

Harrison Chris, Anind K. Dey.(2008) Zoom: Proximity-Aware User Interface and Content Magnification. CHI 2008 Proceedings I am here. Where are you? April 5-10, Florence, Italy.

Hejmady, P., Narayanan, H. N.(2012) Visual attention patterns during program debugging with an IDE. In Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA '12), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 197-200.

Inhoff, A. W., Pollatsek, A., Posner, M. I., & Rayner, K. (1989). Covert attention and eye movements during reading. Quarterly Journal of Experimental Psychology, 41A, 63-89.

Jansen, A. R., Blackwell, A. F., & Marriott, K. (2003). A tool for tracking visual attention: The Restricted Focus Viewer. Behavior Research Methods, Instruments, & Computers, 35, 57-69.

Jermann, P., Nüssli, M. (2012) Effects of sharing text selections on gaze cross-recurrence and interaction quality in a pair programming task. In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work (CSCW '12). ACM, New York, NY, USA, 1125-1134

Jermann, P., Gergle, D., Bednarik, R., Brennan, S.(2012) Duet 2012: dual eye tracking in CSCW. In Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion (CSCW '12). ACM, New York, NY, USA, 23-24

Rayner K. (1998) Eye Movements in Reading and Information Processing: 20 Years of Research. Psychological Bulletin 1998, Vol. 124, No. 3, 372-422

Reingold E. M., Lester C. Loschky, George W. McConkie, David M. Stampe. (2003) Gaze-Contingent Multiresolutional Displays: An Integrative Review. HUMAN FACTORS, Vol. 45, No. 2 pp. 307-328.

Romero, P., Lutz, R., Cox, R., Du Boulay, B. (2002) Co-ordination of multiple external representations during Java program debugging. In HCC ’02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC’02), IEEE Computer Society, Washington, DC, USA, 207.

Learning Programming by using Memory Transfer Language (MTL) without the Intervention of an Instructor

Leonard J. Mselle

Dpt. Computer Science
School of Informatics and Virtual Education
University of Dodoma
Mselel@yahoo.com

Keywords: Program visualization, Memory Transfer Language, program parsing.

Abstract

Visualization as a technique used to teach programming is gaining momentum. Memory Transfer Language (MTL) is a programmer-driven visualizer used to learn programming. This article reports on results obtained from a class experiment where MTL was used by non-novices to learn programming. The experiment was carried out to test the effectiveness of MTL in assisting students to learn programming (in a second language) without the intervention of a teacher. Results between the experimental and control group revealed that the group that studied programming using MTL without teachers' intervention performed better than the group that studied programming using conventional approach.

1. Introduction

Since at least the 1980s, computer science education researchers have searched for ways computer science teachers can better support their students (Du Boulay et al. 1981), (Du Boulay 1986). Recently, program visualization in the form of algorithm animations has yielded positive results (Hundhausen and Brown 2007), (Naps et al. 2003). To understand programming, the learner must be able to analyze/parse the program tokens unambiguously (Ben-Ari and Sajaniemi 2003). Program animation is a means to facilitate a learner to visibly parse the code. However, most of the current animators are entirely machine-driven. Helpful as it is, machine-driven animation subordinates the learner to the machine. The side effect of machine-driven animation is the denial of full authority to the learner.

So far, MTL is the only sketch-based language in programming books that can be used as a learner-driven visualizer and parser. MTL is a sketch-based language used by authors, instructors and learners to parse the code as it affects computer memory (Mselle 2011b). MTL is used to describe the program behavior by means of concrete models and visualization (2011c). In other words MTL is a tool for program parsing without actively depending on the machine.

1.2 A brief discussion and illustration of MTL

MTL is a learner-driven compiler or parser whose general framework is as illustrated in Figure 1.

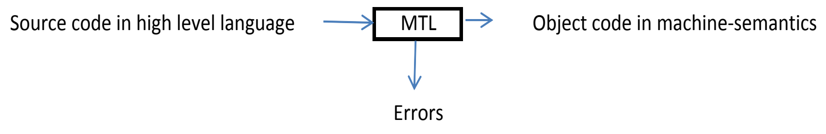
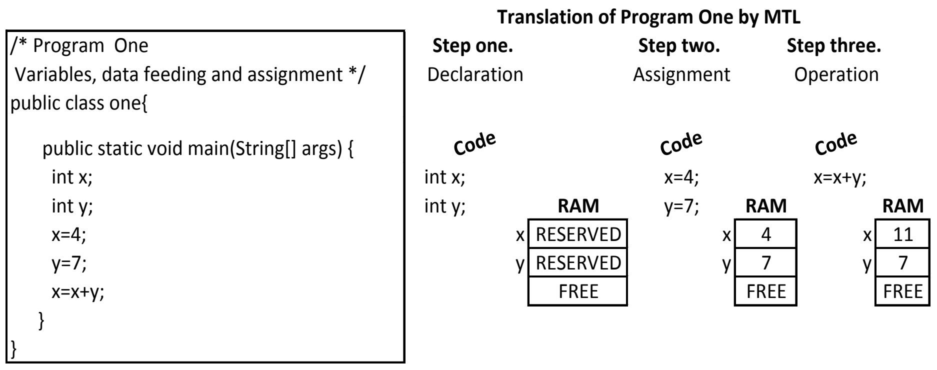

Source code in high level language
MTL
Object code in machine-semantics
Errors

Figure 1- The general framework of MTL

MTL is not machine dependent. It does not demand any knowledge or special rules for the learner to employ MTL. Successful use of MTL is directly tied with understanding the program and nothing else. Examples on the use of MTL to translate different programs and to visualize different programming aspects are demonstrated in Figures 2 through 8.

Figure 2 demonstrates how MTL is used by the learner to parse variable declaration, data feeding and data operation.

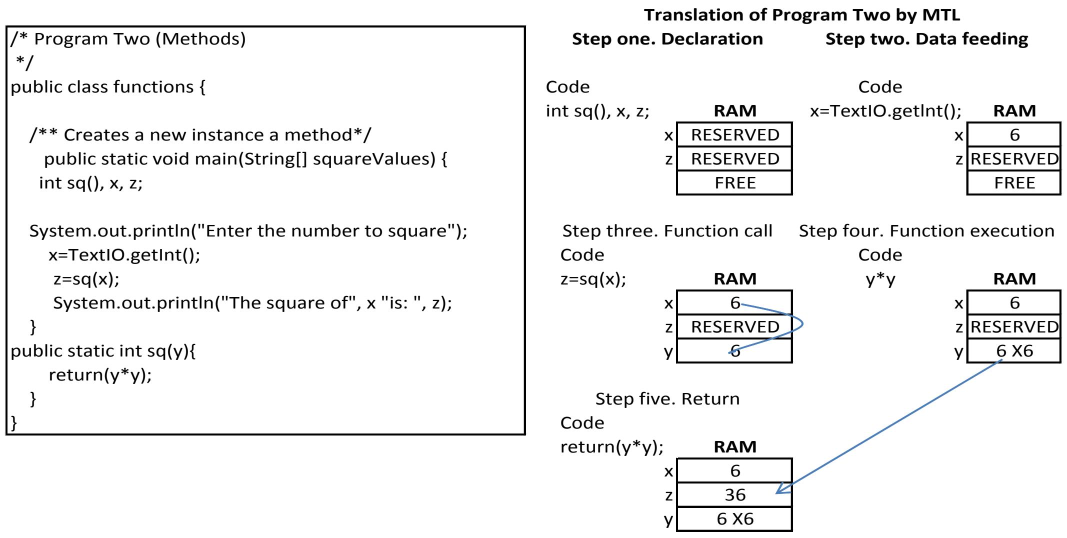


```
/* Program  One
 Variables, data feeding and assignment */
public class one{

     public static void main(String[] args) {
       int x;
       int y;
       x=4;
       y=7;
       x=x+y;
     }
}
```

Figure 2 - Parsing variable declaration, assignment and data operation by MTL

Using MTL, the relationship between variable and computer memory (RAM) is clearly visualized. Concepts of variable declaration, identifiers, data feeding, assignment, data operation such as addition, and data outputting are visualized by simple model which mimics the RAM.

Figure 3 shows how MTL can be used to parse functions/methods.

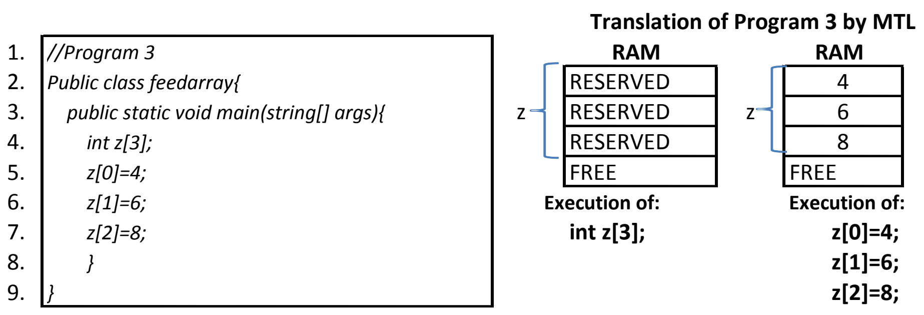


```
/* Program Two (Methods)
 */
public class functions {

   /** Creates a new instance a method*/
     public static void main(String[] squareValues) {
    int sq(), x, z;

   System.out.println("Enter the number to square");
      x=TextIO.getInt();
       z=sq(x);
       System.out.println("The square of", x "is: ", z);
   }
public static int sq(y){
      return(y*y);
   }
}
```

Figure 3- Parsing functions by MTL

Using MTL, abstract concepts such a functions, parameters and *return()* funtion are physically visualized by the same single model.

Figure 4 shows how MTL is used to translate arrays, array declaration and data feeding in arrays.

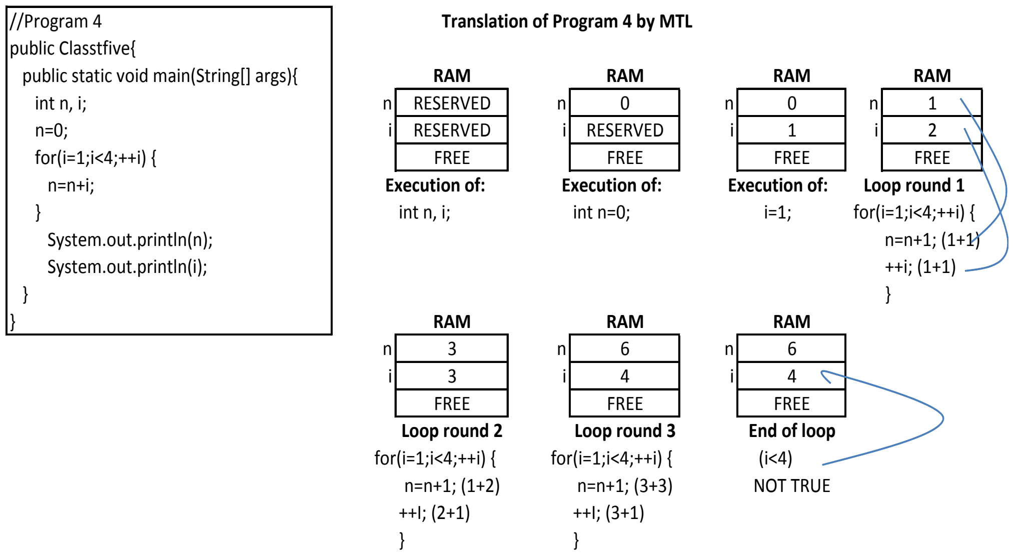


```
//Program 3
Public class feedarray{
    public static void main(string[] args){
        int z[3];
        z[0]=4;
        z[1]=6;
        z[2]=8;
        }
}
```

Figure 4- Parsing array declaration and array feeding by MTL

As demonstrated in Figure 4, the concept of arrays as a data structure different from other simple data types such as integers is clearly visualized. Data feeding in an array using subscripts is clearly visualized.

Figure 5 shows how MTL is used to translate a *for* loop.

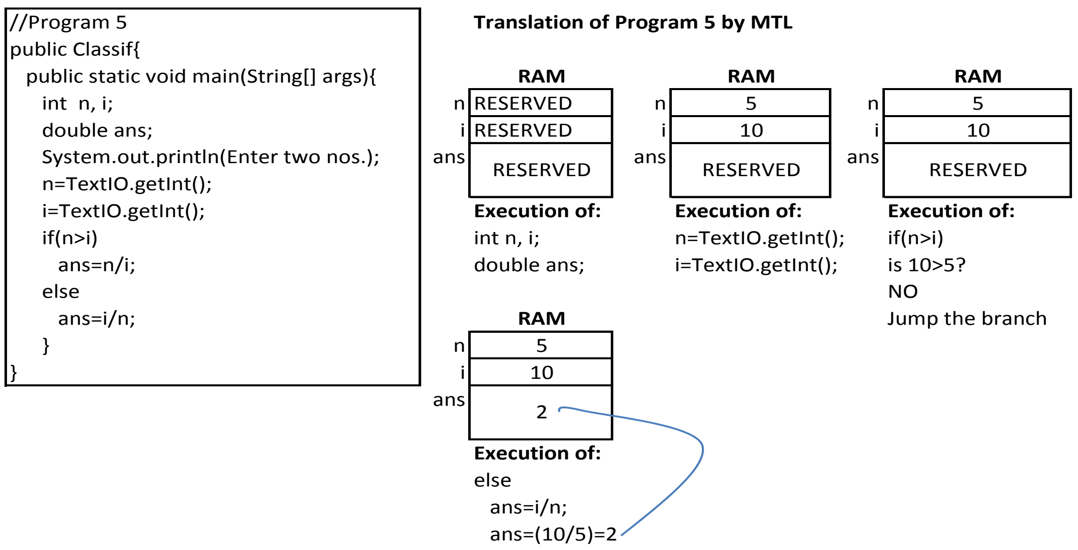


```
//Program 4
public Classtfive{
  public static void main(String[] args){
    int n, i;
    n=0;
    for(i=1;i<4;++i) {
      n=n+i;
    }
      System.out.println(n);
      System.out.println(i);
  }
}
```

Figure 5- Parsing a for loop by MTL

Loops constitute a big hurdle to programming students (Dehnadi and Bornat 2006). With MTL the control mechanisms that are achieved through loops are physically visualized, making it easy for the learner to determine the role(s) of each variable.

Figure 6 shows how MTL is used to demonstrate branching or selection.

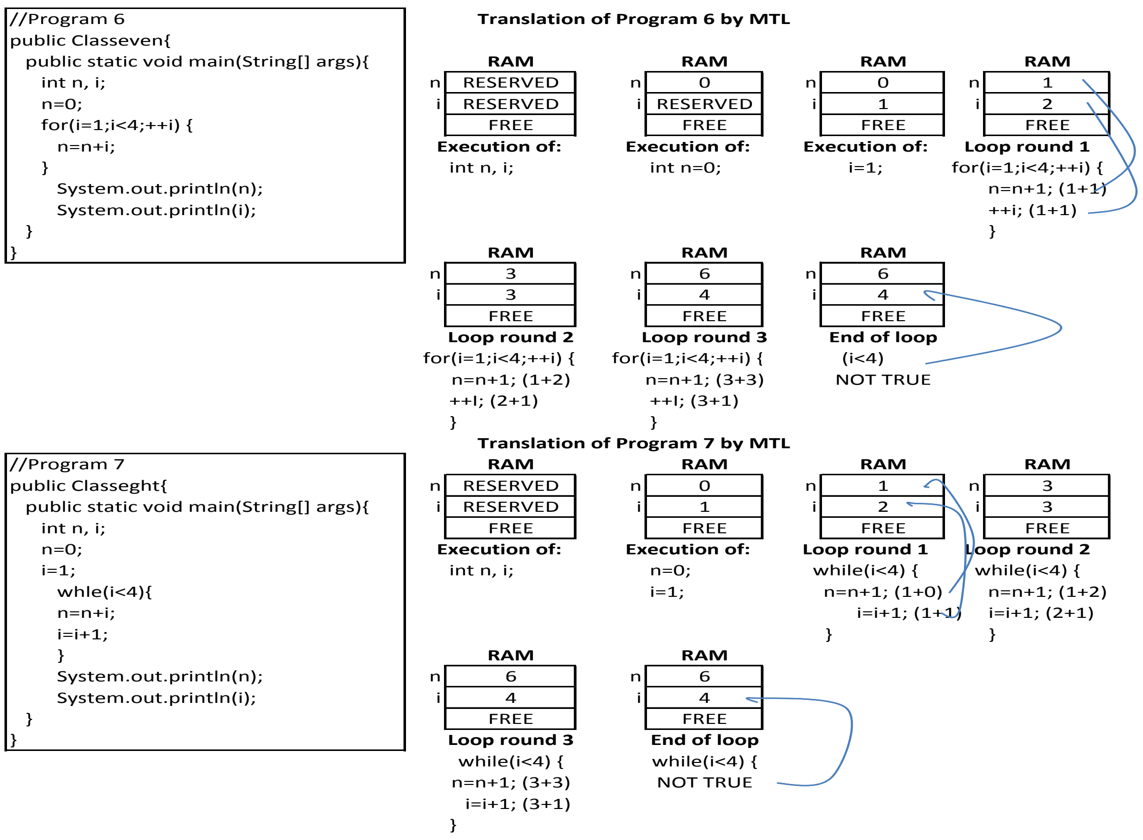


```
//Program 5
public Classif{
  public static void main(String[] args){
    int  n, i;
    double ans;
    System.out.println(Enter two nos.);
    n=TextIO.getInt();
    i=TextIO.getInt();
    if(n>i)
      ans=n/i;
    else
      ans=i/n;
    }
}
```

Figure 6- Parsing a branch by MTL

Selection in programming is another aspect not easily understood by programming students. MTL enables the learner to verify why a certain branch is taken and another branch is not taken.

Figure 7 shows similarities and differences between a *while* loop and a *for* loop.



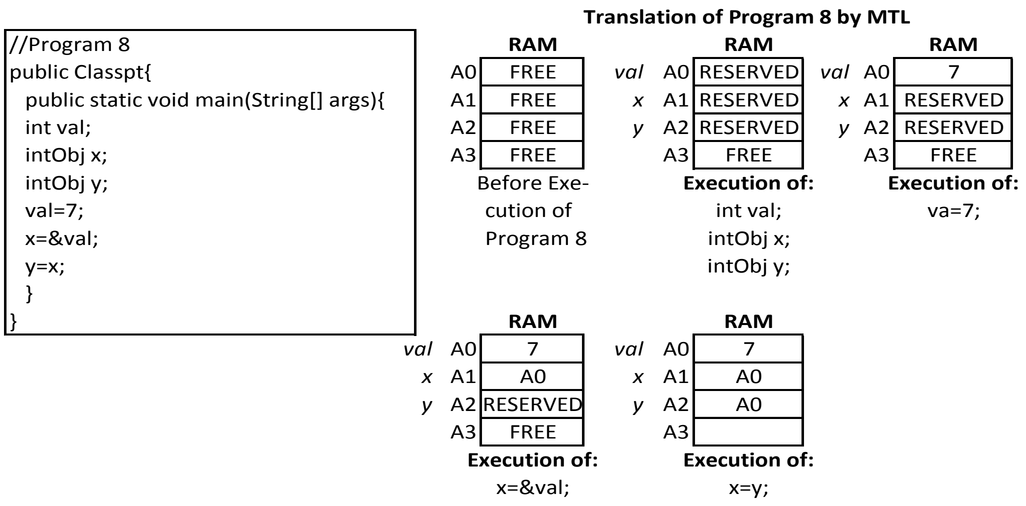
```
//Program 6
public Classeven{
  public static void main(String[] args){
    int n, i;
    n=0;
    for(i=1;i<4;++i) {
      n=n+i;
    }
      System.out.println(n);
      System.out.println(i);
  }
}
```

```
//Program 7
public Classeght{
  public static void main(String[] args){
    int n, i;
    n=0;
    i=1;
      whle(i<4){
      n=n+i;
      i=i+1;
      }
      System.out.println(n);
      System.out.println(i);
  }
}
```

Figure 7- Demonstration of similarities and differences between while and for loop constructs by MTL

Sometimes, having different loop structures which accomplish more or less the same objective is a source of confusion to students. Clear understanding of their similarities and minor differences may mitigate misconceptions. Using MTL as shown in Figure 7 different programming constructs can be compared and contrasted visibly.

Figure 8 shows how MTL is used to illustrate the concept of pointers in programming.



```
//Program 8
public Classpt{
  public static void main(String[] args){
  int val;
  intObj x;
  intObj y;
  val=7;
  x=&val;
  y=x;
  }
}
```

Translation of Program 8 by MTL

		RAM
	A0	FREE
	A1	FREE
	A2	FREE
	A3	FREE

Before Execution of Program 8

		RAM
val	A0	RESERVED
x	A1	RESERVED
y	A2	RESERVED
	A3	FREE

Execution of: int val; intObj x; intObj y;

		RAM
val	A0	7
x	A1	RESERVED
y	A2	RESERVED
	A3	FREE

Execution of: va=7;

		RAM
val	A0	7
x	A1	A0
y	A2	RESERVED
	A3	FREE

Execution of: x=&val;

		RAM
val	A0	7
x	A1	A0
y	A2	A0
	A3	

Execution of: x=y;

Figure 8- Demonstration of pointers by MTL

In Figure 8, the relationship between pointers and hexadecimal address scheme is visualized. The concept of pointers as variables that store addresses as opposed to other data types is visually demonstrated.

As demonstrated in Figures 2-8, MTL enables the author, instructor and the learner to employ the same model and the same symbol-type to show and describe the meaning of different aspects of programming. Questions like; what is a variable, why is it declared, what is the difference between an *int* and a *double* type, what is data feeding, what is data operation, what is an array, what are array subscripts, how are arrays different from simple data types, what is a loop, what mechanisms undergird looping, what are differences and similarities between different loop constructs, what are pointers, how do pointers differ from other types of variables such as integers, etc. are clearly visualized by the means of a single symbol and one model.

As demonstrated in the works of Mselle (2010, 2011a, 2011b, 2011c) and as shown in Figures 2-8, MTL approach reduces ambiguities which are rife in introductory programming.

2. Problem

Although the use of MTL in teaching programming has been reported to produce positive results (Mselle 2010, 2011a), there is no report about experiment conducted on non-novice programmers to find out if students can use MTL to learn a new programming language without an instructor during their second language study.

Many colleges and universities have undergraduate curricula which include learning to program in more than one language. It is a common practice, for example, to find schools or colleges teaching *Programming in C++* during one semester and *Programming in Java* in another semester. The practice of teaching students more than one programming languages is aimed at first making students aware of various languages that can be used in programming and secondly improving students' programming abilities. While the first motive is achieved, the second motive which is essential is hardly achieved (Dehnadi and Bornat 2006). In addition, most research works on program visualization have concentrated on testing the impact of animation tools in circumstances where the instructor is directly involved (Ben Bassat et al. 2001).

3. Objective

The objective of this study was to evaluate the impact of MTL in aiding non-novice programmers to pursue programming studies without the intervention of an instructor during their study of a second language.

To determine the impact of MTL in aiding programmers to pursue their programming lessons without the intervention of an instructor, a class experiment was conducted, where examination results from two groups were statistically compared.

4. The Experiment

To test the hypothesis that MTL can facilitate students pursue their programming classes without the help of an instructor and perform better than the students who pursue the course with the help of an instructor a sample of 108 third year students of the University of Dodoma (UDOM) was used in the experiment.

4.1 The Sample

Students learning "Java Programming" for the first time in the College of Informatics-UDOM, constituted the sample for this study. All students had pursued and passed examination in C++ Programming during their first-year program.

5. Method

A group of 14 students, from the sample of 108 students, who were studying *Java Programming* for the first time, volunteered to pursue the course using MTL. These students (n=14) constituted the experimental group. The rest of students (n=92) constituted the control group. The course syllabus covered; *variable declaration* and *types of variables*, *constants*, *data inputting*, *data manipulation*, *data outputting*, *flow of control*, *functions*, *arrays*, *strings*, *introduction to pointers* and *introduction to file handling, objects, and data encapsulation*. The duration of the course was 52 hours; with 26 hours being used for lectures and 26 hours assigned for laboratory sessions. The lead lecturer for the subject had eleven years experience in teaching programming in Java. For the purpose of this study, the experimental group agreed to use the manual entitled "Java for Novice Programmers" as their main reference book together with any other material while not attending lectures.

5.1 Materials

The materials used included; examination scripts and a programming manual in Java (Mselle 2011b). The programming manual is written to cover introductory programming which includes; variables and variable declaration, data inputting, data processing and outputting. Other topics include; flow of control (bifurcation and looping), arrays, strings, functions, files handling, pointers and objects.

5.2 Procedure

Before the beginning of the course the researcher held a meeting with all students where they were briefed about the experiment and the unique character of the *Java for Novice Programmers* manual. Students who volunteered to pursue the course without attending lectures were given the manual. They were instructed to do all assignments, laboratories and tests as the rest of the class. In the end of semester, the final examination was set by the lecturer who was in-charge of teaching the subject. Questions and solutions were reviewed by an external examiner to ensure adequacy and conformity to the syllabus. Examination scripts (which were all anonymous) were marked by the leading lecturer. Scores from the final examination (60%) were combined with scores from two tests and four assignments (40%). After the examination and publication of results, the researchers performed the statistical analysis to find out the degree of difference on scores between the experimental group and the control group.

6. Results

Statistics on the final scores for the control and experimental group are summarized in Table 1.

Groups	Number	Total scores	Means	STD
Control	92	5728	53	1.543
Experiment	14	797	56.9	2.764

Table 1- The examination scores summary

Using final scores, between the control and the experimental group, where the experimental group used MTL, to learn programming without teacher's intervention while the control group was instructed through the conventional approach, results suggest a significant difference statically (A one tailed student's T-test, R: t=1.921>1.771, p=0.05). These results agree with claim by Wilson and Moffat (2010) that programming may not be a difficult subject; rather, it is the way it is presented to the learners which breeds confusion leaving the learners with various misconceptions which frustrate the effort to learn. With the MTL, aspects such as variable declaration, assignment, variable overwriting and data operations are made obvious at the very beginning.

As demonstrated in Figures 2 through 8 and in (Mselle 2011b), with MTL, the why variables are declared is made obvious and what happens to each variable during any action is clearly distinguished. What and how various operations are carried out and the meaning of *assignment* and *roles of variables* are clearly demonstrated at every turn of the code. Regarding functions/methods; issues such as *function call* and *parameter passing* are clearly visualized. Confusion and ambiguities are potentially mitigated through visualization with MTL.

7. Discussion

The experimental group used MTL to illustrate the execution of code from the machine point of view. In its core, MTL cultivates the sense of "*I am working the machine*" on the part of the learner. On the other hand the control group had no means to visualize their codes. Lack of a tool to illustrate the effect of each line of code on the machine is a major source of misconceptions (Perkins et al. 1986).

Du Boulay et al. (1981) proposes for a tool representing a notional machine. Du Boulay et al. (1981) advises that such a device should observe simplicity, be small and have few constructs. He argues in favor of implementing a language in such a way that, either pictorial or written traces can be displayed. MTL, is a programmer-driven visualization device which bears most of these characteristics (Mselle 2010, 2011a, 2011b, 2011c), (Samurcay 1985). MTL being programmer-driven, has capabilities to transfer programming authority to the programmer while creating the sense that the machine is not responsible for the mistakes committed by the learner.

7.1 MTL vs. Machine-based Animation

Popular animation tools such as Blue J, Jeliot and Plan Ani have been reported to be effective in enhancing programming comprehension (Ben Bassat et al. 2001), (Kuittinen et al. 2003, 2008). Machine-based animations are suitable for precise close tracking through the machine. In effect, they are a plausible break through. Nevertheless, since they are entirely machine-driven they concentrate most authority to the machine at the expense of the learner. In contrast, MTL is an absolutely learner-driven parse/visualizer. Program parsing by using MTL allows the novice to visually step line-by-line through a piece of code. Using MTL the learner can actively reveal the history of variables, and how the machine reacts when each statement is executed.

Mselle (2010, 2011a, 2011b , 2011c) has shown that MTL allows the learner to play-back the code from machine point of view. MTL is a sketch-based language that provides the learner with the absolute authority over the machine. Since MTL is a learner-driven language, it is unconstrained by the initial design of the code. To its additional credit, MTL can be used in conjunction with other animators and flow charts to capture advantages suggested by Ziegler and Crews (1999). MTL is an instrument for the learner to play the role of compiler outside of machine environment, putting the learner at par with the machine on verification of the correctness or incorrectness of the program (Perkins et al. 1986). Furthermore, MTL can be integrated into the current programming materials (Mselle 2011b) a quality not yet attained by the current visualization tools (Naps et al. 2003). These factors seem to provide explanation why the results observed in the experiment are positive.

8. Conclusion

The objective of this study was to test the impact of MTL when used as a learning tool without the intervention of a lecturer. Specifically, MTL has been proved to be a handy sketch language for learners to parse, track, debug and understand their programs without teachers' or machine intervention. Although the subjects of the experiment were not absolute novices their characteristics do not substantially differ from those of absolute novices because all of them were learning Java for the first time. Initial results are encouraging though far from conclusive.

There are, obviously, some shortcomings in this study. The sample size is too small to justify generalization. The population is taken from one university. The students of the experiment group could have been better than those of the control group.

9. Recommendations

More experiments in different settings should be carried out with a much bigger and diverse sample to confirm the effectiveness of MTL. More areas of research on effectiveness of MTL in distance learning, and different age groups are open for future investigation.

References

Ben-Ari, M. and Sajaniemi, J. (2003) Role of Variables from the Perspective of Computer Science Educators. DOI=http://cs.joensuu.fi/pub/Reports/A-2003-6.pdf.

Ben Bassat, L.R., Ben Ari, M. and Uronen, P. (2001) An Extended Experiment with Jeliot 2000. In Proceedings of The First International Program Visualization Workshop. University of Joensuu, Pavoo Finland, September, 2001, 131-140.

Dehnadi, S. and Bornat, R. (2006) The Camel has Two Humps (working title). School of Computing, Middlesex University, UK.

Du Boulay B., O'Shea, T. and Monk, J. (1981) The Black Box Inside the White Box: Presenting Computering Concepts to Novices. *International Journal of Man-Machine Studies*, 14, 237-249.

Du Boulay, B. (1986) Some Difficulties of Learning to program. *Journal of Educational Computing Research*, 2(4), 459-472.

Hundhausen, C. D. and Brown, J. L. (2007) What you See is what you Code: A 'live' Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*, 18(1), 22-47.

Kuittinen, M., Tikansalo, T. and Sajaniemi, J. (2008) A study of the Development of Students' Visualizations of Program State During an Elementary Object-Oriented Programming Course. *ACM Journal of Educational Resources in Computing,* 7(4).

Kuittinen, M. and Sajaniemi, J. (2003). First Results of an Experiment on Using Roles of Variables in Teaching. In M. Petre & D. Budgen (Eds) Proc. Joint Conf. EASE & PPIG 2003.

Mselle, L. (2010) The Impact of RAM Diagrams in Enhancing Comprehension in Programming: Class Experiment. http://www.ppig.org/papers/22nd-Teach-4.pdf

Mselle, L. and Mmasi, R. (2011a) The Impact of MTL on Reducing Misconceptions in Programming. DOI=http://dl.acm.org/citation.cfm?id=1999901&dl=ACM&coll=DL&CFID=38905039&CFTOKEN=61705367.

Mselle, L. (2011b). *Java for Novice Programmers*. LAP LAMBERT Academic Publishing, Berlin.

Mselle, L. (2011c). “Using Formal Logic to Formalize MTL on the Mould of RTL.” PPIG 2011, The University of York, UK.

Naps, T., R¨oßling, G., Almstrum, V., Dann, W. Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Vel´azquez-Iturbide, A. (2003). Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin*, 35(2), 131–152.

Perkins, D. N., Hobbs, H. R, Martin, F. and Simmons, R. 1986. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, 2(1), 37-55.

Samurcay, R. (1985) The Concept of Variable in Programming: Its Meaning and Use in Problem-Solving by Novice Programmers. *Education Studies in Mathematics*, 16(2), 143-161.

Wilson, A. and Moffat, D. (2010) Evaluating Scratch to Introduce Younger Schoolchildren to Programming. In Proceedings of the 22nd Annual Psychology of Programming Interest Group (Universidad Carlos III de Madrid, Leganés, Spain, September 19-22, 2010). Joseph Lawrence and Rachel Bellamy, editors.

Ziegler, U. and Crews, T. (1999). An Integrated Program Development Tool for Teaching and Learning how to Program. In Proceedings of the 30th SIGCSE Symposium, March 1999, 276-280.

Paper Session 6

Tools and Their Evaluation

Evaluation of Subject-Specific Heuristics for Initial Learning Environments: A Pilot Study

Fraser McKay

School of Computing
University of Kent
fm98@kent.ac.uk

Michael Kölling

School of Computing
University of Kent
mik@kent.ac.uk

Keywords: POP-I.B. design of environments; POP-II.A. novices; POP-V.B. interviews.

Abstract

Heuristic evaluation is a “discount” technique for finding usability problems in well-established domains. This paper presents thirteen suggested heuristics for initial learning environments (ILEs). To investigate the usefulness of these heuristics to other developers, we conducted a pilot study that compared two groups of evaluators: one using an older, generalised set of heuristics from the literature, and one using our domain-specific heuristics. In this study, we compare not just the number of problems found, but the way in which the problem reports were expressed. There was a significant difference in the length of written comments when problems were found (those from the new set being longer). New-set reviews touch on more themes – many make suggestions about what would improve the problem; many comments refer to a suggested cause-and-effect relationship. As designers, we find this detail helpful in understanding problems. Quantitative data from this study is not large enough to support any robust conclusions about the relative thoroughness of the heuristics at this time, but we plan to use lessons learned from this study in a larger version shortly.

1. Introduction

There are many initial learning environments (ILEs), using different languages, available for educators to choose from. With the increase in the number of systems on the market, it is becoming increasingly important to be able to make informed decisions, backed by formal argument, about the relative quality of these tools. We advocate a significant increase in more formal or semi-formal evaluations of the quality of educational programming systems.

Heuristic evaluation is a method that is both useful and practical to make such an assessment of many aspects of educational programming systems. Some of the most frequently used sets of heuristics are application-area neutral. They specify goals and guidelines for software systems in general. However, these do not cover all of the problem areas that we are aware of in novice programming. In this case, application-area specific requirements can be included in the heuristics, and deeper insights might be gained. This can be done for a variety of application areas, and has been attempted for programming before (Pane & Myers, 1996; Sadowski & Kurniawan, 2011), though these heuristics have not then been evaluated, or validated, themselves.

In this paper, we propose a set of heuristics specific to ILEs. These heuristics should improve heuristic evaluations of such systems by combining a number of relevant aspects and criteria not formulated in any other set of heuristics. The proposed heuristics include aspects of general usability, as well as aspects (e.g. motivational and pedagogical effects) relevant to our specific application domain. To investigate their usefulness for other developers, we conducted a pilot study that compared two groups of evaluators: one using a popular, generalised set from the literature, and one using our domain-specific heuristics. In this study, we compared not just the number of problems found, but the way in which the problem reports were expressed. As designers, we find this detail helpful in understanding problems. Quantitative data from this study is not large enough to support any robust conclusions about the relative effectiveness of the heuristics at this time, but we plan to use lessons learned from this study in a larger version soon.

1.1. Background

Heuristic evaluation was introduced as a "discount" method of analysing user interfaces without full user testing (Nielsen & Molich, 1990). When performing a heuristic evaluation, experts compare interfaces to sets of heuristics – general principles that describe aspects of an ideal system. Nielsen lists ten separate heuristics, formulated in the style of positive guidelines such as "*The system should always keep users informed about what is going on, through appropriate feedback within reasonable time*." (Nielsen, 2005b).

Heuristics can also be used in designing interfaces. In this case, designers pay conscious attention to the set of heuristics during the design process – taking Shneiderman's "Eight Golden Rules", for example (Shneiderman, 1997).

Nielsen's heuristics are intended to be generally applicable to a wide variety of software interfaces, and other authors have identified more specialised, domain-specific sets of heuristics. Pane & Myers (1996) defined a set of heuristics aimed at ILEs. These extend Nielsen's set with additional heuristics aimed at identifying issues with problems specific to this target domain.

2. Related work

2.1. Cognitive Dimensions and CD questionnaire

The research presented here, being concerned with programming, also conceptually overlaps with the Cognitive Dimensions (CD) framework (Green, 1989). The dimensions describe concepts relevant to cognition and HCI in programming notations, but – unlike heuristics – are not phrased as instructions. There are some attempts in the literature to use them as heuristics (Sadowski & Kurniawan, 2011). There are structured forms of cognitive dimensions evaluation (Blackwell & Green, 2000; Blackwell & Green, 2007), but they are, overall, less restrictive than some heuristic evaluation research (in the "classical" Nielsen sense, at least). Open-ended comment is more encouraged. The cognitive dimensions provide descriptions of concepts such as "viscosity" (resistance to code changes) and "secondary notations" (such as colour and spacing) - a vocabulary for researchers to use when discussing systems.

2.2. Testing/evaluating inspection methods

Heuristic and CD evaluations, like other "expert-based" methods, rely heavily on human evaluators' findings; in this case, driven by a set of heuristics. Therefore, it is critically important that the heuristics are valid in themselves. Validity is generally taken to mean "shown to be useful in uncovering actual usability problems" (Blandford & Green, 2008). Pane & Myers's heuristics are argued from the literature (of the time), but are not evaluated in practice. Nielsen & Molich's original heuristics were tested in user studies to refine them and to assess their validity (Nielsen & Molich, 1990). Hartson, Andre & Williges (2001) present a method of comparing evaluation methods, including three standard metrics (thoroughness, validity and reliability) based on other work. Sears (1997) formally defines the concepts of *thoroughness* and *validity* for evaluating inspection methods. Thoroughness is a measure of how many problem candidates, in a set of real "reference problems", a method finds. Validity is a measure of how many of the candidate problems found are "real" problems (as opposed to false positives).

3. The new heuristics

The need to define new domain-specific heuristics arose during an on-going effort to design a novel beginners' programming tool.

To aid in the design of the new system, we initially applied existing heuristics as design guidelines. We also conducted evaluations using the cognitive dimensions framework, in their role as discussion tools, to study some existing ILEs. Pane & Myers's heuristics seemed a good candidate tool for our purposes, at first, since they are specifically designed for these kinds of system. However, a major drawback we encountered with these heuristics is their length. Using them in depth, they produced very long – and quite cumbersome, thus less useful – evaluations. In total, Pane & Myers have 29 heuristics, grouped under eight of Nielsen's headings. Differences between the heuristics were not always clear-cut; there is duplication across some categories, and using them to categorise some of the problems is distinctly difficult. Some problems seemed to fit two categories equally well, and others did not neatly fit any. Yet, when using Nielsen's original heuristics, in order to avoid the length of Pane & Myers', the evaluation missed some crucial areas that we consider important for early learners' programming systems. While shorter, these heuristics do not achieve the same domain specific precision.

This experience led to the definition of a new set of heuristics with a number of specific goals of improvement. The goals were:

(1) Manageable length: The number of heuristics should be limited; it should be much closer to Nielsen's 10 rules, or Shneiderman's eight, than Pane & Myers' 29.

(2) Domain-specific focus: The heuristics should cover aspects specific to novice programming.

(3) Avoidance of redundancy and overlap: As much as possible, problems should fit clearly into a single category.

(4) Clarity of categorisation: The categories should be clear to evaluators, cover all possible issues and be easily distinguished.

Blandford and Green (2008) suggest requirements that a new HCI research method should be useful ("it should reveal important things…"), usable, and used ("would ultimately be used by people other than its developers and their close friends").

After identifying target candidates for heuristics based on known problem instances, the authors evaluated the resulting sets by applying them to existing systems from the target domain. This process was repeated over several iterations to refine and improve the set.

The systems used for systematic evaluation of the heuristics were Scratch, Alice, BlueJ, Greenfoot and Visual Basic. Partial evaluations using the new heuristics have also been carried out for Lego Mindstorms NXT, StarLogo TNG, Kodu, and C#. The first group of systems are the ones that were used for testing each iteration of the heuristics; the second group are systems that we have evaluated using the heuristics. We have also used the heuristics to evaluate new designs, which we are working on as part of the wider project to design a new system.

3.1. New set

There are thirteen heuristics in our set. They are used to evaluate a complete ILE, including the environment aspects and the programming language aspects. The new heuristics are neither sub- nor superset of any previously existing set, and have been developed from scratch using well-established principles from the literature in this development domain.

(1) **Engagement**: The system should engage and motivate the intended audience of learners. It should stimulate learners' interest or sense of fun.

(2) **Non-threatening**: The system should not feel threatening in its appearance or behaviour. Users should feel safe in the knowledge that they can experiment without breaking the system, or losing data.

(3) **Minimal language redundancy**: The programming language should minimise redundancy in its language constructs and libraries.

(4) **Learner-appropriate abstractions**: The system should use abstractions that are at the appropriate level for the learner and task. Abstractions should be driven by pedagogy, not by the underlying machine.

(5) **Consistency**: The model, language and interface presentation should be consistent – internally, and with each other. Concepts used in the programming model should be represented in the system interface consistently.

(6) **Visibility**: The user should always be aware of system status and progress. It should be simple to navigate to parts of the system displaying other relevant data, such as other parts of a program under development.

(7) **Secondary notations**: The system should automatically provide secondary notations where this is helpful, and users should be allowed to add their own secondary notations where practical.

(8) **Simplicity/Clarity**: The presentation should maintain simplicity and clarity, avoiding visual distractions. This applies to the programming language and to other interface elements of the environment.

(9) **Human-centric syntax**: The program notation should use human-centric syntax. Syntactic elements should be easily readable, avoiding terminology obscure to the target audience.

(10) **Edit-order freedom**: The interface should allow the user freedom in the order they choose to work. Users should be able to leave tasks partially finished, and come back to them later.

(11) **Minimal viscosity**: The system should minimise viscosity in program entry and manipulation. Making common changes to program text should be as easy as possible.

(12) **Error-avoidance**: Preference should be given to preventing errors over reporting them. If the system can prevent, or work around, an error, it should.

(13) **Feedback**: The system should provide timely and constructive feedback. The feedback should indicate the source of a problem and offer solutions.

We have categorised the heuristics into four sets: those that affect the **System** as a whole, the **Mental Model**, the **User Interface**, and **Interaction** with the system. The *System* and *Mental Model* categories relate mostly to functionality, while the *User Interface* and *Interaction* categories are concerned with the presentation of the system. Figure 1 shows the relationship of these sets.

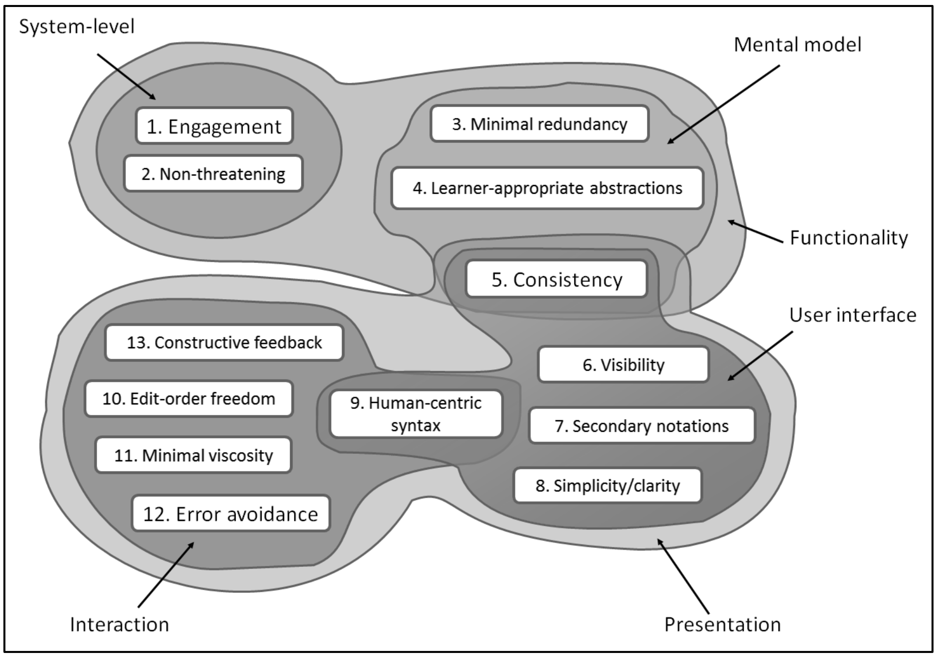


Figure 1. How the heuristics apply.

4. Method

Participants were asked to perform a heuristic evaluation of Scratch and Greenfoot, using a given set of heuristics. They were given tasks to carry out as part of their evaluation and record any usability problems they found. Each participant was assigned to use either the new heuristics, or Nielsen's heuristics (n=6, n=3, respectively). Participants were not told that the purpose of the study was to evaluate the heuristics themselves. They were not aware that other participants might use a different set of heuristics. All participants first completed the Scratch task. For various reasons, only five completed the Greenfoot task, so the Greenfoot task was excluded from this study. The participants are described in Table 1.

Participants were given a short "refresher" talk on the heuristic evaluation process, as part of their instructions. Between one and three participants at a time worked independently, in different parts of a large, quiet room. As well as a copy of the task, participants were given a sheet of paper listing the heuristics to use. The new heuristics were as presented above, and the existing set was taken from Nielsen (2005b). Both sets were formatted in the same plain style, and the title did not indicate anything special about there being multiple sets of heuristics. Participants were asked to record usability problems on paper pre-printed with columns for recording the problem, and which heuristic that problem breaks.

Twelve participants initially volunteered. However, due to three either withdrawing or not being available on the day, only nine participants actually participated in the study. The undergraduates were recruited through an email list following on from an HCI module. The postgraduates were recruited through a departmental email to research students. All of the participants were experienced programmers, and understood that some programming knowledge was required for the task(s). Though all knew *of* Scratch, only one had actually used it before – that reviewer had been taught using Scratch through secondary school. As seen later, the user discovered significantly more problems than others using the same heuristics.

We did not disclose until afterwards that the purpose of the study is to compare two sets of heuristics.

Reviewer	Profile	Heuristics	Interview
A	Undergraduate, had not used Scratch before	New	Y
B	Undergraduate, had not used Scratch before	New	Y
C	Postgraduate, had not used Scratch before	New	N
D	Postgraduate, had not used Scratch before	New	Y
E	Postgraduate, had not used Scratch before	New	N
F	Postgraduate, had not used Scratch before	New	N
G	Postgraduate, had not used Scratch before	Nielsen	Y
H	Postgraduate, had not used Scratch before	Nielsen	Y
I	Undergraduate, had long-term school experience with Scratch	Nielsen	Y

Table 1. Reviewer profiles.

At the end of the sessions, six participants who had time (three with each set) participated in short, informal individual interviews. The subject of the interview were the heuristics themselves, rather than the system under evaluation. The participants were asked open questions about ease or difficulty to understand and apply the heuristics; how they would describe the heuristics in their own words; and whether they thought any of the problems they found would have fitted with none of the heuristics, or more than one.

5. Results

5.1. Number of problems found

There was no statistically significant difference in the number of problems found by each group.

Sears's (1997) and Hartson et. al's (2001) validation methods rely on comparing "thoroughness" and "validity" per review(er). Using Equations 1 and 2, thoroughness and validity can be calculated for the Scratch problems (Table 3). However, the small size makes any real statistical analysis of *t* and *v* of little actual value in this case. In the nine results tested, there is little difference in thoroughness in the two groups. Measuring *v* does not produce anything notable with these results – none of the comments were clear "false positives". If the number of false positives is low (but non-zero), it would also have little or no reliable predictive value in this small study.

Group	N	Median	Avg Rank	Min	Max
New	6	0.177	5.167	0.065	0.290
Old	3	0.161	4.667	0.065	0.258

Table 2. U-test of thoroughness (p=0.794).

$$t = \frac{\#\,Real\ problems\ found}{\#\,Total\ problem\ set} \qquad v = \frac{\#\,Real\ problems\ found}{\#\,Suggested\ problems\ found}$$

Equation 1 and Equation 2.

Figure 2 is a diagrammatic representation of problems found per user. In the figure, every column represents a usability problem, and every row represents one reviewer. If that reviewer found the particular problem, the grid is marked with a black square. The first six evaluators used the new heuristics, and the last three used Nielsen's set (marked by a horizontal line). Similar diagrams are used in Nielsen's online material (Nielsen, 2005a). Problems to the left of the vertical centre-line were found by more than one evaluator. Problems on the right were only found by one person. Problems are numbered (horizontal, bottom).

Figure 3 summarises the problems found by each heurstics set. Any problem which was found at least once, for the given set of heuristics, is marked in black. Some problems were found with only one set, and not the other. A total of 9 problems were identified only with the new set (problems 6, 8, 10, 14, 15, 16, 17, 19, 21). Four problems were found only with the Nielsen set (problems 12, 13, 18, 20). Of problems which were identified by at least two evaluators, three were uniquely identified using the new set (6, 8, 10), and none with Nielsen's set.

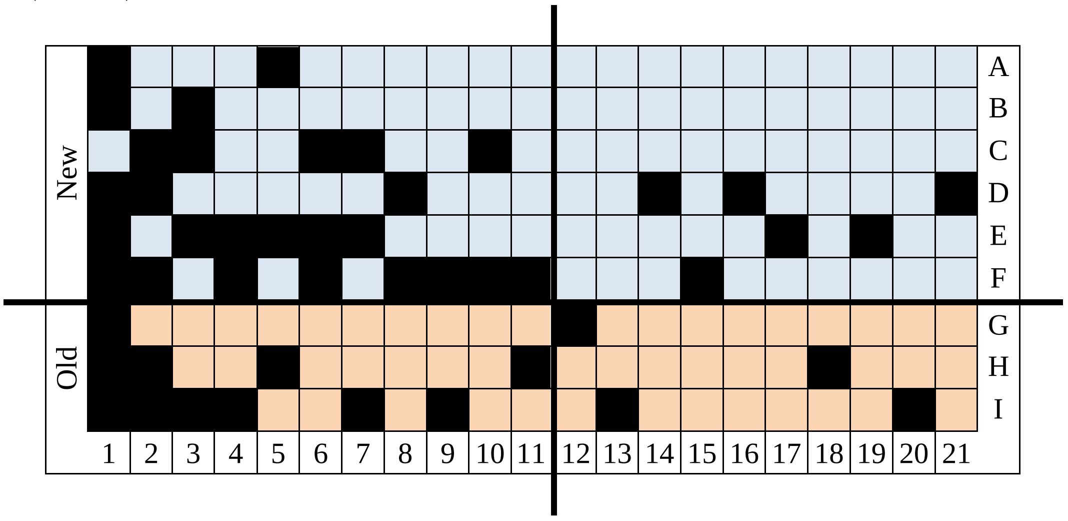
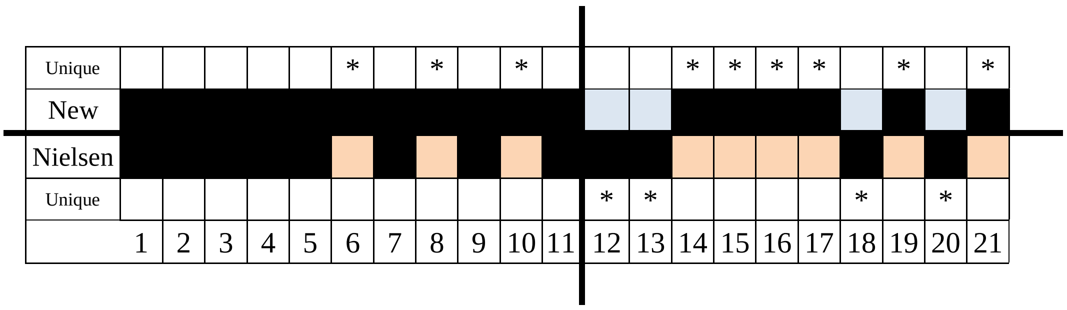


Figure 2. Problems identified per reviewer. Problems (horizontal) are numbered, and participants' (vertical) identifiers correspond to Table 1.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Unique						*		*		*				*	*	*	*		*		*
New	■	■	■	■	■	■	■	■	■	■	■	🟦	🟦	■	■	■	■	🟦	■	🟦	■
Nielsen	■	■	■	■	■	🟧	■	🟧	■	🟧	■	■	■	🟧	🟧	🟧	🟧	■	🟧	■	🟧
Unique												*	*					*		*	

Figure 3. Problems identified per heuristics set. Problems found uniquely with one set are marked.

5.2. Comments made

The reviewers made 29 written comments, with the new heuristics; and 19 with Nielsen's set. The mean numbers of comments each were 6.0 and 6.18, respectively.

Though the number of comments per-reviewer was similar in most cases, the individual comments varied greatly in length (as well as giving additional qualitative information – see below). The values in Table 3 show the differences in numbers of characters. The new-set comments are generally longer.

Group	**N**	**Mean**	**Min**	**Max**
New	29	111.172	28.0	499.0
Nielsen	19	52.105	25.0	113.0

Table 3. Comment length by group (t-test p-value = 0.018)

Two typical comments, from different reviewers, are,

> "H8 simplicity. There is something strange with the way a whole block of blocks is moved, and one has to separate the first of these in."

> "H10, 8. I can not delete an [sic] statement by right clicking it if it's in between a sequence therefore I had to do 2 steps (separated them, deleted and put the rest back together)."

The longest comments with the new heuristics are paragraph-length. For instance,

> "H11. Dragging and dropping the blocks felt slow when doing a sequence of 10, doing the edit on both cars separately was annoying, and also replacing blocks was annoying, though I found a trick that helped (first add the new block beneath the old block, then drag it (and all thus all that follows it) to the right spot above the old block, then remove the old block which as it's the last item, doesn't uncouple the other blocks). Still, a right-click--delete option would have saved a lot of time."

In contrast, a longer comment from the Nielsen group reads,

> "H8. Some of the control colours are very similar, slowing down rate that you can guess where commands are stored."

(referring to Nielsen's heuristic 8 – aesthetic and minimalist design).

We are aware that having longer comments, per se, is not the aim of the new heuristics. However, in addition to being longer, the comments contained more detailed and descriptive feedback. Below, the comments are discussed in more detail.

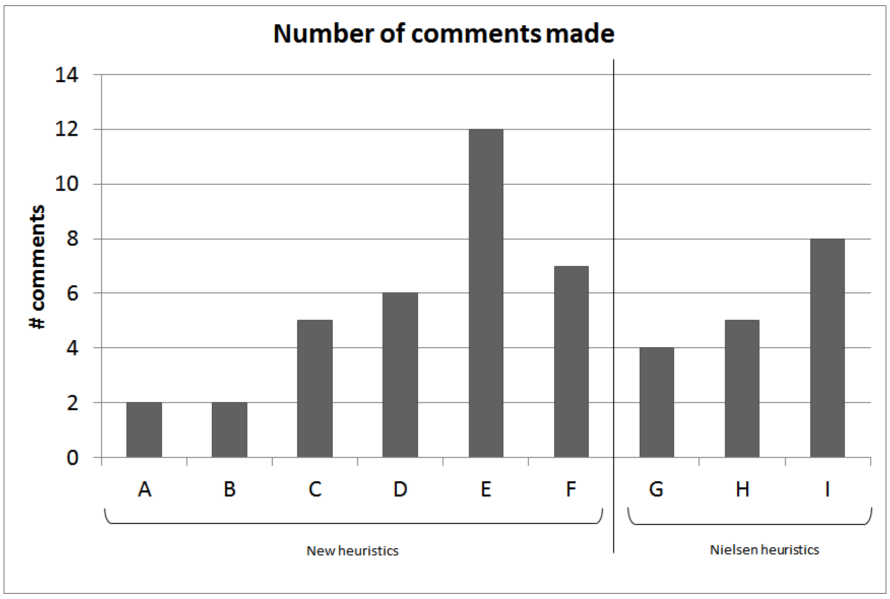


Figure 4. Written comment distribution.

5.3. Comment types

All the comments were coded: for the problems they described, and the nature of the comment. The coding for types of comment was open – there were no predetermined categories of response that we were determined to fit eventual results into.

Even though not explicitly instructed to do so, many of the evaluators included suggestions (sometimes quite detailed) in their problem reports. The following are examples of suggestions, all from the new heuristics (emphasis added):

> "Had to switch tabs a lot, **process would be more efficient if they were rearranged, or more visible at once**".
>
> [About there being a work-around to shuffling blocks] "**Still, a right-click delete option would have saved a lot of time**".
>
> "maybe have a way of integrating the left hand, e.g. by being able to switch between block-menus (which is done by clicking a button in the top-left square) with the key-board?".
>
> "[Would be] **Easier to make changes then duplicate rather than editing each one**".
>
> "**Maybe try, and balance the sprite pictures so there are a bit more girlish ones…**"
>
> "When the last change is compiled the button is still active (even though there is a state indicating). **However it would be better to block it**".

Reviewers made roughly the same number of comments using each set. However, there was a marked difference in suggestion frequency. There was only one suggestion made with the existing heuristics, out of 19 comments.

Some of the comments with the new set were phrased as cause-effect pairs. The problem was described as in other reports, but was also linked to an underlying issue or concept in the system. For example, the following three:

> "No types on the variables, **so** string/int confusion of 90/ninety could not be prevented".
>
> "[…] **as it's the last item**, doesn't uncouple other blocks".
>
> "The dragging blocks is very slow, [because] having to use the mouse a lot makes me use the right hand a lot more than the left."

5.4. Post-interviews

Qualitative data from the interviews refer to the nature of the heuristics themselves. Some of the comments queried the wording of particular heuristics (in both sets). For example:

- The wording of "intended audience" in the new set was ambiguous (it having been left open to reflect the range of systems it could apply to) (New set: 1);
- "Viscosity" (New set: 11) was a more difficult concept to grasp than, for example, error-proneness;
- "Aesthetic and minimalist design" (Nielsen: 8) did not quite convey, in their opinion, that "dialogues should not contain information which is irrelevant or rarely needed" (Nielsen, 2005b);
- They could not understand (or in one case, was sure there was none) the difference between "simplicity", in the sense of "visual simplicity" (New set: 8) and viscosity.

Although it was not the main purpose of the interview, a few of the participants pointed out an additional usability problem during interview. These were not solicited by the researcher, but typically expressed as something that the participant was unsure whether they should record, or say out loud in the interview. We decided it would be counterproductive to actively ignore a usability problem once it had been suggested; the researcher wrote on the interview notes that a problem was mentioned aloud. However, these are not included in the written comments being analysed here, and remain recorded as part of an actual review of the two systems, not the heuristics.

6. Discussion

6.1. Reflections on the study

Ultimately, this study is not large enough to definitively answer a question about problem coverage/discovery. The "evaluator effect" – the variability that necessitates using multiple reviewers – is well reported in the literature (Hertzum & Jacobsen, 2001). Within the nine participants in this study, we can see variation in *how many* problems different users found with the same heuristics. The small sample was then more open to skew. As mentioned, the Nielsen-group evaluator who had been taught to use Scratch at school found more problems than the others in their group. In a larger group, this might only have been a single outlier. Alternatively, it might have shown *no* relationship between specific prior experience and problem discovery. The evaluators from the other group who found a similar number of problems had not previously used Scratch. Quantitative data from this study is not large enough to support any robust conclusions about the relative thoroughness of the heuristics at this time.

However, this study does show that the new heuristics have found some actual, verifiable problems, at least as well as Nielsen's did, and possibly better. We know from the literature that different methods, like heuristics or user testing, will generally find different problems, and that a more complete picture of a system is obtained by inspecting it with multiple techniques (de Kock, van Biljon, & Pretorius, 2009; Hertzum & Jacobsen, 2001). In heuristic evaluations, the recommended number of evaluators varies (Cockton & Woolrych, 2001; Nielsen, 2005a).

Through qualitative analysis, we found extra kinds of helpful information embedded in the problem reports. This fits with the observation that problem reports made with the new set were significantly longer. In addition to comments describing a problem, there were many positively-phrased comments with both sets. These are not usability "problems", but it would be unproductive to discard them altogether. For the purposes of this paper, they are not included – in the graph of problem coverage, for example – but in a "real" evaluation they would certainly provide additional heuristics-based feedback on the system being tested. The "checkerboard" style summary used in a stricter test would obscure this entire category of information.

In those comments that did describe a *problem*, there were two "special" forms of comment observed: problems with suggested fixes; and problems where the reviewer linked two issues as having a cause-and-effect relationship. Examples of both were quoted in the results section of this paper.

The meaning of "viscosity" was an issue in several interviews. One participant, for instance, was quite confident that,

> "viscosity and simplicity are the same anyway."

The same participant described a problem related to viscosity (and correctly identified as such by other participants) using the word "simplicity", and categorised it under the heuristic relating to visual simplicity/clarity:

> "H8 simplicity. There is something strange with the way a whole block of blocks is moved, and one has to separate the first of these in."

Another interviewee understood that the word meant "thick", or "sticky", and understood what we meant by viscosity in this context, but did not think the choice of word "was quite right".

However, this was not universal – some clearly understood what viscosity meant. For example,

> "…first add the new block beneath the old block, then drag it (and all thus all that follows it) to the right spot above the old block, then remove the old block…"

> "Blocks "stick" together by default."

6.2. Heuristics as a set

The new heuristics may evolve in time, but they are adequate as a useful, and usable, starting point. We believe they are an improvement over Pane and Myers's (1996) draft set in the same domain. Some interview participants commented finding it difficult to categorise problems. Miscategorising is known to occur in heuristic evaluations, generally (Cockton & Woolrych, 2001).

It may be beneficial to investigate severity ratings – part of the "canonical" Heuristic Evaluation method by Nielsen, used sometimes but not always (Sim, 2011). We did not use them in this study because we felt they would simply complicate the comparison; we do not know if having to include a severity rating makes the evaluators more, or less, likely to report a problem, and whether this varies depending on the heuristic. It is also important that we see these heuristics as being useful during design, not just a traditional evaluation. In that case, it seems unnecessary for a designer to numerically rate problems that they have found themselves.

7. Conclusion and future work

Usability heuristics are used in the evaluation of a range of systems. This paper compares the characteristics of problem reports made using either new, domain-specific, heuristics, or an established set of generalised heuristics from the literature in the evaluation of an ILE. Within this pilot study, we found a significant difference in the kinds of comments reviewers made: reviews made with the new set typically included longer, more detailed, comments than those with the general set. Through interviews, we gathered the reviewers' opinions about the structure of the new heuristics.

As expected with a small group, the quantitative data is not definitive. The structure of this study proved that problems can be found, in these systems, with these sets of heuristics, and by these kinds of evaluators. An extended version of the experiment is planned later in the current academic year, using larger groups of evaluators.

8. References

Blackwell, A. F., & Green, T. R. G. (2000). A cognitive dimensions questionnaire optimised for users. Paper presented at the *Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group*, 137-152.

Blackwell, A. F., & Green, T. R. G. (2007). *A cognitive dimensions questionnaire.* Retrieved, 2011, from http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf

Blandford, A., & Green, T. R. G. (2008). Methodological development. In P. Cairns, & A. L. Cox (Eds.), *Research methods for human-computer interaction* (pp. 158-174) Cambridge University Press.

Cockton, G., & Woolrych, A. (2001). Understanding inspection methods: Lessons from an assessment of heuristic evaluation. In A. Blandford, J. Vanderdonckt & P. Gray (Eds.), *People and computers XV* (pp. 171-192) Springer.

de Kock, E., van Biljon, J., & Pretorius, M. (2009). Usability evaluation methods: Mind the gaps. Paper presented at the *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists,* 122-131.

Green, T. R. G. (1989). Cognitive dimensions of notations. *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group,* 443-460.

Hartson, H. R., Andre, T. S., & Williges, R. C. (2001). Criteria for evaluating usability evaluation methods. *International Journal of Human-Computer Interaction, 13*(4), 373-410.

Hertzum, M., & Jacobsen, N. E. (2001). The evaluator effect: A chilling fact about usability evaluation methods. *International Journal of Human-Computer Interaction, 13*(4), 421-443.

Nielsen, J. (2005a). *How to conduct a heuristic evaluation.* Retrieved September, 2012, from http://www.useit.com/papers/heuristic/heuristic_evaluation.html

Nielsen, J. (2005b). *Ten usability heuristics.* Retrieved 08/12/09, 29, from http://www.useit.com/papers/heuristic/heuristic_list.html

Nielsen, J., & Molich, R. (1990). Heuristic evaluation of user interfaces. Paper presented at the *CHI '90 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Empowering People*, 249-256.

Pane, J. F., & Myers, B. A. (1996). *Usability issues in the design of novice programming systems.* (No. 96-132). Pittsburgh, Pennsylvania: Carnegie Mellon University.

Sadowski, C., & Kurniawan, S. (2011). *Heuristic evaluation of programming language features.* (No. UCSC-SOE-11-06). Santa Cruz, CA: University of California at Santa Cruz.

Sears, A. (1997). Heuristic walkthroughs: Finding the problems without the noise. *International Journal of Human-Computer Interaction, 9*(3), 213-234.

Shneiderman, B. (1997). *Designing the user interface: Strategies for effective human-computer interaction* (3rd ed.). Boston, MA, USA: Addison-Wesley.

Sim, G. (2011), Evaluating heuristics. *Interfaces, 89*, 16-17.

Exploring the design of compiler feedback

Thibault Raffaillac

School of Computer Science and Communication
KTH, Royal Institute of Technology, Stockholm
traf@kth.se

Keywords: POP-I.B. transfer of competence, POP-II.B. maintenance, POP-III.D. compiler feedback

Abstract

Nowadays, programmers willing to start optimising their code must undergo a lengthy interaction with dedicated profiling tools. This paper proposes as an alternative to make compilers generate feedback messages aimed at explaining how they understand the code, and how it could be improved. The study aims at foreseeing the technical integration of feedback notifications in modern compilers, as well as sketching how Integrated Development Environments (IDE) would display them.

A first comparison of three related works enables the core differentiators to be highlighted: letting the compiler inform where code is actually fine and does not need any refinement, displaying the notifications along the relevant source lines rather than in a separate interface, insisting on the absence of artificial intelligence, and evaluating the importance of each message to filter in a handful. Then, a preparatory field study is carried to observe different programmers and poll their receptiveness to a compiler feedback. The findings relate the usefulness of optimisations' suggestions to fit where users lack expert knowledge, the existence of dormant interrogations calling for serendipitous information retrieval, and the avoidable mistakes inherent to Message of the Day windows.

Three prototypes are designed to embody three distinct approaches, using Web tools to provide an appearance close to code editors along with decent interactivity. With the help of a new user study with the prototypes, a final set of refinements is discussed so as to shape a coherent result and differentiate it further: users can create and share sets of feedback messages to supplement the ones included in their compiler, a list of rules is provided to help designers compose the messages, an emphasis is laid on transparency to help exhibit the absence of artificial intelligence, and the heuristic used to evaluate the importance of messages is sketched.

1. Introduction

Compiling a program nowadays is simple. Once the source code itself is written, a single action is needed to turn it to an executable file. With an IDE such as Eclipse or Microsoft Visual Studio, it is synonym with clicking on a "Build" button. With a command-line compiler such as the GNU Compiler Collection (GCC), it is computed with a single command. Past the errors and warnings, as soon as a working executable is output the compiler will not provide any more interaction.

When it comes to optimization, however, the procedure becomes trickier. By setting the proper options and command-line flags, it is normally handled transparently by the compiler, yet in practice this support is irregular (see Aho, Lam, Sethi, & Ullman, 2006, for a technical overview of modern compilers). While instructions scheduling and register allocation are decently achieved nowadays[1], improvements such as making parallel loops or exploit the locality of memory accesses *will* require tuning specific options, or the source code itself[2].

On the other hand, a significant knowledge gap separates the programmer from the compiler. For the

[1] With the example of GCC, see http://gcc.gnu.org/wiki/InstructionScheduling and http://gcc.gnu.org/wiki/RegisterAllocation (both accessed 03.09.2012).
[2] See the example of GCC at http://gcc.gnu.org/onlinedocs/libgomp/Enabling-OpenMP.html, and for Intel at http://software.intel.com/articles/automatic-parallelization-with-intel-compilers/ (both accessed 03.09.2012).

former, the language syntax[3], the diversity of architectures and systems, and the basic skills required for a software engineer (Kreeger, 2009; Lethbridge, 2000), are potentially overwhelming. For the latter, as quoted from Bose (1988), *the compiler is not designed to fully "understand" the high-level, algorithmic intentions expressed by the user in his (or her) source code*.

As a consequence, it is believed that users do not obtain the performance and security they should expect from their programs. This paper is based on a Degree Thesis which investigated the design and integration of feedback messages from the compiler, to leverage opportunities of tuning and improve the user's mastery in software programming. The method was structured based on Saffer (2009) proposed methodology, with problem framing as Introduction, competitive analysis and differentiators as Related work, design research and structured findings as Preparatory study, prototyping in the dedicated chapter, and testing as User study and future work. My own experience being that of a C/C++ programmer with a passion for performance, I focused on C++, as this general-purpose language is widely used in industry nowadays.

1.1. Problem definition

The compiler should extend interaction *after* the creation of a working executable, to suggest opportunities of improvement, for example. Software such as Intel VTune Amplifier, SmartBear AQtime Pro, or Microsoft Visual Studio Analyzer, can typically provide such functionality. However, they are rather a collection of tools, and require dedicated learning, yet the knowledge gap is not intended to be *dug*. The suggestions should not require the user to learn a convoluted interface, hence the idea of mere feedback messages.

Such notifications are not limited to suggestions; fundamentally they provide feedback on the compiler's operation while parsing the source code, on how it was "understood". They could assure that a hand-coded optimisation is unnecessary when it is transparently done by the compiler. Furthermore, they would allow the removal of conditional compilation features such as preprocessing and generics from programming languages. Indeed, nowadays compilers are capable of evaluating certain run-time expressions at compile-time. With lack of information, however, users still resort to the dedicated syntax to enforce early evaluation. Now, if a user is aware of *which conditions* trigger the former behaviour, the language can be trimmed from its compile-time semantics.

Beyond feedback, with a little more work the compiler could be able to directly query the programmer, to suggest local tunings when the language semantics show their limits. Think about the problem of specifying the underlying search tree structure of a `set` object. Since the C++ standard library does not offer this choice, users must cope with the default implementation shipped with their compiler. If the latter provides several implementations though, it could probably be specified through *pragmas* (the compiler-specific preprocessing instruction in C/C++) but again this would require learning the particular syntax. An alternative here would be to generate a query with radio buttons for the programmer, and automatically insert the correct corresponding `#pragma` call.

1.2. Challenges

Many potential issues were identified ahead of this work, with the help of the reactions to simulated intelligent help highlighted in Carroll (1988). The biggest difficulty would be to properly relate the messages to their context, that is to output messages which actually *interest* the user, so as not to be disabled like a *Message of the Day* in IDEs. Technically, suggesting improvements, guessing the critical parts of a program, querying the programmer and binding a low-level transformation to the source code are challenging tasks. In order to avoid calling for unmanageable artificial intelligence, the system will aim for a simple implementation. Finally, for a reasonable project the amount of notifications are expected to become tremendous, hence the need to filter them, as discussed further. With real-world compilers in mind, the work was conceived to be a realisable project amidst the acknowledged difficulties. See the differentiators and future work for discussion on how they are tackled.

[3] Refer to the C++ specification, for example (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372, accessed 03.09.2012).

2. Related work and differentiators

I started by analysing three previous similar attempts (Table 1), then identified a few differentiators so as to make this thesis a conceivable choice for real-world development environments.

	VISTA (Zhao et al., 2002)	EAVE (Bose, 1988)	Matlab Code Analyzer
Learning the interface	Moderate: GUI separate from the editor	Lengthy: text interface separate from the editor	Easy: advisory messages under the relevant lines
Preliminary knowledge	Transformations (cited by name) and RTL	Transformations and capabilities of the machine	None required, everything is explained, technically
Amount of information	Huge: showing step-by-step optimisations on RTL	Decent since limited to loops	Decent, but limited to critical advices
Limits	No binding to original source	Only vectorisation, need to *request* advice	No simple feedback or querying for tuning

Table 1 – Comparison of three related works

2.1. The compiler queries the programmer

A mere advice solicits the user where code needs updating; here we introduce the possibility to simply inform where code was properly understood, as well as enable direct querying of the programmer. More specifically, interaction with the programmer can be divided into four distinct tasks (Figure 1). The compiler should be able to:

- *inform*: tells how well a portion of code was compiled, relates a compilation technique, promotes a coding practice, introduces a feature from the standard, etc.
- *alert*: incites the user to correct a supposed flaw, notifies about a potential vulnerability which could arise with further lack of attention, recommends a performance tweak. This is the task at hand in Code Analyzer. As opposed to the previous task, here we request some code to be fixed. Also, the difference with compiler warnings is that the latter concern code which might not execute with the intended meaning, though here only tuning is involved.
- *ask*: inquires a clarification about a portion of code. Sometimes an alert is insufficient, when the clarification cannot be expressed by updating the code. In cases like choosing the implementation of a tree structure (`set` or `map` in C++) or the character set (Latin-9, UTF-8, etc.) of a `string` object, the interface could trigger radio buttons to query the programmer. This task would further benefit from languages being designed with a supplemental *detailed* semantic, accessible with *pragmas* or menus in the IDE to avoid burdening the main syntax.
- *answer*: responds to a direct interrogation from the user. The requirement for artificial intelligence is not discussed in this paper, though an attempt to test its design was made further in the second prototype.

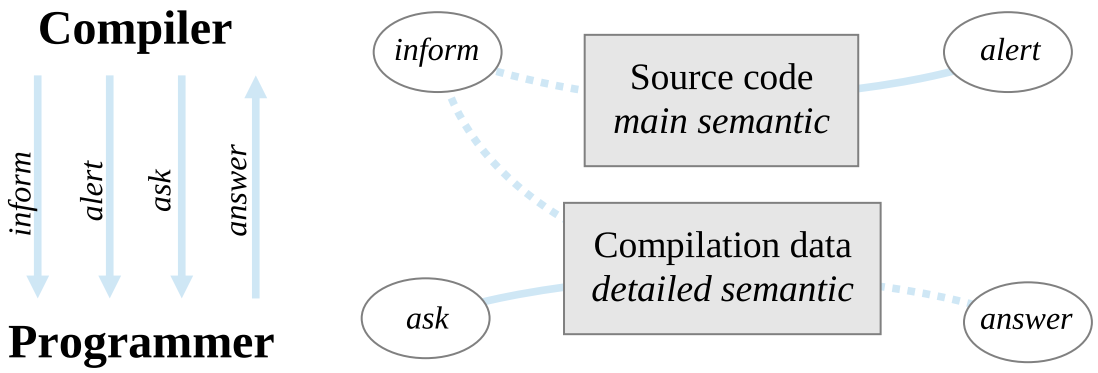


Figure 1 – On the left image, arrows indicate who initiates the communication. On the right one, lines bind each task to which data is discussed (a dotted line indicating the data is not to be modified).

2.2. No separate interface

VISTA, EAVE, and to a greater extent profiling tools in general provide the performance analysis and suggestions of improvement as an interface distinct from the editor. This requires users to learn how to use it, and this knowledge acts as a disincentive on their commitment to start profiling. Here we will bind the messages to the relevant source line, much as in Code Analyzer.

No interface has to be learned here, the only limit to the user's will to tune the program is the messages' clarity, which is discussed further in this paper. Moreover, such an interface can easily be implemented: in practice a compiler like GCC already outputs a line number along with every warning and error, and an IDE such as Eclipse is able to display warnings and errors in relation to the targeted line. This also helps the context to be accurately identified, as opposed to one of the challenges exposed earlier.

2.3. Cooperation instead of assistance

VISTA, EAVE and to a lesser extent Code Analyzer illustrate what an *assisting* agent is: it waits for the user to request help, it does not query him/her, and it focuses on helping the user fix an issue rather than improving his/her knowledge. By contrast, a *cooperation* is similar to a discussion, in which both interlocutors can engage the conversation, ask questions and answer them. Moreover, there must be no assumption that the programmer is familiar with any of the concepts involved. Contrary to VISTA and EAVE which refer to the optimization techniques by their names, here the tasks will give a short explanation and always cite their source, so that the user is never responsible for owning the proper reference.

This is actually not meant as a human-machine cooperation, since no artificial intelligence is intended. Instead, it is the designer behind the interaction tasks who is cooperating with the user. Neutrality needs not be sought then, as the feedback messages and the importance heuristic will embody the designer's point of view on how to improve a program. *Users Need Rationales*, as Carroll and Aaronson (1988) state, and a liberty for arguing is a decent mean to satisfy it.

2.4. The filtered notifications

In a simple technical design such as the first prototype shown further, or in Code Analyzer, a single pass on the code generates notifications to be displayed in the development environment – for simplicity, let us call messages the questions from the third task *ask* too. In EAVE and Code Analyzer, the suggestions generated all have critical importance, however we want to consider *every* possible feedback here. To avoid burdening the programmer with countless notifications, only a handful should be selected to be displayed at each build, hence the need to filter them. A simple solution proposed here is to evaluate a coefficient of importance for each message generated, then display the few highest rated ones. See the User study and future work for a technical description of the formula.

Besides, for the sake of transparency and to allow users to retrieve missed feedback, the full list of published messages must be kept available, that is all the notifications generated before they were filtered to keep a handful. The integration of this list in code editors is not covered in this paper, though.

3. Preparatory study

At this point, a series of interviews was necessary in order to evaluate the users' preferences regarding the contents of the messages, and ensure there would be no clear rejection of "improvements" to compilers. I chose to start with a simple algorithmic task. Being in familiar working conditions the interviewees could share their interrogations through a *think aloud*, and explain afterwards how they could optimise their code further. In need for participants with experience in programming, I selected KTH peers whom I knew had such experience. Each interview would be conducted on a platform the participant would be familiar with, be it his/her own laptop or a school desktop computer. I would sit next to the interviewee and give the instructions while he/she had the IDE in sight.

Three problems were written, covering a broad range of expertise, on three *distinct* typical goals in programming: *performance*, *security* and *maintainability/extensibility*. The problems would be given in any order, usually two in an interview, so as to fit in 40 minutes. Five interviews were carried over a month. This rather limited number of participants was fortunately mitigated by the range of their core fields: Numerical Analysis, Networking, Software Engineering, Cryptography, and Robotics.

It was first observed that *all* participants had interrogations or misconceptions regarding optimisations the compiler can perform, which was pounded by their average experience of 9 years in programming, and 2.5 years in their current language. This incomplete knowledge covered for example: where objects are stored in memory, the unrolling of loops, or the replacement of a multiplication by 2^n with a binary shift. One could argue that asking a quick working draft then its optimization induced the production of sub-efficient code in all three problems. This is however meant as a reflection of the IT industry, where delivery of working software under tight schedule is a core target[4]. The interviews showed that people perform hardly well at optimizing code, the interface should thus provide help to produce *efficient working code* at first draft.

It also appeared that proper optimization is not barely a matter of time. One *has to be* an expert in the specific field corresponding to the specific aspect targeted. This leads to projects centred around one aspect, the others becoming sub-efficient. A perfect example is the BSD family of operating systems: FreeBSD (performance), OpenBSD (security), NetBSD (portability)[5]. The interface should thus help to compensate where users lack expertise.

Furthermore, the participants often showed interest in the answers to questions they had previously been wondering. The existence of such unanswered interrogations which somehow *haunt* the users, calls for serendipitous information retrieval (De Bruijn & Spence, 2008). The interface should output several different messages at each execution and cite a source in each one, so as to expose the user to much information, that potentially answers a dormant interrogation. Moreover, for the same purpose the messages should be the shortest, and the number of sources limited to one.

Finally, when observing those using an IDE, I noticed they were all disabling the *Message of the Day* tooltip at startup. The reasons they gave were that much of the information displayed would document basic functionalities, the first few messages were not teaching anything, and they were neither contextual nor relevant. The first note motivated the requirement for technicality of the messages, that is they should always seem relevant, not worth being disabled, even if they would be quite complex. A source link could then provide the necessary details to users willing to follow it. As for the third note, it instructed to avoid citing what the compiler *can do* in general, in favour of informing what it *will do* on a particular line of code. Thereby, the feedback is closest to the context which triggered it.

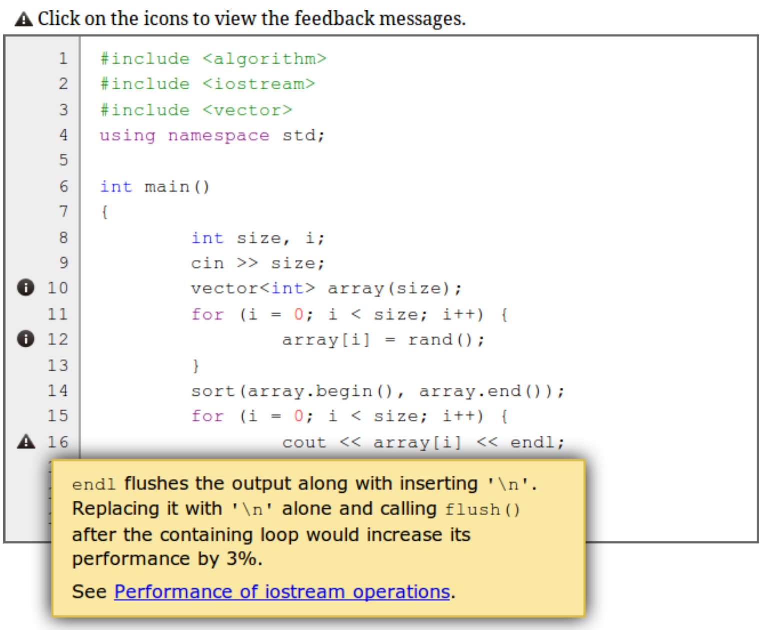
4. Three prototypes

As advocated in Dow et al. (2010), I chose to design several prototypes in parallel, each embodying a distinct approach to the solution. This work does not include their iterations though, which are simply discussed in the next chapter.

4.1. First prototype: a stripped version for GCC

This prototype (Figure 2) emerged after an unsuccessful proposal for a Google Summer of Code for GCC[6], and corresponds to the interaction tasks *inform* and *alert*. Confronting the design of a compiler's feedback to the internal workings of GCC helped identify the difficulties, and overcome them with a simple technical scheme. HTML5, CSS3 and JavaScript were used for the neat look and interactivity they provide. Note that the sample feedback messages presented in all three prototypes are not meant to be true for any particular compiler; they simply look technical and precise.

[4] Refer to the first, third and seventh principles of the Agile development method at http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/ (accessed 07.09.2012).
[5] For a brief history and comparison of the three major BSD systems, see http://www.freebsdworld.gr/freebsd/bsd-family-tree.html (accessed 07.09.2012).
[6] Available at http://www.google-melange.com/gsoc/proposal/review/google/gsoc2012/traf/2001 (accessed 13.09.2012).



⚠ Click on the icons to view the feedback messages.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
        int size, i;
        cin >> size;
        vector<int> array(size);
        for (i = 0; i < size; i++) {
                array[i] = rand();
        }
        sort(array.begin(), array.end());
        for (i = 0; i < size; i++) {
                cout << array[i] << endl;
```

`endl` flushes the output along with inserting `'\n'`. Replacing it with `'\n'` alone and calling `flush()` after the containing loop would increase its performance by 3%.

See Performance of iostream operations.

Figure 2 – Screen from the first prototype[7]

The original proposal involved generating the messages from every compilation unit in GCC, entangling the messages in its source code. The proposed approach here is to use files to store all possible messages, as pairs {*text*, *trigger*}, the latter being a condition on each instruction processed which enables the output of the corresponding *text* as a feedback. Though outside the scope of this thesis, the standardisation of a *trigger* syntax would allow messages to be written for several compilers, and be shared among users thanks to their storage in files.

The most important task along with designing the interface was to provide an exhaustive list of possible messages, to give a clearer idea of its usefulness. In order to manage the enumeration, I focused on the standard optimizations performed among most modern software:

- Register Allocation (explaining how faster an operation is performed in registers, telling whether for an inner loop or a function all automatic variables could be stored in registers or spilling happened, citing the allocation technique used, informing which conditions allow a data structure to be stored in registers)
- Strength Reduction (indicating when a multiplication by a loop index was carried with an addition, warning about the use of floating point functions on integers and propose alternatives, showing the replacement of multiplications with powers of 2 by binary shifts)
- Common Sub-expression Elimination (informing where an expression was found to be redundant and how the code was replaced, enumerating which operations are taken into account in CSE)
- Value Range Propagation (telling when a constant has been properly propagated, showing that the detected range of values of a variable leads to a performance gain, enumerating the types which can be propagated by the compiler, citing the Static Single Assignment technique)
- Branch Prediction (informing when a dead section was detected and will not be compiled, showing how branch probabilities translate into code and how performance improves)
- Functions Optimizations (telling when and why a particular function was inlined, informing about the compilation flags toggling inlining, warning when too many variables are passed to a function as the registers are limited, telling when Tail Recursion could be applied on a recursive function, describing how complex objects like classes are passed as arguments and returned)

[7] The first prototype is available online at http://www.csc.kth.se/~traf/thesis/proto1.html.

- Data Alignment (explaining why the size of a structure can be bigger than the sum of its fields' sizes, telling in which case padding was added inside a data structure, informing about the performance penalty when accessing unaligned data)
- Stack Layout (pointing which variables are stored on the stack, giving figures as to how performance increases with such storage, proposing buffer overflow protection techniques and informing which flags enable them, giving the typical stack size on the target system)
- Vectorisation / Parallelisation (indicating whether and why a loop could be vectorised on a SIMD-capable architecture, describing how to best control vectorisation through flags and tools, providing figures as to how the performance of a loop increased with the use of SIMD instructions, telling whether several similar operations could be packed in a single instruction, suggesting parallelisation libraries to execute simultaneous iterations on parallel threads)

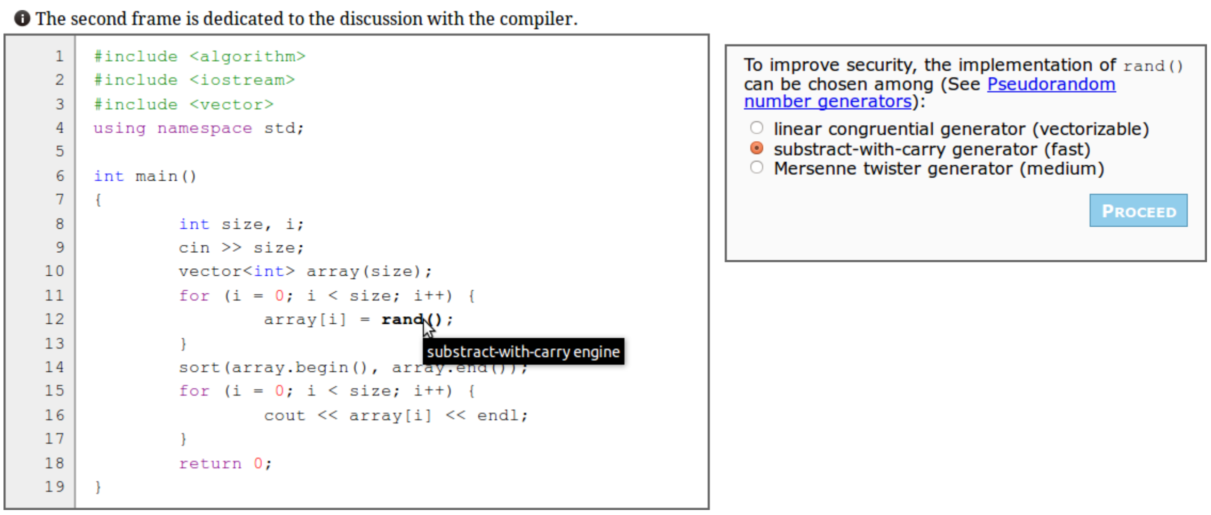
I also focused on the various aspects of a language design (mainly C++) to enumerate a few more topics for feedback messages:

- Manipulation of files (explaining the difference in performance/cache use/security between the various input/output functions, pointing out risks for unsafe/unchecked input)
- Time management (warning about the year 2038 bug and discussing means to circumvent it)
- Strings and characters (providing a comparison between null-terminated and sized strings, explaining how the fast comparison and copying functions translate into code)
- Assertions (giving figures as to how enabling assertions impacts performance, telling whether assertions are used for value range analysis)
- Style and formatting (warning when a function is too big that it would not fit in a cache, telling which character set was detected for the source file and how strings are stored in the output, here the messages can greatly depend on the designer/community)
- Run-time checks (enumerating the list of available checks and the flags enabling them, informing when such checks have been inserted and their cost)
- Classes (showing when constructors and destructors are inlined, giving the number of system calls involved in the use of dynamically sized objects, describing the actual implementation of standard complex classes like bit-fields or hash tables, telling which operations will leave a certain iterator stable)
- Functions (introducing the overhead of a function call, telling which registers are saved/used)
- Data storage (explaining where in memory the standard instructs a particular static/automatic variable to be stored, informing where in memory constants are saved, introducing endianness and why one should care about it, suggesting faster initialization methods like `memset`)
- Floating point types (warning about the use of equality with such variables, describing the range of acceptable values including subnormal numbers and the expectable precisions)
- Operators (telling how certain ambiguous operations like integer division behave with negative operands, comparing the speed of an addition versus a multiplication on the target architecture, warning when an apparently small operation like a norm has a non-negligible cost, informing about the possibility of integer overflow and proposing various means to avoid it)
- Control flow structures (explaining how switch statements are converted into fast code)
- Exception handling (describing how this mechanism is translated into code, citing which types of exceptions are the most easily dealt with by the compiler)

These lists are certainly not exhaustive but already give a strong basis of feedback messages the interface could implement.

4.2. Second prototype: a communicative compiler

The second prototype (Figure 3) was designed to complement the feedback side with the possibility to query and discuss with the programmer, corresponding to the interaction tasks *ask* and *answer*. After a successful compilation, the second frame displays a set of spontaneously generated queries. As with the first prototype, these notifications will change each time a compilation is run. Answering them is never required; they will default to safe values.



The second frame is dedicated to the discussion with the compiler.

```
 1  #include <algorithm>
 2  #include <iostream>
 3  #include <vector>
 4  using namespace std;
 5
 6  int main()
 7  {
 8          int size, i;
 9          cin >> size;
10          vector<int> array(size);
11          for (i = 0; i < size; i++) {
12                  array[i] = rand();
13          }
14          sort(array.begin(), array.end());
15          for (i = 0; i < size; i++) {
16                  cout << array[i] << endl;
17          }
18          return 0;
19  }
```

To improve security, the implementation of `rand()` can be chosen among (See Pseudorandom number generators):

- linear congruential generator (vectorizable)
- [✓ substract-with-carry generator (fast)]
- Mersenne twister generator (medium)

Figure 3 – Screen from the second prototype[8]

Technically, this prototype requires a compiler-specific semantic to express the answers to queries. As an example, clicking “Proceed” on the question in Figure 3 would add `#pragma rand substract_with_carry` before line 12, effectively giving this hint for the next build run.

Triggering these messages would then be similar to the first prototype. They would become triplets {*text*, *trigger*, *pragma*}, where *text* would be formatted to generate a query, and *pragma* would contain the text added to the source code after answering the question.

Enumerating the messages to include in such an interface is not as straightforward as for the first prototype. It requires seeking the aspects of a language semantics which are incomplete. For C++, I focused on the aspects which would benefit from the increased expressiveness without burdening the main semantic:

- Choosing the character encoding of a string or stream, which will influence functions such as `strlen` or `isspace`
- Setting the locale of the program, since the current C standard dedicates a single library to it
- Asserting that a certain variable will never overflow, which could enable certain optimizations
- Choosing the algorithm behind certain mathematical operations such as computing the inverse square root, while giving the precision of each
- Querying the expectable branching probabilities in a critical portion of code
- Asking whether to maintain an assertion in release mode when sample cases show its failure
- Gathering the properties of an variable read from a stream, to enable the use of faster routines
- Selecting the algorithm to sort an array, the default being usually Quick Sort
- Setting the precision for the storage of time or delays
- Selecting the implementation method to generate random numbers (in C++ a library is dedicated to it, though in C it is a single function)

[8] The second prototype is available online at http://www.csc.kth.se/~traf/thesis/proto2.html.

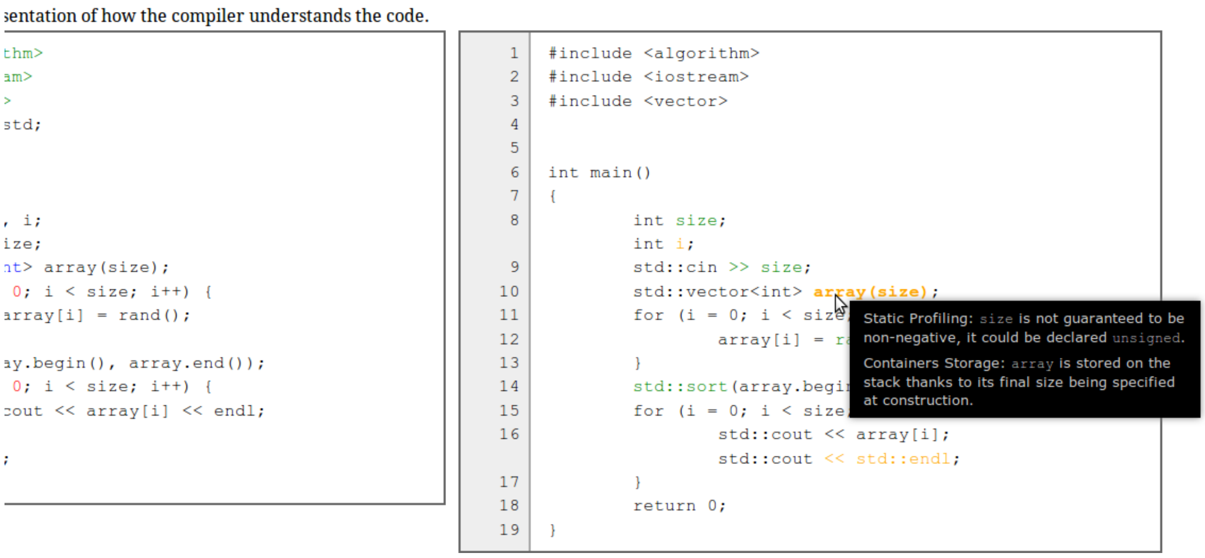
- Choosing the underlying storage of an array of bits (in C++ it is stored as a `bitset` though it sacrifices performance in comparison with an array of integers)
- Setting the properties of a container (the implemented directions of iteration, whether the size often increases, where items are appended, whether stable iterators are required)
- Asking for the implementation method of a binary tree or a hash table
- Proposing the stack protection method against memory corruption including buffer overflows

These semantics being optional, they must not have critical importance on the program. They will rather be set to tune its various aspects, when the program was already proven to work properly. As with the previous prototype, this list is certainly not exhaustive, but gives an insight for the usefulness of the *querying* improvement.

4.3. Third prototype: a far-fetched alternative

This prototype (Figure 4) goes one step further in the coupling between the compiler and its interface. It uses a second frame to graphically represent the compiler's understanding of the various elements found in the code. It was mostly intended as a place for open suggestions from the testers, and is not to be matched with the interaction tasks previously mentioned.

Having noticed in the interviews that users had a will to do good despite their refusal of an embodied interface, I chose to depict the compiler as a living system. The interaction then consists in the user helping the compiler understand the code, a simple colour scheme being used to inform how an element is apprehended. The two frames are representing the code as text, however the right one *should* evolve towards a more suitable representation, such as a coloured dataflow diagram.


sentation of how the compiler understands the code.

```
thm>
am>
>
std;



, i;
ize;
nt> array(size);
 0; i < size; i++) {
array[i] = rand();

ay.begin(), array.end());
 0; i < size; i++) {
cout << array[i] << endl;

;
```

```
#include <algorithm>
#include <iostream>
#include <vector>


int main()
{
        int size;
        int i;
        std::cin >> size;
        std::vector<int> array(size);
        for (i = 0; i < size
                array[i] = ra
        }
        std::sort(array.begi
        for (i = 0; i < size
                std::cout << array[i];
                std::cout << std::endl;
        }
        return 0;
}
```

Static Profiling: `size` is not guaranteed to be non-negative, it could be declared `unsigned`.

Containers Storage: `array` is stored on the stack thanks to its final size being specified at construction.

Figure 4 – Screen from the third prototype[9]

5. User study and future work

A new series of interviews was scheduled to ensure the goals set after the first one had been met, and estimate how users would consider the value added. Five interviews of 30 minutes each were conducted over two weeks, with the requirement that the participants had not participated in the previous interviews, had had a previous experience with an IDE, and could understand basic C++. After testing the three prototypes in order, the interviewee would choose his/her favourite and explain it, then answer a few additional questions about the use of *personae* (see the next dedicated section).

[9] The third prototype is available online at http://www.csc.kth.se/~traf/thesis/proto3.html.

Both the first and third prototype were praised, the former for its technical insights, and the latter as a quick overview of the compiler's job. As with the preparatory study, few participants were initially showing interest or added value in a feedback from the compiler. I had to pursue the description to the personae, until the concept of compiler feedback became clear and coherent. Then they all agreed that they would not disable the feedback like a *Message of the Day*, which was my main concern. Some even expressed they were actually looking forward to seeing a working release in the future. Surprisingly though, very few interviewees understood the third prototype's colour scheme at first glimpse; further iterations should probably explore accentuating the areas requiring attention inside the very tooltips.

On the downside, the second prototype was generally little understood. This might have been influenced by the unusual situation of being queried by the compiler. However, in my opinion it was the presentation as a separate frame which made it difficult to spot the context at hand, that is which part of the program the question was dealing with. The integration of queries along source lines as in the first prototype then remains to be tested for future iterations.

5.1. The personae

This idea appeared with the possibility to store messages in files, as discussed in the first and second prototypes. Provided a *trigger* syntax is defined, each notification can be stored aside the compiler, under a triplet {*text*, *trigger*, [*pragma*]}. Sets of notifications can then be stored as files forming categories of similarly related items. The addition of a field along every message of the first prototype could further allow the programmer to be aware of the category at hand, and increase or decrease its further occurrences, in order to receive the most interesting feedbacks.

Categories form the default set of messages shipped with the compiler. To extend and customise this set, users could create their own files and share them. The *persona* here is the idea to bind an author to a file with notifications. Knowing who wrote a certain suggestion could give value to it and mitigate the effect of a poor feedback, particularly if the author is known for being a good programmer. Here, a simple and recommended way to store the category and author's names is though the file's name.

In practice most testers were very receptive to it, with different intents. One tester did not care about the author's name, as long as he/she was a specialist. Another one conceived the sharing of files inside teams of developers, in companies. A last one considered contributing in online communities of developers rather than friends.

5.2. A few rules for composing the messages

During the tests, feedback was sought for the relevancy and clarity of the messages. While the testers were often puzzled with the feedback's technicality, they were very fine with it. Indeed, two actually argued that they were used to this situation. The links to references were intended to balance this complexity, and in practice were praised by all interviewees. The quality of redaction had a great influence on the participants' reception of each message, though. The first query in the second prototype, for example, was systematically deemed too complex, and I always had to explain it. This difficulty motivated further the addition of personae, to let programmers choose a good teacher, and sketch a set of rules to help the redaction of further messages:

- *Technicality*: The feedback should rather be too technical than not enough, and provide a substantial benefit which will be highlighted.
- *Context*: Indicate what the compiler *will do* for a particular line/object rather than what it *can do* in general.
- *Referencing*: Cite one and only one source giving details for the corresponding feedback.
- *Neutrality*: Balance the amount of positive and negative feedback, that is when a line was well understood or when it needs tuning.
- *Clarity*: To be read and understood quickly, each message should receive careful attention and go straight to the point.

5.3. Transparency is crucial

As advocated in Sinha & Swearingen (2002), transparency has been a key design choice along this work. Providing a reference link along each feedback, targeting the programmer's knowledge rather than hacking tips, binding the author to the messages, defining a syntax for notifications and storing them in text files directly accessible to the programmer, were all choices motivated with transparency in mind. It should be noted here that transparency is preferred over translucency – selecting what is to be shown – since no *thought* restriction on the information given was intended.

In my opinion, transparency is the key means to show and insist on the absence of artificial intelligence, or "smart assistant", to govern the suggestions. Giving the qualifier *smart* to the robot could make the users feel it is asserted to be smarter than them. Carroll and Aaronson (1988) bring out many receptiveness issues from interacting with such an agent, which transparency would greatly mitigate. Indeed, with access to the database of possible messages, and knowledge of who lies behind them, users are aware of the bounds of the compiler's intelligence, and will not expect more than it could actually help.

5.4. A formula to evaluate the importance of each message

As hinted in the differentiators, taking into account the huge range of possible feedback messages will require the computation of an importance factor along with every message generated, in order to select a handful to display. For the purpose of being exposed to the programmer and implemented easily, the formula constructed here aims at simplicity.

Let us note f_c a constant criticality factor for the message, n the number of its previous occurrences, f_p a preference factor assigned to its category, and m the number of previous messages displayed on the same line. A simple recommended formula here would be the average of f_c, 2^{-n}, f_p, and 2^{-m}, provided all are bounded by [0;1]. The strength of such a formula is the simplicity to graphically represent an average. As a drawback, it does not allow to completely disable one category, though this is actually possible by simply deleting the corresponding file. Also, the factors averaged might need to receive an additional scale, which estimation is left to implementation.

Note that reinitialisation of the heuristic is to be taken into account. The compiler might be reinstalled often, possibly clearing the memory of previously displayed messages (n and m) and preferences (f_p). The more files/categories are stored, the more time will be needed to reach their previous f_p values (considering the programmer can only increase/decrease this factor on every message received). The number of categories should thus be limited to a dozen on average.

6. Discussion and concluding words

This project started with the idea to have the compiler return performance-helper messages, much as in the first prototype. The expected design was then pretty clear, though the content of the messages remained to be defined. However, most of the time was actually spent on communication tasks. Indeed, the project suffered from the difficulty to clearly state what was intended by a *feedback*, because this word can be interpreted quite at will. Moreover, it evolved vastly to seek a coherent system which could be obviously differentiated from mere compiler warnings, and the dreaded paperclip assistant from Microsoft. Designing the prototypes with the precision of HTML and CSS helped dramatically to show *exactly* what was intended. An actual challenge during this Degree Thesis was the management of the two rounds of interviews. Though in this paper they occupy a marginal space, in practice a sheer amount of time was dedicated to them, especially for preparing the interview plans and the algorithmic tasks.

A few points were left aside during this work, either by lack of time or because they were a matter of debate. Among them, the initialisation of the system should be mentioned. Introducing the feedback is important, so that users do not disable it instinctively like a *Message of the day*. An example for an introductory text could be: *This compiler can output feedback messages telling how it understands the code, as well as technical suggestions. The list of feedback messages it can generate is contained in [folder], and can be extended by adding .cfb files downloaded from trusted authors*.

Also, this paper does not cover how to identify the level of knowledge of the users. Receiving too complex/simple feedback might eventually annoy them. While they can choose and download the notifications' files to add to the compiler, the initial set of categories could be specifically tailored to each one's knowledge, by estimating it with a question along the introductory text, for example.

One last issue which might eventually arise with the possibility to share authored feedback files is the lack of secure signing. If the author's name is stored in the name of the file as suggested in this paper, nothing prevents it to be overwritten, or a wrong set of messages be imputed to the same author. The rationale behind this choice is similar to the availability of coding guidelines on the Web: it is the user's responsibility to fetch the file at the source he/she trusts.

As shown in the tests, the introduction of a feedback from the compiler was well received, either indirectly for the "big picture" overview it would provide, or for its relevant technical insights. Using this interface does require very little learning, if any, which is in my opinion a cornerstone of this work. As for the direction to give to the prospective future iterations of the prototypes, since both the first and third were deemed promising the focus should be laid on implementing the common basis, namely generating the feedback messages. An option could then be available to choose *how* to display them. Indeed, this choice might depend on the progress of the coding project. At the early stages when raw code is written, a few technical messages targeting the most critical aspects would be needed, as in the first prototype. Later in the process when these fragments are assembled, a broader overview like the third prototype would become useful. Giving these tools to the programmer would then make optimisation more accessible, outside the sphere of expert programmers.

7. References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Prentice Hall.

Bose, P. (1988). Interactive program improvement via EAVE: an expert adviser for vectorization. *Proceedings of the 2nd international conference on Supercomputing - ICS '88* (pp. 119–130). New York, New York, USA: ACM Press. doi:10.1145/55364.55376

Carroll, J., & Aaronson, A. (1988). Learning by doing with simulated intelligent help. *Communications of the ACM*, *31*(9), 1064–1079. doi:10.1145/48529.48531

De Bruijn, O., & Spence, R. (2008). A new framework for theory-based interaction design applied to serendipitous information retrieval. *Transactions on Computer-Human Interaction*, *15*(5). doi:10.1145/1352782.1352787.

Dow, S. P., Glassco, A., Kass, J., Schwarz, M., Schwartz, D. L., & Klemmer, S. R. (2010). Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction*, *17*(4), 1–24. doi:10.1145/1879831.1879836

Kreeger, M. N. (2009). Security testing. *ACM SIGCSE Bulletin*, *41*(2), 99. doi:10.1145/1595453.1595484

Lethbridge, T. C. (2000). What knowledge is important to a software professional? *Computer*, *33*(5), 44–50. doi:10.1109/2.841783

MathWorks. (n.d.). Using the MATLAB Code Analyzer Report. Retrieved September 7, 2012, from http://www.mathworks.se/help/techdoc/matlab_env/f9-11863.html

Saffer, D. (2009). *Designing for Interaction: Creating Innovative Applications and Devices* (2nd ed.). New Riders Press.

Sinha, R., & Swearingen, K. (2002). The role of transparency in recommender systems. *CHI '02 extended abstracts on Human factors in computer systems - CHI '02*, 830. doi:10.1145/506558.506619

Zhao, W., Jones, D. L., Cai, B., Whalley, D., Bailey, M. W., van Engelen, R., Yuan, X., et al. (2002). VISTA. *ACM SIGPLAN Notices*, *37*(7), 155. doi:10.1145/566225.513857

Evaluating application programming interfaces as communication artefacts

Luiz Marques Afonso[1], Renato F. de G. Cerqueira[1,2], and Clarisse Sieckenius de Souza[1]

[1] Departamento de Informática, PUC-Rio
{lafonso,rcerq,clarisse}@inf.puc-rio.br
[2] IBM Research Brazil
rcerq@br.ibm.com

Abstract. Application programming interfaces (APIs) allow the reuse of software artefacts by providing abstractions to other software layers, and their design is critical to enable the effective use of the underlying software and avoid programming errors. As such, the role of an API designer should be strengthened in any software project that has reuse among its goals. Also, we should be able to evaluate the effectiveness of an API in communicating its design to programmers and identify the tools and techniques that help the designers to accomplish this task, so that APIs may be easier to understand and use. This paper describes a work in progress that proposes the use of a combined semiotic and cognitive method to evaluate APIs as an artefact mediating the communication process between designers and programmers, and also aims to investigate some possibilities of enhancing this communication.
Keywords: POP-I.B Barriers to programming; POP-II.B Program comprehension; POP-III.C Cognitive dimensions; POP-V.B Research methodology

1 Introduction

Abstraction is one of the central concepts in Computer Science [30], permeating all the activities related to software construction and use. It is a cognitive resource that allows us to remove details in order to simplify things and focus attention on the core properties of a complex object [24]. As such, it seems intuitive that, although executed by a machine, a software artefact has its creation process deeply based on human interpretation.

Reuse is one of the main goals of Software Engineering. To be achieved, software reuse relies on abstractions in the form of libraries, components, objects, and other artefacts. At each level, software interfaces allow a programmer to construct new abstractions to be provided to another layer, exposing new concepts and design, and hiding internal details as needed.

Reusable software components are specified and implemented by programmers to be used by other programmers in order to construct new software. The reuse of a software artefact is achieved via its interfaces, which allow new software to call the available operations and create new functionalities on top of the existing abstractions. Programmers need to realise the concepts and the design behind the interfaces available in order to use them effectively. From a human-centric perspective, we can consider that a communication process takes place between programmers, mediated by the software artefacts involved.

We refer to software interfaces, or APIs (application programming interfaces), as any set of semantically related operations and data, usually associated with a specific domain. Software components, modules, libraries and frameworks usually provide APIs that expose their functionality to other software elements. This definition is similar to the one used in the work by de Souza et al. regarding the study of APIs in the context of cooperative work [13]. APIs play a central role in modern development environments and languages, since even the most simple programs depend on the provided library and framework interfaces [23].

Software projects are known to be a difficult endeavour, and defects are usually expected. Although it is not easy to estimate the percentage of defects related to the incorrect use of APIs or to misinterpretation of its design, probably every seasoned programmer has already experienced difficulties and errors when writing code involving a complex API.

The use of software interfaces may impose a considerable amount of cognitive load on the programmer, depending on the abstractions involved and the design of the artefacts provided. The higher this load is, the higher is the intellectual effort needed, which may increase the error-proneness of the activity of software development.

To illustrate how API design and communication of intent may have an impact on the work of developers, we present a simple example from the Java API, based on the book from Bloch and Gafter [5]. The example shows that even well known concepts like date and time, used in almost all application domains, can be a source of problems. The Java class *Calendar* implements some of the functionalities regarding date and time in the language API, and it contains many *set* methods that allow the programmer to change an object's internal fields. One variant of these *set* methods is defined as below:

```
void set(int year, int month, int date)
```

At first, the method's short description in the documentation looks pretty obvious: *"Sets the values for the calendar fields YEAR, MONTH, and DAY_OF_MONTH"*. Although it may seem straightforward, a very simple program reveals what is behind the implementation:

```
Calendar c = Calendar.getInstance();
c.set(2012, 8, 31);
System.out.println( c.getTime() );
```

Surprisingly, the output of a Java program containing this code snippet is similar to the following:

```
Mon Oct 01 13:21:27 BRT 2012
```

Most people would expect something like "Aug 31". The explanation comes from the fact that the *Calendar* class handles months in a zero-based integer representation (e.g. 0=Jan). So, the parameters in the example mean "Sep 31", which is itself an invalid date. In this case, the *Calendar* class silently "corrects" the date overflow to the next day, which is "Oct 1".

Despite the fact that this simple example may be considered as a bad design decision, it illustrates how the designer intentions behind an API may differ from the first user interpretation of its meaning. And if this may happen with a well known concept like date and time, more complex domains can be a real challenge to the designer in order to represent the concepts, meanings and behaviour behind an API, and also to the programmer, who has to interpret the designer's message and use the software artefact as originally intended.

This paper describes a work in progress in which we aim to analyse software interface specifications from a human-centric perspective, trying to identify how its effectiveness can be evaluated and what can be done to enhance the communication of a software artefact design to programmers. We intend to use a combined semiotic and cognitive inspection method in this investigation.

The remainder of this paper is organised as follows: section 2 discusses some aspects related to the view of API design from a communication perspective that motivate this work. Section 3 analyses methods for evaluating APIs from a semiotic and cognitive perspective. Section 4 arguments about the use of some techniques that may influence the communication of design intent in the context of programming interfaces. Section 5 presents a sample scenario for the application of the referred methods and possible findings for the type of experiment proposed. Finally, we describe related work in section 6, and conclude with our final remarks and future work in section 7.

2 APIs: communicating design

In this section, we look at APIs as a design artefact and analyse the communication aspects involved in their representation of the designer's intent to programmers.

Programming is a hard mental work, and a developer usually has to deal with a great amount of information to write functional code. Problem domain, requirements, specifications, algorithms, programming language, tools and frameworks are among the kind of knowledge that is demanded from a programmer to perform his tasks. APIs are used most of the time when programming in modern languages, especially when dealing with distributed systems and enterprise frameworks.

API design is critical because it is intended to be written once and used many times, and later changes may impact users due to compatibility issues. Complex APIs may be daunting, and difficult usage may discourage adoption, as it increases the demand for experienced programmers that may be able to use it effectively and efficiently. An example of the consequences of a overly complex API can be found in Henning's work [19]. Another article by the same author discusses the importance of API design [18].

An interesting aspect related to the design of APIs is that they can have many different and specific goals, which generally make them unique. There can be subtleties behind it that make some decisions critical to its completeness, usability and flexibility. The design process should anticipate some crucial aspects about API usage, and this may influence the underlying system design. For example, the work by Ierusalimschy et al. [22] provides interesting insights about how the design of the API for embedding Lua scripts in C programs influenced the design of the Lua language, and vice-versa.

When developing software, a programmer should have a good mental model of the software artefacts being reused in order to correctly apply these models to his own design, and call the available operations and services accordingly. If there is not a good understanding about the abstractions being provided, this can lead to subtle errors that may appear later in the software life cycle. In a research work dedicated to identifying common software defect causes and characteristics [20], the authors report that "API misuse is the single most prevalent cause" of bug patterns detected. Although this work is specific to the Java language, it is a good illustration of how poor API design may have a strong impact on software quality.

Designing an API is about describing abstractions through type and interface specifications, and it is usually work assigned to development team members without specific expertise or training in this design task. Although this may provide good results depending on the team's talent, more attention should be paid to the role of an API designer, given the impact that this activity may have on the overall project outcome. At least, the most experienced members of a team should be involved, as they have probably seen more of badly designed APIs and may know better what should be avoided [18].

If we consider that an API represents abstractions created by its designers that need to be understood by programmers in order to be used effectively, this may be viewed as a communication process taking place between these two parties, mediated by the software artefacts involved (specifications, documentation, code, binaries, messages, etc.). More precisely, these artefacts communicate to the programmers how they should interact with the API, which is itself another piece of communication. So, in this sense, API design may be regarded as a metacommunication process taking place between designers and programmers.

In their work [27], Robillard and Deline describe qualitative findings regarding API learning obstacles and conclude that documentation of intent is one of the most important factors that impacted learner's experience. This stresses the importance of effectively communicating to programmers the API designer's intentions when developing its abstractions and concepts, and thus supports our argumentation. Also, they state that "the responsibility from documenting an API cannot be cleanly separated from the responsibility for designing the API, even though different skills are involved". This reinforces the need for a specific role of API designer in the development team.

The idea of studying API design as a communication process between designer and programmer builds on the discipline of Semiotic Engineering [11], which provides a semiotic theory for

HCI. Semiotic Engineering has been successfully developed and applied in the last two decades to study interactive computer systems as “one-shot messages sent from designers to users”, taking into account the meaning-related and signification processes that occur in the design and construction of software artefacts.

Although the concepts and methods in this discipline have been traditionally applied to the analysis of interactive systems in the context of graphical user interfaces, we aim to develop them further to evaluate the interaction of programmers with APIs, adapting to the differences of this type of system (a programming interface).

When dealing with interactive computer systems, the designer may convey his message to the user by means of choosing proper graphical elements in the interface, screen layout, dynamic system behaviour, and other representations, most of them visual. User interfaces have been the focus of much attention in the last decade with the availability of new technologies that allow different forms of interaction, especially touch and voice-based. This gives the designer a rich toolset to communicate his intent to the user during interaction time.

As to programming languages and APIs, the tools available to the designer usually do not offer as many possibilities to represent his intentions at interaction time, when compared to graphical user interfaces for computer systems. The most common resources for communicating API design are the syntactical representation of the interfaces, and the textual documentation explaining its concepts and behaviour. The syntactical structure of an API comprehends the names of the interfaces, operations and data structures, as well as their types. Good name choices are very important at this level, because they are almost the only representation of intent available to the designer. They should also be consistent throughout the various elements.

To overcome the limitations of syntactical specification, textual documentation is the most common medium to complement the representation of the designer’s intentions. Natural language documentation is commonly used, but sometimes it may be ambiguous or incomplete. Documentation may also be augmented with formal descriptions of the operations using a rigorous mathematical notation, in an attempt to reduce or eliminate these ambiguities. On the other hand, this type of notation can be more difficult to be understood by programmers with less formal background.

In order to give a more dynamic approach to textual documentation, some integrated development environments (IDEs) try to bring them closer to the code writing activity, suggesting operations, parameters and showing their description to the programmer. Although this can be of great help when writing code, it does not eliminate the need for a more complete description of the concepts and abstractions behind the API design, which are fundamental to its adequate use.

Code examples are another important technique to represent the intent behind the design of an API, as they present use cases from the perspective of the designer’s interpretation of the abstractions involved, which can be different from the user’s, at least while there is not a clear comprehension about the message being conveyed. Good examples can eliminate possible gaps between syntactical and textual description of the API, and its actual use, as they usually make the concepts and the intent behind the design more concrete. The downside of code examples is that they can be a source of “copy/paste programming” without a real understanding of the concepts and the dynamics of the API, but this type of practice is up to the programmer to be avoided.

Differently from graphical software, a programmer’s interaction with an API occurs when a program is written, compiled, executed and debugged. The programmer reads the documentation, writes code to perform a certain task, evaluates the return codes, handles exceptions, runs the program, analyses the outcome, reads messages, output traces and logs, and so on. This type of interaction inherently limits the designer options to be “present” at interaction time when compared to more visual and dynamic system interfaces. One of the research questions that arises when investigating this subject is how the use of different tools and techniques, other than syntactical specification and natural language documentation, can help the designer perform a more effective communication to the programmers of how interaction with the API is expected to be done.

As already mentioned, programming with different APIs demands a high cognitive load from the user perspective. So, the expected result of making the communication of API design more effective is to lower the hurdle for the programmer to accomplish his task. If we intend to analyse the representation of API design from the Semiotic Engineering perspective, it seems natural to combine this analysis with a view of the cognitive impact on the programmers tasks.

In this context, the Cognitive Dimensions of Notations framework (CDN) [2] has been successfully employed in previous works related to the evaluation of programming-related tasks [8, 25], and it can provide an interesting counterpart to the semiotic view of API design. In the next section, we describe this combination of semiotic and cognitive inspection methods in the context of API design evaluation.

3 Evaluation of APIs

This section refers to existing methods for the evaluation of APIs and proposes the use of a combined semiotic-cognitive inspection method that can be used in a technical context, to evaluate a particular API, and in a scientific context, to generate new knowledge in HCI and Software Engineering.

The discipline of API design and implementation has been extensively studied in the past decade, as it is a main concern for software development companies that publish APIs to a large client base. Some of the most representative work in this field come from large software companies like Microsoft or Google [8, 3]. Their concern originates from the fact that getting it right before publishing is mandatory, because post-release fixes are costly and may break legacy code.

Many recent studies in API design have been developed under a usability or learnability context ([27, 28, 9, 7]), putting stronger emphasis on the user side. Although they are of great value, it is also important to study this communication process from the designer perspective, analysing how tools an techniques can be used and improved to make it more effective. We believe that the combination of semiotic and cognitive methods can be a powerful resource to help us understand the human-related aspects of API design and software reuse.

Building on related work concerning the validity of new knowledge generated by inspection methods [12], we intend to adapt the Semiotic Inspection Method (SIM) to the evaluation of programming interfaces, analysing the communicability of these software artefacts. We expect that this type of qualitative research may produce interesting insights regarding the effect of using different approaches to API design, which can serve as input to a more specific quantitative research in the same subject.

In Semiotic Engineering, signs play a central role in the investigation of the message being conveyed from the designer to the user. Their analysis relies on a classification scheme according to the interactive conditions that express their representation. They are divided in three classes: static, dynamic and metalinguistic.

From the definitions in [12], static signs are "those whose representation is motionless and persistent when no interaction is taking place". Dynamic signs are the ones "whose representation is in motion regardless of users' actions or whose representation unfolds and transforms itself in response to an interactive turn". And, finally, metalinguistic signs "represent other static, dynamic, or metalinguistic signs".

The designer's message conveyed to the users is strongly influenced by the choices made when combining those different types of signs. This message can be described by instantiating the metacommunication template [10] :

"Here is my understanding of who you are, what I've learned you want or need to do, in which preferred ways, and why. This is the system that I have therefore designed for you, and

this is the way you can or should use it in order to fulfil a range of purposes that fall within this vision'.'

The signs that compose the software artefacts under analysis should be able to represent the implicit message described by the instantiation of the metacommunication template, and SIM's goal is to evaluate the communicability of these artefacts from the perspective of the designer. (i.e. the communication sender). The full description of the SIM method is beyond the scope of this paper, but it can be summarised as a sequence of five steps, where the first three *deconstruct* the designer's message by performing a segmented analysis of the different classes of signs, and the last two *reconstruct* it by integrating and interpreting the deconstructed signs. The result is a characterisation of the designer's message structure in terms of signs and meanings.

Also in the semiotic context, the work by Tanaka-Ishii [29] provides an interesting account of the use of signs in programming, stressing the role of identifiers in programs and providing a semantic classification for them in three levels:

- Hardware: an identifier represents a memory address that stores a pattern of bits
- Programming language: an identifier represents the definition or use of a variable, routine, module, etc.
- Natural language: an identifier represents a "message" from the programer who writes the code to another programmer who reads it

In the context of evaluating APIs as a communication artefact, the natural language level can be viewed as the most relevant, as it represents the designer intents when creating a software artefact. At this level, the appropriate choice of metaphors is an important resource for the effectiveness of an API design.

The inspiration for a combined use of qualitative research methods to evaluate APIs comes from a recent work regarding the investigation of visual programming environments and computational thinking acquisition [16], which proposes the use of discourse analysis and inspections based on Semiotic Engineering methods and the CDN framework. Although the nature of the object of analysis is different (visual programming environments vs. APIs), we intend to adapt the method used in order to obtain qualitative findings which are expected to be relevant to the field of API design.

The basic idea behind the combination of these research methods is to provide different perspectives for the same experiment and obtain a more complete cycle of analysis of the findings. SIM offers insights about sign systems and notations used by the designer to communicate his vision. CDN provides the basis for a cognitive analysis of the experiments, which is relevant due to the fact that the use of a new API implies a learning experience for programmers. These two methods may be complemented by discourse analysis by providing additional evidence to support or contradict other findings.

To the best of our knowledge, there is no report of a similar experiment using the combination of these methods. This means that there is no previous evidence that this approach is effective. On the other hand, there is a high expectancy of obtaining relevant results based on the application of these methods to the evaluation of visual programming environments.

4 Enhancing the communication of API design

In the previous sections, we presented how the design of an API may be regarded as a communication process, an how it can be evaluated. In this section, we discuss some concepts and techniques regarding API specification that may influence this communication by offering more expressiveness to the designer in order to convey his vision of the system to the programmers.

In our current work, the main expected contribution is the evaluation of API design from a communication perspective and the cognitive impact of the designer's choices by using a novel combination of research methods. This may serve as the basis for the comparison of different API specification techniques and tools, as a secondary contribution from this work.

The most common form of API specification is the combination of its syntactic elements written in a formal language and a textual description in natural language of the semantics of its operations and parameters, as well as its behaviour. The designer has a few options to represent his vision of the software artefact to the users, namely:

- names for operations and parameters
- types for parameters and return values
- textual description of semantics and behaviour
- exceptions and error codes to indicate misuse of API or unexpected conditions

The sole use of syntactical constructs in interface specifications limits the designer's options to be "present" at interaction time to provide more dynamic information to the programmer. In a programming environment that offers mechanisms for behavioural specification and runtime monitoring, the designer is able to include a more formal description of its intents that may give a richer interactive experience to the programmer when dealing with an API.

For instance, contracts [26] are a lightweight formal specification technique based on the use of preconditions, postconditions and invariants to describe the behaviour of software artefacts. This form of specification provides more powerful tools to the designer in order to describe how the interface is expected to be used, and which are the results of the operations depending on the calling context. This type of specification may interfere on the programmer's activity by giving feedback while code is written (static analysis), as well as when the program is executed (dynamic analysis), since the violation of any assertion may give more information about the cause and the nature of the problem.

From a cognitive perspective, the use of contracts may also have an impact on the programmer's work, since they provide a more precise description of the API behaviour than textual documentation, helping the programmer to understand the causes of possible errors by giving immediate feedback related to API misuses. Putting in CDN terms, contracts may have, for instance, a higher closeness of mapping to the API behaviour than textual documentation, and also make hidden dependencies between operations in the interfaces more explicit.

The use of formal behavioural specification languages [17] provide an even higher expressiveness to describe a software artefact, and allows the use of tools like model checkers to validate the specification. Although they can be a very powerful specification resource, they may also impose a higher demand for abstractions on the user, specially in mathematical terms, which can possibly have a negative impact on learnability. These are also interesting aspects to be investigated.

Beyond behavioural specifications, there are further levels that can be approached to describe a software artefact. In [1], the authors provide a classification of contracts in four levels: syntactic, behavioural, synchronisation, and quality of service. Most contract systems support the second level, described in terms of invariants, pre and post conditions, which consist of logical assertions to describe the system state before and after operation calls, supporting modular reasoning and runtime verification of these conditions.

The specification of synchronisation contracts can be a valuable resource in expressing the designer's intents, as they offer a formal definition of the allowed sequence of operations, which can be enforced at runtime to prevent an unexpected call pattern, informing the user the precise reason to why the API does not work as expected in that particular scenario.

Quality of service contracts open the possibility of specifying non-functional aspects of a software artefact that are more related to the execution environment or the preciseness of the results of the computation being carried out. Although they usually do not represent a correctness constraint, they offer the designer the opportunity to specify the limitations or requirements of an API in terms of its execution environment.

Runtime monitoring of these API constraints may allow a richer interactive experience to the programmer, as there is a constant verification of the designer's assumptions and intentions, with a corresponding "alert" in case of violation of these conditions. This signal may come in the form of a runtime exception, log or trace message, routine call, and so on. These mechanisms may pinpoint the source of API misuses and provide the programmer with a "higher level" message indicating what is wrong, which may also have a positive impact from the cognitive perspective.

We intend to investigate the aspects described in this section using the combination of methods proposed in section 3. We are currently selecting the appropriate tools and elaborating the experimental scenario in order to capture the most representative aspects in the context of API design and use, and we will build on our previous experience regarding user experiments, as well as on the reports from similar research, in an effort to make the most of the results.

5 Example of application scenario

In this section, we illustrate the concepts presented throughout the previous sections with an example scenario of application of the inspection methods, and the issues that may arise when investigating the design of a particular API from a semiotic and cognitive perspective. Once again, the example comes from the Java language, as it is a popular language that may be familiar to many readers.

The designers of the Java language and the core API created a single rooted class hierarchy based on the *Object* class. This class defines common operations to all Java objects. In particular, the method *Object.equals()* provides logical equality comparison, and its default implementation is as restrictive as possible, since it compares the object address in memory, which makes any object different from all objects but itself.

Another method in the *Object* class is *hashCode()*, which calculates a hash value for an object. The API designers created it in anticipation of the need of a standard way of allowing any object to be part of a hash-based collection such as *HashMap.* The result of the *hashCode* method determines the distribution of the objects in this kind of collection, which has a direct impact on the performance of the searching algorithm.

Although the *Object* class provides a default implementation for these methods, it is often necessary to override them. If the programmer wishes to provide logical equality for different instances of the same class, the *equals()* method should be overridden. For example, a programmer might consider two objects of a *Car* class equivalent if they have the same registration number. In this case, it would be necessary to provide a specific implementation for the *equals()* method. The method's documentation specifies that any implementation should satisfy the requirements of an equivalence relation, which means that it should be reflexive, symmetric and transitive. Also, it should be consistent, returning always the same value (true or false) when called, provided the internal state of the objects being compared do not change between calls.

There is a strong relationship between these two methods that is commonly overlooked or ignored by Java beginners. The "rule of thumb" says that, if one overrides one method, the other should also be overridden, in order to maintain the class consistence, since objects that are considered "equal" must return the same hash code. One typical consequence of not getting this right is that an object inserted into a hash-based container may never be recovered.

A quick search for terms like "java hashcode equals" in Google provides many results regarding tutorials, articles and discussions in development forums about this topic. The "Effective Java" book [4] dedicates almost 20 pages to the discussion of these methods, which shows that it is not a trivial subject. Due to the inherent complexity of this feature, the misinterpretation of the designer's intent behind the API specification may cause subtle defects that can be difficult to trace [21].

One particular issue related to this characteristic of the Java language can be found in the Java API itself. The class *java.net.URL* overrides the *equals()* method, based on the assumption that two URLs are equivalent if the name of their host components resolve to the same IP address.

The use of domain name resolution inside an equality comparison of URL objects is, by itself, a bad idea. It makes the *URL.equals()* operation dependent on the network status, which means that it may fail or take a long time. Also, virtual hosting allows two different websites to share the same IP address, which breaks the assumption that two URLs pointing to the same host may be considered equal.

Apart from being a bad design example, the URL class does not comply with the *Object.equals()* contract, since it breaks the consistency requirement. As it depends on factors that are external to the object state (i.e. network), the method may return different results between calls. Also, the timing issues related to name resolution makes the use of URL objects in hash-based collections impractical. For example, if a *HashMap* contains URL objects as keys, a *get* operation will compare the requested key with the collection elements, calling *URL.equals()* repeatedly, and each call will perform a host name resolution in the network.

The URL class example illustrates that the communication of API design intent may involve three different roles: the designer of the API, the implementor of the API, and the client programmer. In this case, the developer of the *URL* class misinterpreted the design of the *Object* class, and provided a broken implementation of the original intended artefact. The application programmer, at the end of the chain, has to deal with the burden of interpreting different "messages", one coming from the original *Object* class designer, and the other from the *URL* class developer.

In the context of this work, some of the research questions that arise regarding this example are: how does the Java API "communicate" these design decisions to programmers ? Could it be more effective ? What are the cognitive aspects involved, and what tools or resources could be used to help the programmers get things right, in the first place, especially for beginners ?

From a Semiotic Engineering perspective, the main signs used by the API designers in order to send their message to the users are the method signatures with names and parameters (static signs), the return values for these methods and other related operations like inserting and removing from collections (dynamic signs), and the textual description in the Java API documentation (metalinguistic signs). A detailed inspection might reveal if the signs are appropriate, and what changes or additions to the API could make this communication more effective. For instance, there could be better code examples in the documentation, methods to test a class implementation for consistence regarding these methods, formal specifications (e.g. contracts) that could be enforced statically or dynamically, and so on.

From a cognitive perspective, a CDN based inspection may provide interesting insights regarding this particular design. For instance, the issue described may be considered a hidden dependency between classes in the API, as it is not obvious at first, especially to a novice Java programmer, that inserting the class into a container may not work if the methods are not overridden. Also, any change in the internal structure of an object may impact its *hashCode()* or *equals()* implementation, which can be another example of hidden dependency, or even viscosity. Premature commitments may also arise when a programmer creates a new class, as it is necessary to anticipate if it will be used in a hash-based container or need a logical equality comparison.

The scenario described in this section can serve as the basis for a user experiment concerning programming tasks carefully selected to provide qualitative findings regarding a particular API design, combining the semiotic and cognitive approach, complemented with discourse analysis. It can also be used in the evaluation of the effectiveness of advanced specification techniques and tools for the communication of design intent of APIs.

6 Related work

This section presents a brief description of related work concerning the evaluation of API design and programming activities from a semiotic or cognitive perspective that inspire or influence our current research.

Clarke and Becker's work [8] is one of the first cognitive approaches to API design evaluation based on the CDN framework. They created a modified version of CDN in order to assess the usability of an object-oriented library, and conducted empirical studies based on a set of development tasks which had to be implemented by the participants during videotaped sessions. The results were analysed to extract patterns of behaviour from the participants that might help to identify problems in the library's design.

Maia et al. [25] present a qualitative method to evaluate the flexibility of middleware implementations based on a CDN inspection of representative adaptation tasks to be performed on the middleware platforms under analysis. Although this work is not specific to API design, it presents a good example of CDN instantiation to evaluate cognitive aspects of programming tasks, which is closely related to our objectives.

The work by Farooq et al. [15] describes API usability peer reviews, an inspection method conducted as a group-based walkthrough of the source code. They contrast this method to usability tests, arguing that both methods can be used in conjunction and complement each other, because peer reviews have a lower cost and shorter execution time, but identifies less usability defects than usability tests.

The work by Dubochet [14] evaluates programming languages as a medium for human communication, based on an experiment using an eye-tracking device and distributed cognition. Although the goals of this work differ from ours, it provides interesting insights about the programmer's cognitive experience when dealing with two different programming languages.

Cataldo et al. [6] performed a quantitative study of the impact of interface complexity on the error-proneness of the source code of two large systems. In their findings, they concluded that the increase in interface complexity leads to more bugs in the source code files that use these interfaces. This may be a good reference for future quantitative research based on our qualitative results, and reinforces the importance of evaluating the effectiveness of API design communication, specially when dealing with complex ones.

7 Final remarks and future work

In this paper, we discussed the importance of API design in the context of software reuse, and presented the motivation for the evaluation of software artefacts from a communication and human-centric perspective. Also, we proposed the use of a combined semiotic and cognitive method to perform this kind of evaluation, describing a typical scenario of application, as well as possible contributions.

We are currently refining the application of the methods to the API evaluation context, based on the previous experiments concerning visual languages, and selecting scenarios for a user study to maximise the relevance of qualitative findings. One possible scenario would be the evaluation of the Java API features described in section 5 by performing an experiment with undergraduate students in Computer Science. After performing the user experiments based on the described methods, we intend to analyse the results and report the most relevant findings in a future work.

We also intend to apply the combined semiotic and cognitive inspection methods to evaluate APIs which have been previously analysed in related works. This can be an interesting opportunity to experiment the methods in a context of API evaluation and compare the findings with the previous results, and also to improve the methodology itself.

In the long term, we expect to achieve the more general objective of providing a practical and effective approach to API design evaluation, in order to support software projects in which APIs are considered a critical asset.

Acknowledgements

Luiz Marques Afonso is partially supported by CAPES (PGCI-CAPES-013/10).

References

1. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.
2. Alan Blackwell and Thomas R. Green. Notational systems – the Cognitive Dimensions of Notations framework. In John M. Carroll, editor, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, Interactive Technologies, chapter 5, pages 103+. Morgan Kaufmann, San Francisco, CA, USA, 2003.
3. Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.
4. Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.
5. Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, 2005.
6. M. Cataldo, C. R. B. de Souza, D. L. Bentolila, T. C. Miranda, and S. Nambiar. The impact of interface complexity on failures: An empirical analysis and implications for tool design. *Technical Report CMU-ISR-10-101, School of Computer Science, Carnegie Mellon University*, 2010.
7. Steven Clarke. Measuring API usability. *Dr. Dobb's Journal*, 29:S6–S9, 2004.
8. Steven Clarke and Curtis Becker. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In M. Petre and B. Budgen, editors, *Proc. Joint Conf. EASE & PPIG*, pages 359–366, April 2003.
9. Aniket Dahotre, Vasanth Krishnamoorthy, Matt Corley, and Christopher Scaffidi. Using intelligent tutors to enhance student learning of application programming interfaces. *J. Comput. Sci. Coll.*, 27(1):195–201, October 2011.
10. Clarisse S. De Souza. *The Semiotic Engineering of Human-Computer Interaction*. The MIT Press, 2005.
11. Clarisse Sieckenius de Souza. *Semiotics: and Human-Computer Interaction*. The Interaction-Design.org Foundation, Aarhus, Denmark, 2012.
12. Clarisse Sieckenius de Souza, Carla Faria Leitão, Raquel Oliveira Prates, Sílvia Amélia Bim, and Elton José da Silva. Can inspection methods generate valid new knowledge in HCI? the case of semiotic inspection. *Int. J. Hum.-Comput. Stud.*, 68(1-2):22–40, January 2010.
13. Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 63–71, New York, NY, USA, 2004. ACM.
14. Gilles Dubochet. Computer code as a medium for human communication : Are programming languages improving ? *Psychology of Programming Workshop (PPIG 2009)*, pages 174–187, 2009.
15. Umer Farooq and Dieter Zirkler. API peer reviews: a method for evaluating usability of application programming interfaces. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 207–210, New York, NY, USA, 2010. ACM.
16. J.J. Ferreira, C.S. de Souza, L.C.C. Salgado, C. Slaviero, C.F Leitão, and F.F. Moreira. Combining cognitive, semiotic and discourse analysis to explore the power of notations in visual programming. *To appear in the Proceedings of VL-HCC'2012 - IEEE Symposium on Visual Languages and Human-Centric Computing.*, 2012.
17. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.
18. Michi Henning. API design matters. *Queue*, 5(4):24–36, May 2007.
19. Michi Henning. The rise and fall of CORBA. *Commun. ACM*, 51(8):52–57, August 2008.
20. David H. Hovemeyer. *Simple and effective static analysis to find bugs*. PhD thesis, College Park, MD, USA, 2005. AAI3184274.
21. J. Howell. What's the deal with Java equals() and hashcode() ? http://www.summa-tech.com/blog/2010/01/26/what's-the-deal-with-java-equals-and-hashcode/.
22. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Queue*, 9(5):20:20–20:29, May 2011.
23. John M. Daughtry Iii and John M. Carroll. Perceived self-efficacy and APIs. *Psychology of Programming Workshop (PPIG 2010)*, 2010.
24. Jeff Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42, April 2007.
25. Renato Maia, Renato Cerqueira, Clarisse de Souza, and Tomás Guisasola-Gorham. A qualitative human-centric evaluation of flexibility in middleware implementations. *Empirical Software Engineering*, 17:166–199, 2012. 10.1007/s10664-011-9167-7.
26. Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.
27. Martin P. Robillard and Robert Deline. A field study of API learning obstacles. *Empirical Softw. Engg.*, 16(6):703–732, December 2011.
28. Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *ICSE*, pages 529–539. IEEE Computer Society, 2007.

29. Kumiko Tanaka-Ishii. *Semiotics of Programming.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.
30. Jeannette M. Wing. Computational thinking and thinking about computing. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 366(1881):3717–3725, October 2008.

Sketching by Programming in the Choreographic Language Agent

Luke Church
Computer Laboratory
University of Cambridge
luke@church.name

Nick Rothwell
Cassiel
nick@cassiel.com

Marc Downie
OpenEndedGroup
marc@openendedgroup.com

Scott DeLahunta
Random Dance R-Research
scott@randomdance.org

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

Keywords: POP-I.C. artistic process, POP-III.C. visual languages

Abstract

We describe the Choreographic Language Agent, a programming environment designed for use by dancers and choreographers in the context of improvisatory composition methods. CLA provides a means for dancers to interact with a simple but powerful set of 3D geometric transforms, creating a wide variety of kinematic and dynamic configurations expressed in the form of phrases. These phrases can be composed in a dynamic visual arrangement, offering sophisticated facilities for provisionality and version control. Direct interaction with a live 3D rendering is a key feature of the system, although this rendering is only an intermediate product, designed to offer ample space for alternative interpretations when mapped onto dance movement. The result is a programming language that emphasises transience, ambiguity and creative flow rather than the conventional requirements of professional software engineering contexts.

1. Introduction

The Choreographic Language Agent (CLA) is a project directed by Scott DeLahunta within the R-Research arm of the London-based contemporary dance company Wayne McGregor|Random Dance. Working in collaboration with choreographer Wayne McGregor, CLA builds on more than a decade of research investigating the cognitive processes of making dance (e.g. McCarthy et al 2006). The early period of this research programme explored the sketch notations used by Wayne McGregor and members of the company as a component of the creative process. As in many other design disciplines, the formal notations associated with documenting dance (e.g. Labanotation, Benesh) are of limited value during exploratory design activity. A Cognitive Dimensions analysis (DeLahunta et al 2004) identified some of the notational strategies adopted by McGregor as improvised or habitual responses to the limitations of those more familiar conventions. During the period of this research, McGregor has been recognised in numerous ways as one of the world's leading choreographers (for example, he was appointed principal choreographer of the Royal Ballet, in addition to contributions to popular culture such as choreographing Harry Potter films and delivering a TED talk on the science of choreography), so the context of this research can reasonably be considered as representing respected leading approaches to contemporary dance.

The objective of the CLA project was to develop a computer support system that could contribute to this creative process. The term "language" refers to the need for novel notation, as well as the fact that each of McGregor's works is conceived as developing or extending new languages of dance, created through processes that include both studio improvisation and analytic perspectives involving multi-disciplinary collaborations.

2. Requirements

Extensive preparatory work with a wide range of academic collaborators, including a two day workshop with 12 scientists and 40 participants at a public session, made it clear that the scope of this project corresponded neither to any existing programming "language", or any existing choreographic notation. For those coming to the project from outside the world of professional dance (this includes some of the present authors), it is worth documenting some of the features that we find to be contrary to expectations among technical colleagues, although not necessarily surprising to those who have already worked in contemporary dance.

First, it was not considered necessary (or even desirable) for the display of the CLA to have any resemblance to the human body. On the contrary, dancers and choreographers find it both natural and convenient to represent configurations of human bodies using their own bodies. The CLA representation was intended to be open for interpretation either as movements of an individual dancer, or of a group, or a more abstract architectural form, not a simple maquette to be copied by a dancer.

Second, it was not required that the CLA should represent movements that fit within the motion constraints of the human body. On the contrary, Wayne McGregor's choreography is renowned for the way in which it extends the range of conventional body motion. To be a useful tool in McGregor's choreographic process (and that of many other contemporary choreographers), the CLA should present the dancers with a problem that must be solved through searching for new movements, rather than a solution in the form of a movement to be incorporated into the creative work.

Third, the CLA was not intended to create visual content for digital projection in a stage production. Although several of the authors have significant experience of programming visual arts commissions, and Downie has provided live computer graphics during Random Dance performances, the CLA is purely a creative tool, for use in the studio as one component of an improvisational process.

3. Strategy

Wayne McGregor often creates new work through a research process that requires dancers to conduct various conceptual or intellectual preparations, either in parallel with improvisatory exploration, or in advance of studio improvisation. During the period that CLA was being developed, several other scientific projects were also being conducted in association with the creative process – these included instrumentation of the dance studio with motion recording cameras, and introducing dancers to new conceptualisations of their dance process through the use of Barnard's Interacting Cognitive Subsystems framework (Barnard et al 2000).

CLA played a role that bridged the intellectual and embodied improvisation aspects of this process, by providing dancers with new abstract conceptions and representations of their work. Some of these drew rather literally on existing programming concepts – for example, the CLA display would include a verbal "language" in the form of geometric terms that can be aggregated into "phrases". These aspects of the programming language are analogous to familiar aspects of dance – individual movements that can be aggregated into dance phrases (although explorations of the linguistic structure within such phrases in contemporary dance does not find any clear token boundaries or grammar – McCarthy et al 2006). However some aspects were less typical of conventional programming tools – the program is a starting point from which to explore behaviour, rather than a precise specification of a required result. It is in this respect that it should be considered a sketching tool, rather than a software engineering tool.

Nevertheless, there is one respect in which CLA does draw on aspects of software tool design, which is to introduce some practices of software engineering into the studio context. We particularly emphasised models of version control, retaining intermediate work products, reviewing version history, and creating new branches in order to explore alternatives within an iterative and exploratory process.

4. Implementation platform

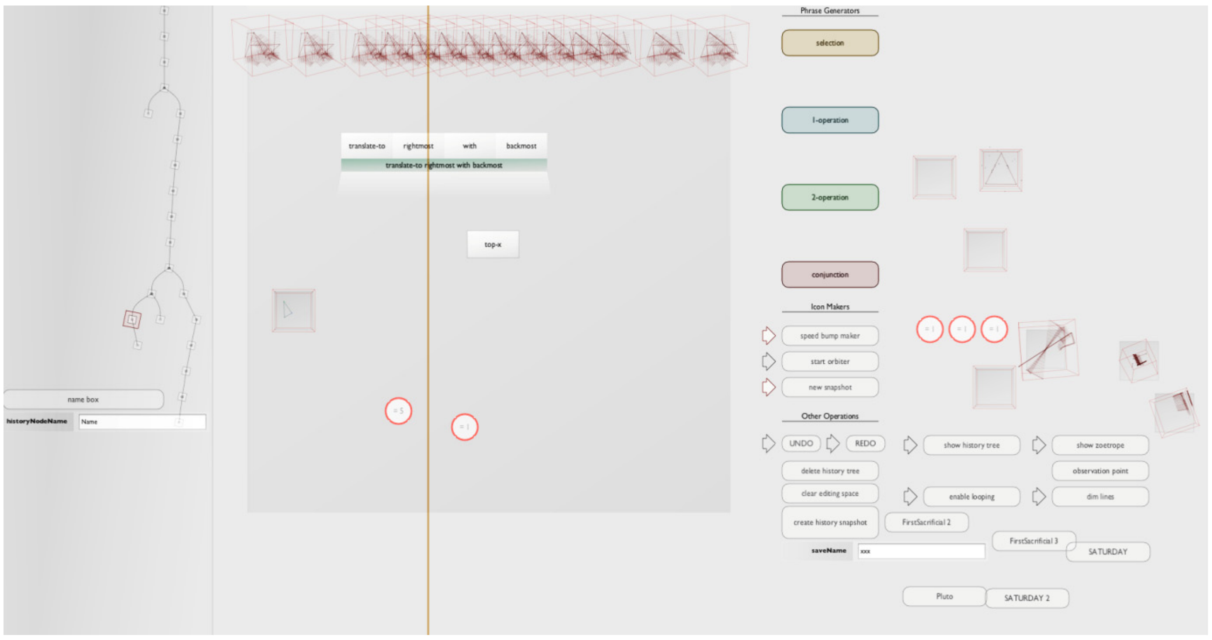


Figure 1 – the right-hand screen of CLA, as implemented using the Field drawing surface (individual features are discussed later). The execution cursor is represented by a vertical red line.

CLA was completely implemented within the Field environment from OpenEndedGroup (Downie 2008). Field is primarily a development environment for making digital art, but is sufficiently powerful that we were able to use it as the implementation platform for an interactive programming environment. In addition to relatively conventional text editing facilities, Field also includes a drawing surface that can be used either as a canvas, to arrange pieces of textual code, or to express control flow between other executable elements. In the case of CLA, control flow is represented as a cursor that sweeps from left to right across this drawing surface, triggering language phrases as it passes across them (Fig 1).

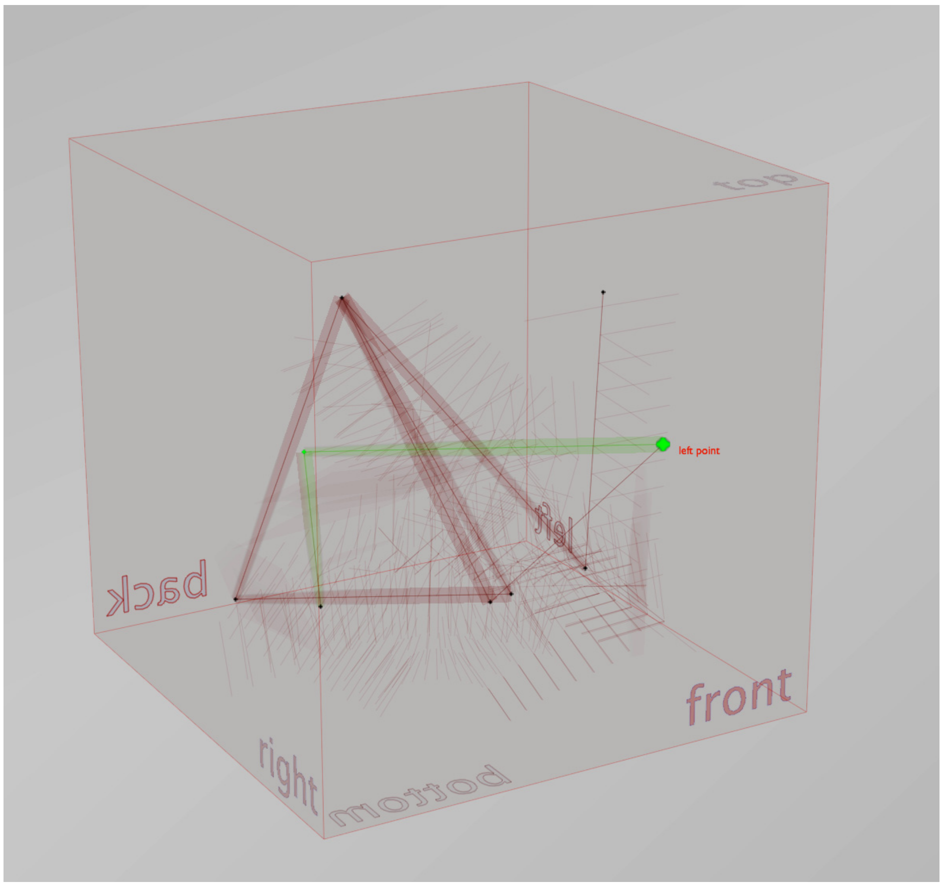


Figure 2- the left-hand screen of the CLA workstation, showing an abstract 3D figure animated within a virtual stage. The stage is represented by the surrounding box, with labels indicating which sides are the front/back, left/right of the stage. The thick lines have been created by the user to represent the skeleton of an articulated figure. The fine lines are added by the system as the figure moves, leaving a trace of that movement in the 3D space.

In the CLA deployments, Field is augmented with a second screen, which permanently displays a rendering of a 3D space – a "stage" within which abstract figures can be rendered and controlled by the language elements (Fig 2). The rendering of walls and floor around this space was an essential cue, both to navigation, and to understanding of the geometric figure as physically situated rather than simply an abstract "form in space". The view of this rendering is controlled by a 3D mouse (3D Connexion Space Navigator), allowing users to explore the figures by interactively rotating the viewspace during program creation or execution. The rendering of the geometric configuration in this space is intentionally minimal – points are simple spheres, connected by armatures. Both the points and connecting links are blurred, to emphasise their intention as material for interpretation, rather than precise specification of dance movement.

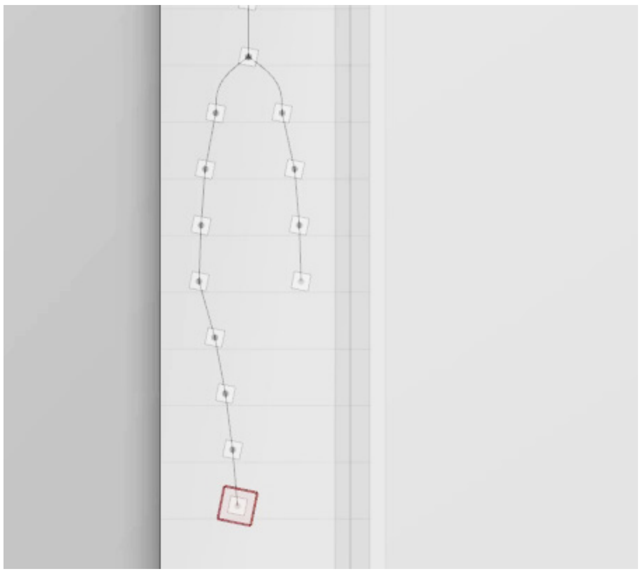


Figure 3 – detail of the CLA window showing the history tree

Finally, CLA relies on the Mercurial version control system to manage the capture and branching of exploratory changes. Every action within the Field drawing surface results in a change being logged to Mercurial, and users can return to any previous edit state by clicking on a node within a graphical tree at the side of the drawing surface (fig 3). Further changes then result in an automatic branch, starting from that node, and growing alongside the previous history. This provides considerably more power than conventional stack-based undo and redo (even when the whole stack is available, as in the Photoshop History palette).

5. Features of the CLA editor

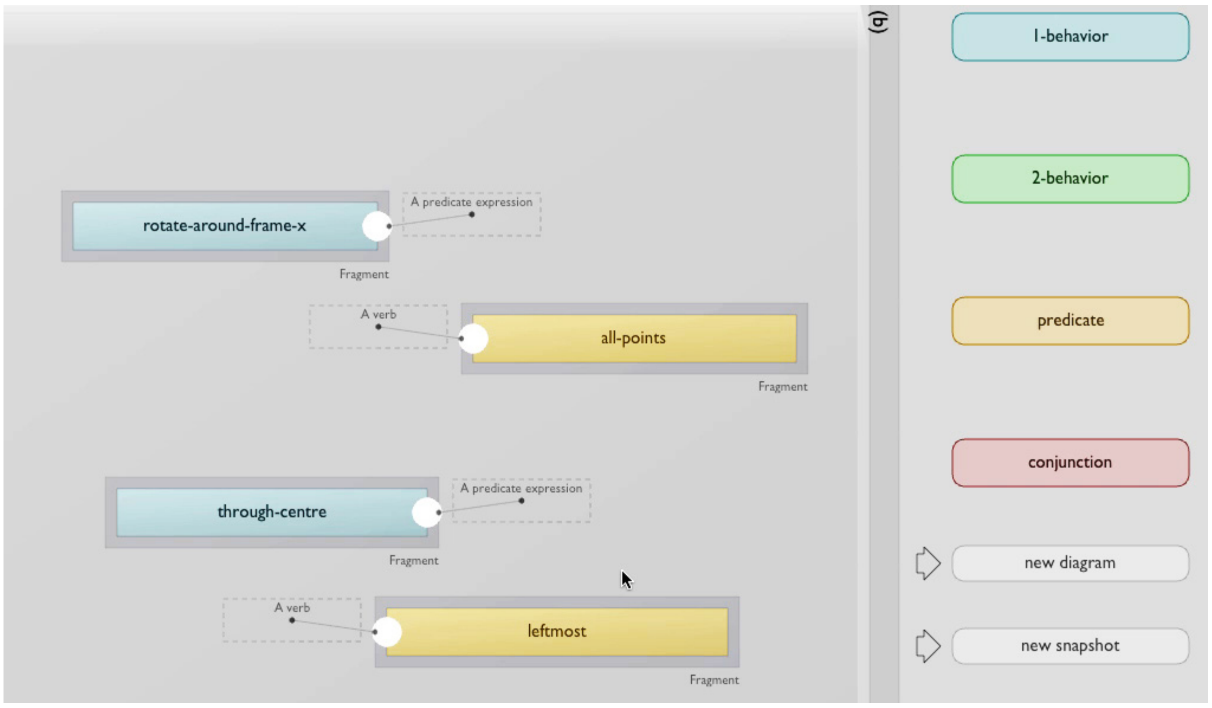


Figure 4 – the structure of the phrase language in CLA

The fundamental elements of the phrase language have three types, distinguished by colour-coding in the Field drawing surface (Fig 4). The first of these specifies a transformation to be applied to these points (e.g. “rotate-about-centre”). The second specifies one or more points, usually by reference to the 3D space (e.g. “leftmost”). The third specifies conjunctions of other phrases – either to be executed concurrently, or with one immediately following the other. Rather than the jigsaw-like visual cues that have become routine in systems such as Scratch, unbound connection sites are indicated more organically, as whiskers extending to the side of incomplete phrase elements offering hints about what might be placed there.

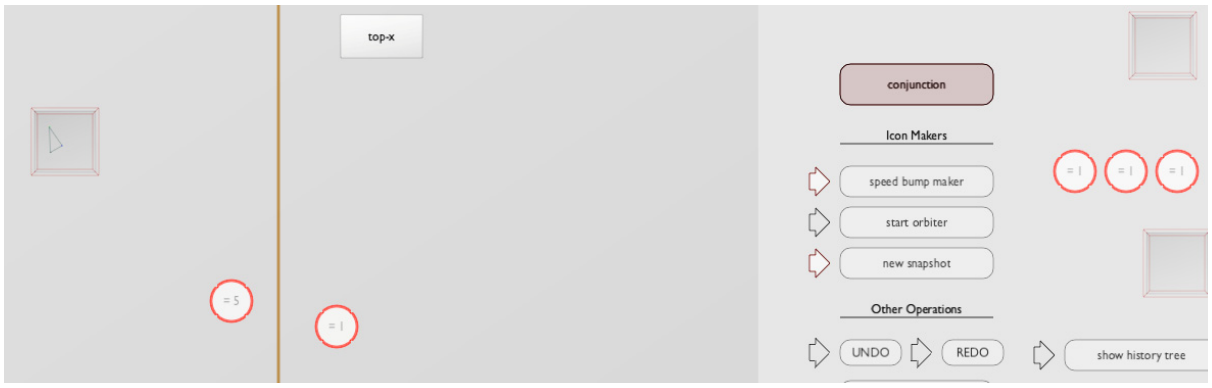


Figure 5 – “speed bumps” to change the relative animation speed

Any number of these phrases can be distributed around the drawing surface. On pressing play, the cursor moves across the drawing surface from left to right, animating the 3D transformations as it crosses them. The relative duration of a completed phrase corresponds to its width on the screen – it is possible to stretch or squash it by dragging the sides. The resulting combination of sequential and concurrent phrases of different durations can produce highly complex geometric effects. In response to initial trials, a feature was also added to control the global time-base – the speed with which the Field cursor moves. These “speed bumps” can be used to modify the dynamics of a composed piece independently of the relation between its elements (fig 5).

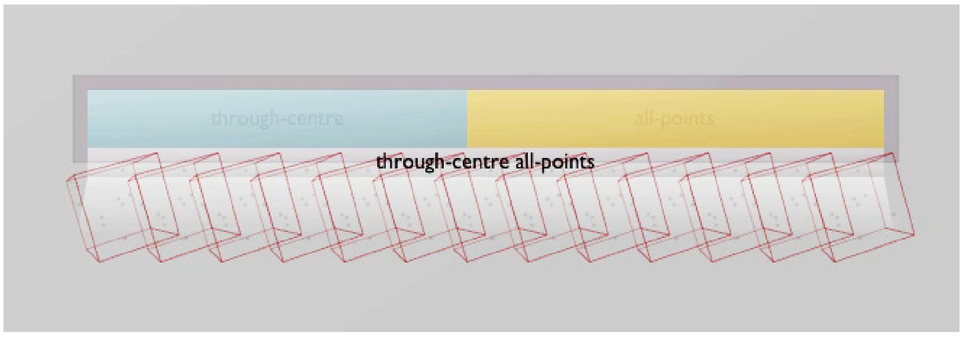


Figure 6 – Zoetrope, visualising the effect of a single phrase

As each phrase is executed, a sequence of snapshots is compiled, visualising at a glance the overall shape of the point trajectories. The whole series of snapshots is displayed above the phrases that generated them, in a form that we describe as a “zoetrope” (fig 6). Individual snapshots of the 3D configuration can also be dragged to the side of the screen, saved for reference, and used to initialise or combine system states. The zoetrope visualisation, although useful as a visual reference to a dynamic process, also has great visual appeal, leading us to provide zoetrope-like structures within the stage itself (fig 2).

As will be apparent from the figures in this paper, the visual style of CLA, as with the visual style of Field, is unlike that of most programming languages. Rather than bright colours, high contrast, and crisp edges, these systems intentionally avoid association with technical or educational diagrams. The intention of a diagram, like most computer programs, is to be unambiguous. But the CLA output must be open to interpretation. Although a given version of a CLA program has deterministic output (important, because it must always be possible for users to return to the state that produced a particular effect of interest), the visual and verbal rendering of the phrase should not constrain the potential reading of its output - and the two are necessarily linked, given that the creator and reader, in this end-user programming context, are often the same person.

6. Observations of use

The CLA has now undergone several iterations, in response to trials by Wayne McGregor and the members of Random Dance. Initial demonstrations made it clear that users would need extended tutorial support – this was initially addressed in the form of a live video produced by Nick Rothwell, walking through the system capabilities. Given the increasing prevalence of video walkthroughs as a tutorial medium for explaining system features of user-customisable platforms (examples range from YouTube videos of iPhone settings to live feeds from hack days), this deserves further study as a key element of end-user programming. In our first meeting after Wayne McGregor's initial period of experimentation with CLA, Wayne commented that he had watched the tutorial video so many times, it felt like Nick's voice had become lodged inside his own head!

As with many deployments of programming tools in highly novel contexts, some findings are relatively mundane from a psychology of programming perspective. Just as with a previous observation that live coding languages must still be usable when the programmer is inebriated (Blackwell & Collins 2005), we were relatively surprised to learn that a busy international choreographer does much of his "office work" in airport lounges or using WiFi in a Starbucks. The practical constraints of these contexts made the twin-screen setup and special 3D controller rather an obstacle in the choreographic process, by comparison to a compact laptop. It is almost certainly the case that a great deal of end-user programming research implicitly assumes a user who is sitting at a desk (or at least in a classroom), whereas field observation would probably identify a surprisingly large range of physical contexts and postures adopted by end-user programmers.

The use of a non-standard interface device (the 3D mouse) alongside a regular mouse/trackpad introduced consistency problems of button assignment and command shortcuts between the devices. These are relatively familiar everyday annoyances for programmers, but more disconcerting for professional dancers who might be best described as "social" computer users in their everyday lives. In retrospect, we might also have been more aware from the outset of the embodied interpretations – more salient to dancers – of gestures such as using the middle finger to select points, or the fact that the 3D mouse and regular mouse both assigned the same function to "left" buttons, despite the fact that a dancer (and perhaps any normal person) mirrors actions from right to left. The native Field environment also incorporated some of the features of MIT-style command chords (use of shift/ctrl/meta modifiers) that were inappropriate to an end-user audience, however convenient in the context of professional exploratory programming.

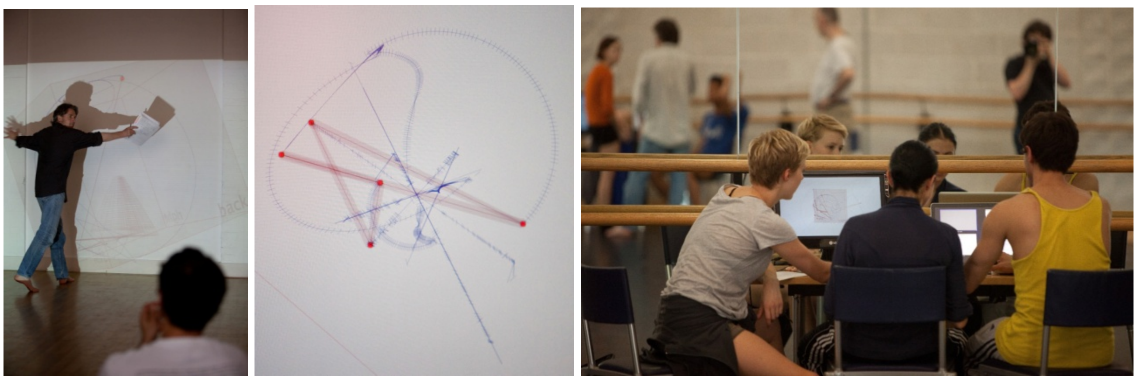


Fig 7 – 3 day workshop

The main period of research observation took place over a three day studio workshop, attended by all members of the company, together with the authors as instructors/facilitators, also providing technical support and observation (Fig 7). Some changes were made to the CLA operation at the end of each day – in particular, refinement of the model used to define the magnitude of the "speed-bumps", changing them from being relative to the previous speed (which involved fractional mental arithmetic to explain the results of successive speed bumps), to being expressed relative to an absolute reference value.

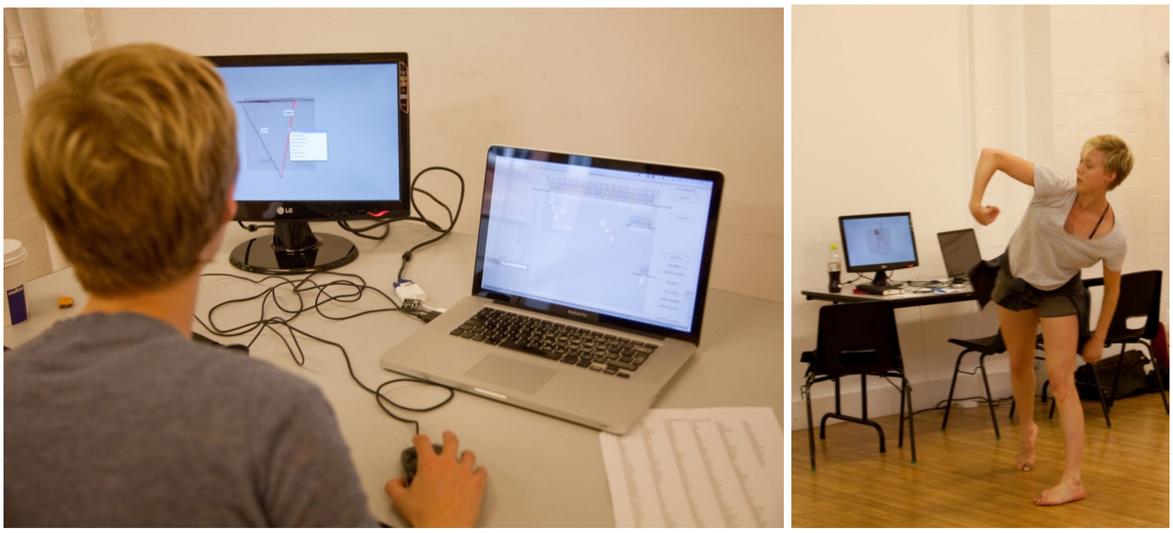


Fig 8 – a) Experimental sketching of a new movement sequence, followed by b) dance exploration of the geometry created.

The structure of the CLA sessions within the workshop was based around relatively long periods of experimentation with the CLA (from 90 to 120 minutes) (fig 8a), followed by extended periods (more than 60 minutes) in which the dancers worked to resolve the "problems" that they had set for themselves in the CLA animations, turning them into dance (fig 8b). The dancers worked in groups of 2 or 3 when interacting with the CLA (a constraint partly imposed by the number of workstations we were able to install in the studio, although group working was also seen as a valuable part of the creative process). Each dancer then developed a different interpretation of the kinematic and dynamic characteristics of the animation they had created. These interpretations could be as diverse as mapping joint configurations onto particular parts of the body (an arm, foot or head), mapping point motions onto the space of the room, or interpreting the dynamics of a swept line through jumping or rolling. Where groups of dancers worked together, this mapping-as-problem-solving became especially intense, as the dancers negotiated among themselves how the mapping might be achieved, and which aspects would be assigned for interpretation by which dancer.

On the second and third days of the workshops, those dancers who had become most confident in CLA operation had spent time overnight, planning new CLA programs that they would create the following day. The group activity then made a transition to these individuals as leader-operators, with other dancers observing and making suggestions to refine the animations. Given the status of the CLA animation as a creative sketch, in which the final "product" was the dance movements created, there was little desire to return to refinement of the CLA program after the transition had been made to working out the dance problem. In fact, even where there was a "bug" (as a programmer might call it) in the animation, dancers explicitly told us that they avoided the temptation to "fix" it, because the unintended behaviour was more valuable to them as an artistic challenge.

Observation of these sessions resulted in many small discoveries, refinements, and understanding of design decisions that we wished we had taken differently. The following discussion of these observations employs analytic concepts from the Cognitive Dimensions of Notations (Green & Petre 1996, Blackwell and Green 2003), with dimension names italicised in accordance with standard practice, as well as concepts from the Attention Investment model of abstraction use (Blackwell 2002) and Engelhardt's Language of Graphics (2002).

6.1. Version History

As with most version-controlled software development, our version history applied to the "code" of the transformation phrases, but not the "data" of the points arranged on the stage. Execution snapshots preserved data, but only if the code was executed. This distinction was of course invisible to our users, for whom a carefully constructed arrangement of points was just as valuable as the code that manipulated them – and thus a source of significant disappointment when it was accidentally deleted. Clearly, both code and data should be managed within a version history.

The version history itself was not as useful as we would have liked, because of the uniformity of the rendered nodes, making it difficult to return to interesting points. We added a facility to assign labels to nodes, but this requires the user to recognise in advance which versions are going to be the most interesting. This form of *premature commitment* is unrealistic in the context of an artistic process, especially considering the abstraction investment involved in stopping to think of a name just as the creative process is at its most exciting. Ultimately, we had a combination of *role-expressiveness* and *hidden dependencies*. Both the history tree, and "snapshots" of the 3D rendering, have related functions – to allow explicit reference to states within the creative process. Despite the fact that we were aware of this requirement, and designed these features to address it, we did not succeed in integrating them into a usable tool.

6.2. Sub-devices

As would be expected from Cognitive Dimensions analysis, we identified liberal use of sub-devices to enable offline processing or reduce *hard mental operations*. Some dancers copied pieces of geometry from the screen onto paper. Others composed explicit mappings from the screen onto the body of a dancer. *Juxtaposability* was a particular challenge – it was almost impossible for dancers to watch the screen while dancing. Some suggested that future systems might make the animated motion continuously visible via headset displays, or at least with wall-sized projections around the studio. In the absence of such equipment, substantial amounts of time during the dancing periods were spent walking over to the workstation, then watching the screen intently, attending to the specific aspects of the animation that constituted the current problem focus, and memorising them before returning to the dance floor. This hard mental operation was mitigated by the rendering of a blurry swept "sheaf" of lines that overlaid multiple frames over a second or so, thus allowing direction of movement to be seen at a glance (fig 2).

6.3. Visual feedback

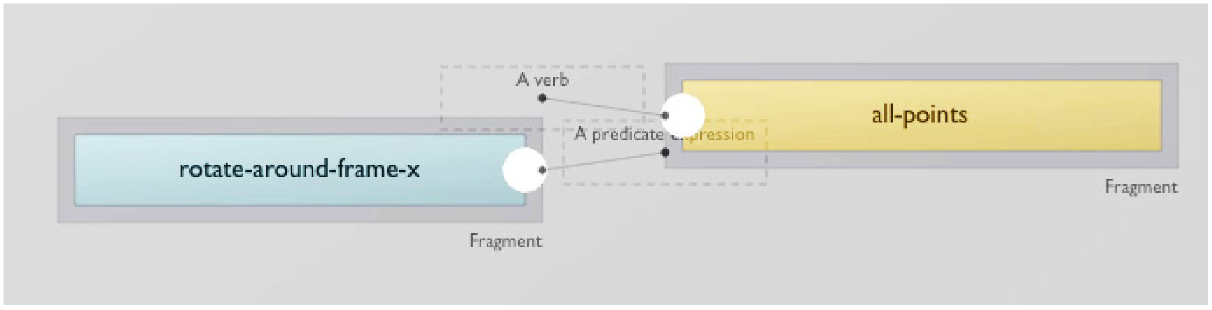


Figure 9 – visually unobtrusive syntactic cues in CLA

Although the visual style of CLA, like that of Field, is distinctively oriented toward the sensibilities of professional artists rather than engineers, this sometimes introduced unfortunate consequences for usability. The rendering of the phrase elements included a number of visual cues, generally designed to be unobtrusive and non-directive (fig 9). We were concerned that it should be possible to execute the program regardless of the degree of syntactic completeness (allowing *provisionality* and *progressive evaluation*), meaning that incomplete phrases were often present in the display area. There was a visual cue to indicate that a phrase was complete – the words underneath a complete phrase were highlighted with a continuous bar (fig 6), while incomplete phrase elements used highlighted open circles and whisker-hints to draw attention to the location of missing elements (figs

4 and 9). However, none of the dancers noticed this visual feedback until the second day, meaning that they were often unsure whether particular elements of the display would or would not have an effect on the execution behaviour.

6.4. Secondary notation versus verbal reference

Another unanticipated drawback of the visual notation (and one that is not directly reflected by a Cognitive Dimension) is that the collaborative nature of the studio work makes it necessary for dancers to discuss elements that they see on the display. In a textual programming language, it is always possible to simply read out a passage of text if it is necessary to refer to it in speech. With the visual notation of the CLA, we heard sentences such as “you need to hold that thing and go past that thing”. To some extent, additional *secondary notation* would have helped with this problem of verbal reference – for example, changing the colour of one or more points so that they could be referred to as a group. However, any such visual detail carries the danger that users might be distracted by making something “pretty” – they observed that it was not always easy to predict, from the visual properties of the animation, whether it would be an effective starting point for dance. In particular, some animations that were visually beautiful offered little opportunity for further improvisation, while those that were complex and hard to read were “rich” as a source of potential movement cues.

6.5 Explicit representation of dynamics

Despite the fact that CLA is a domain-specific language, designed specifically to meet the needs of this group of dancers, there were some respects in which we seem not to have achieved the *closeness of mapping* that might be expected in a DSL. In reflecting on the three day workshop, many dancers spoke of the way in which they struggled to produce the particular animation effect that they wanted. They described this in terms such as “bouncing”, “momentum”, “energy” and “vroosh”. In retrospect, these are all ways in which we might expect dancers to describe salient aspects of abstract motion (although they derive in part from a choreographic task used in the workshop, which involved interpretation of expressive motion terms). To some extent, the “problem-solving” aspect of the improvisation process depends on removing any obvious mappings – as in the avoidance of direct body representations. Nevertheless, one dancer who had found CLA least engaging said that it took at least 10 minutes to create anything he could relate to the body, and 40 minutes to make something interesting. It is possible that a language paying more attention to dynamics, as well as kinematics, might provide more expressive power in this respect.

6.6 Lack of syntactic constraint

Our focus in this project on creative arts practices offers a relatively extreme example of exploratory design activity. This has allowed us to observe some Cognitive Dimensions trade-offs that may not have occurred in a more conventional software engineering context. For example, syntactic constraints on drawing area manipulation were made to be absolutely minimal – the syntactic elements can be dragged anywhere at any time. In principle, this provided a high degree of *secondary notation* (with respect to location in the visual plane), and minimum *premature commitment*. However this extreme design choice unexpectedly resulted in high *viscosity* – because any element of a phrase could be dragged elsewhere at any time, it was then necessary to move the other components of the phrase to reassemble it. As with classically viscous visual dataflow languages, we watched CLA users spending a lot of time reassembling phrases in order to move them to different positions on the screen. However, whereas visual dataflow languages like LabVIEW present a tradeoff between *hidden dependencies* and viscosity in the routing of the dataflow connection paths, the CLA exhibited similarly high viscosity through its lack of syntactic boundary constraints. A more reasonable approach to visible syntax maintenance can be seen in the Scratch language – although it is possible to drag syntax tiles out of the expression they belong to (and sometimes to do this accidentally), most dragging operations affect the whole of a syntactically bounded expression. Many such languages seem to rely on an implicit virtual “syntactic physics” that simulate gravity, magnetism or adhesion to provide the operator with subtle interactive cues – ToonTalk provides another case study, in which manipulation can be *error-prone* due to lack of physical resistance to significant syntactic changes.

7. Conclusion

The context of the programming tasks in this project offers an intriguing counter to the usual practices of software engineering, in which a sketch might be made on paper or whiteboard before translating it into a code "product". At that stage in a software engineering project, there is certainly little desire among software engineers to return to preliminary conceptual sketches and revise them after the product has started to take shape. In CLA, we have created a programming environment whose purpose is conceptual sketching, and whose output is a transient representation to be discarded, rather than a final product.

In some ways, this might be compared to student programming exercises, which are also transient, and not the "final product" of the programming course – that product is a qualified programmer rather than a program. However, it is also useful to consider this transient form of programming within the context of live coding, where the program is improvised in front of an audience (Blackwell and Collins 2005).

Experimental languages of this kind are valuable in two respects. Firstly, they offer an opportunity for programming to become more inclusive, through creating languages to support users with different skills and work practices. Secondly, they help us to explore boundary cases that test general principles of programming language design outside of typical design parameters. The CLA project has, of course, also made contributions to choreographic practice and arts research (e.g. Blades 2012), and dance material developed in the workshops described here has become a component in new work for Wayne McGregor|Random Dance. However this research also extends beyond these immediate objectives, leading to new design-oriented programming languages, for contexts in both arts and business, that are now under development.

8. Acknowledgements

CLA was supported by Portland Green Cultural Projects. Luke Church's research was supported by Kodak. Nick Rothwell's work was supported in part by a Digital Futures in Dance residency at Dance Digital.

9. References

Barnard, P., May, J., Duke, D. & Duce, D. (2000). Systems, Interactions and Macrotheory. ACM Transactions on Human-Computer Interaction, 7, 222-262.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.

Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of PPIG 2005*, pp. 120-130.

Blackwell, A.F. and Green, T.R.G. (2003). Notational systems - the Cognitive Dimensions of Notations framework. In J.M. Carroll (Ed.) *HCI Models, Theories and Frameworks: Toward a multidisciplinary science*. San Francisco: Morgan Kaufmann, 103-134.

Blades, H. (2012). Creative computing and the re-configuration of dance ontology. In Proceedings of Electronic Visualisation in the Arts (EVA 2012), pp. 221-228.

DeLahunta, S., McGregor, W. and Blackwell, A.F. (2004). Transactables. *Performance Research* **9**(2), 67-72.

Downie, M. (2008). Field - a new environment for making digital art. *Computers in Entertainment* 6(4).

Engelhardt, Y. (2002). *The Language of Graphics. A framework for the analysis of syntax and meaning in maps, charts and diagrams* (PhD Thesis). University of Amsterdam

Green, T.R.G. & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' approach. *Journal of Visual Languages and Computing*, 7,131-174.

McCarthy, R., Blackwell, A.F., DeLahunta, S., Wing, A., Hollands, K., Barnard, P., Nimmo-Smith, I and Marcel, T. (2006). Bodies meet minds: Choreography and cognition. *Leonardo* 39(5), 475-478.

A Field Experiment on Gamification of Code Quality in Agile Development

Christian R. Prause[1], Jan Nonnen[2], and Mark Vinkovits[1]

[1] Fraunhofer FIT, Germany
christian.prause@fit.fraunhofer.de
[2] University of Bonn, Germany
nonnen@cs.uni-bonn.de

Abstract Internal quality of software reduces development costs in the long run but is often neglected by developers. CollabReview, a web-based reputation system for improving the quality of collaboratively written source code, was introduced into an agile development team. The goal was to improve the quality of developed source code as evidenced by the amount of code entities furnished with Javadoc comments. A money prize as an extrinsic reward and peer-pressure in form of a published ranking table were tied to reputation scores. We report on the conduction of a field experiment, our observations and experiences, and relate the results to answers from concluding interviews. Although the gamification had less effect than we had hoped, our experiment teaches valuable lessons about social effects and informs the future design of similar systems.

1 Introduction

We investigate the efficacy of a reputation system as a tool to improve source code quality. According to the Oxford Dictionary, reputation is what is generally said or believed about the abilities or qualities of somebody or something. A reputation system is a software that determines a user's reputation from his actions. Scores are computed to predict future user behavior or to create peer-pressure. Reputation systems are a core component in web-based communities, where they promote well-behaving and trust (Jøsang et al., 2007).

Writing documentation is a form of well-behaving in software projects. A problem is, however, that "developers don't like to do documentation, because it has no value for them." (Selic, 2009). Source code, in particular, combines executable instructions relevant to machines with human-readable documentation. It is an important a medium of communication between humans (Dubochet, 2009). Comments, for example, include "background and decision information that cannot be derived from the code" and are "one of the most overlooked ways of improving software quality and speeding implementation" (Raskin, 2005). The developers' dislike for documenting leads to a lack of internal quality, which has become a pervasive problem in software projects (Prause, 2011). Software quality is a complex concept without a simple definition or common way of measuring it. ISO 9126-1 defines quality as Functionality, Reliability, Usability, Efficiency, Portability and Maintainability. We focus on understandability and documentation of source code as means of improving Maintainability and internal quality.

CollabReview is a reputation system for collaboratively written texts like source code. While responsibility is essential for preventing careless development and achieving quality, it is difficult to assign. CollabReview acquires responsibility information from the documents' evolution history. By making personalized data about contribution quality available, it enables self-monitoring and learning processes within a development team (Prause and Apelt, 2008). Experts believe that reputation systems can improve source code and documentation quality in agile projects but are difficult to get right (Prause and Durdik, 2012). Prause and Eisenhauer (2012) compared CollabReview's scores to actual social reputation, and found correlations of $r = 0.64$ (source code) and $r = 0.88$ (wiki). Dencheva et al. (2011) report suitability for wikis.

This paper presents a field experiment of CollabReview in a software project with the purpose of improving in-line documentation (Section 2). As Section 3 shows, the effects of the

CollabReview intervention did not have the expected effects, and with regard to this, the experiment failed. However, we interviewed our developers afterward to investigate what happened and to learn success factors for future reputation systems for quality improvement (Section 4). Section 5 discusses threats to validity. Related work is reviewed in Section 6. Section 7 summarizes the insights we gained from the "autopsy". They are a major contribution and assist the future design of similar systems. Section 8 looks out to future work and concludes the paper.

2 Experimental setup and methodology

This section describes the research methodology of our field experiment and its setup. Harrison and List (2004) define a field experiment as an experiment taking place under "natural" conditions regarding several dimensions like the subject pool or the environment the subjects operate in. They note that there is no sharp line between lab and field experiments and that there are often varying degrees of naturalness. While limiting experimental control, field experiments have the big advantage of a natural context which is essential for studying human behavior. Therefore the size of the study is small: it is the typical size of an agile team. Note that any presented students' names are pseudonyms taken from the Simpsons cartoon series. The postmortem autopsy of what went wrong based on developer feedback follows in Section 4.

2.1 XP-lab 2011

The eXtreme Programming Lab and Seminar (XP-lab) was a post-graduate teaching activity. The lab takes students into the daily work of software development, familiarizes them with agile development methodology, and generates software artifacts that are intended to be actually used in the department's work. Its realism made it a favorable environment for our field experiment. The topic was to improve a static analysis toolkit for Java software. The toolkit's analysis components are written in Prolog, while the library, its Application Programming Interface (API) and IDE integration are written in Java. Our study considered only the Java parts.

The development team consisted of instructors and ten student developers doing agile development with pair programming. The students had previously received training on the topics software engineering in general and agile methodologies in special lectures and seminars. Solid programming experience was expected. The project ran for full four weeks. Development took place in a large office space from 9am until 5pm to 6pm every day. 725 revisions were committed to the Subversion repository, including a few contributions from non-lab participants.

2.2 A pragmatic definition of internal quality

Maintainability means "the capability of the software to be modified" (ISO 9126-1, 2001). The idea of coding rules is to ease the understanding of source code by reducing distracting style noise so that it gets easier to read (King et al., 1997; Seibel, 2009; Spinellis, 2011). We acknowledge that neither maintainability nor understandability are all about rules-compliant code. For instance, identifier naming is an important part of making code understandable. However, since only style (but no understandability) checkers are readily available, following coding rules is a pragmatic decision.

How to measure internal quality was a matter of discussion prior to the lab. Consensus was that quality was to be defined through the understandability of source code. Yet the full set of Java code conventions that the Checkstyle[1] toolkit can check were not considered sensible by all discussants. Some rules were considered as too much to not hinder the effective realization of functionality, too constraining or, like "do not use tab characters for indentation", even counter-productive. Also, the organizer's wanted to discuss some rules with the developers

[1] `http://checkstyle.sourceforge.net`

while development was on-going. In the end, only rules regarding the use of Javadoc (API documenting comments) were left. As smallest common denominator, internal quality would be defined through the correct use of Javadoc comments.

Checkstyle was configured to only check for completeness of Javadoc documentation: every source code entity missing a Javadoc comment or according tags (like @param, @return or @throws) was a rule violation. The higher the density of violations (i.e. violations per line of code) in a file, the lower its quality rating. A file not missing any comments had a quality rating of +10, while higher violation densities caused lower ratings. With this definition, a file's quality rating could be arbitrarily low, so we limited it to −10.

2.3 Control and experimental phases

The arbitrary re-combination of developers in programming pairs and their small number made it infeasible to split them into a control and an experimental group. So instead, the project was split into two phases: a control and an experimental phase. First, comparison data would be collected during the earlier phase without the developers' knowing. In the second phase, after about half of the project duration, the intervention would start. The experimental phase lasted for 8 working days and served to measure the effect of CollabReview. We call the day when the intervention started "day 0".

2.4 The CollabReview intervention

CollabReview accesses the project's revision repository to determine personal reputation scores. A developer who has contributed to good files has a higher score than a developer who contributed more to rather bad files. Developers are held statistically responsible for the quality of their source code. Their score is the average quality of files that they contributed to. In the best case, if a developer has only contributed to files with a quality rating of +10, then his own score will be +10, too. The average quality is twice weighted: once by the contribution ratio of the developer for that file (more contribution means more weight) and once by size of the file (larger files have a bigger influence). For example: 75% of the lines of a file (size 1) were written by Alice, while 25% were contributed by Bob. Its quality rating is $q_1 = 8$ (good), leaving Alice and Bob with quality scores

$$s_A = \frac{0.75 \times 1 \times 8}{0.75 \times 1} = s_B = \frac{0.25 \times 1 \times 8}{0.25 \times 1} = 8$$

But Alice also solely wrote a file twice as big and with a quality rating of only poor $q_2 = 0$:

$$s_A = \frac{0.75 \times 1 \times 8 + 1 \times 2 \times 0}{0.75 \times 1 + 1 \times 2} = \frac{6 + 0}{2.75} \approx 2.2$$

If Bob also contributed 50% to a flawless file $q_3 = 10$ with size 1, he would have $s_B = 9.3$ points.

The beginning of the experimental phase was signified by an introductory email sent to all participants. It explained that developers were taking part in the experiment, how their personal scores depended on Javadoc in their code, and what Javadoc would be expected. It was noted that the measurements were not totally accurate but reflected trends.

CollabReview then sent all developers a daily email in the late afternoon. This email contained a ranking list of all developers (including their individual points), and repeated a short explanation of how developers could improve their quality ratings by writing Javadoc. In addition to the published ranking list, a 30EUR Amazon voucher was announced as a prize for the developer who would have the highest reputation score in the end. A short information on the top scores was given in the standup meeting at the end of each day. Standup meetings are used in agile development to discuss current issues and to coordinate work between team members.

There were two partly conflicting goals set for the project: the primary goal was to deliver a functional software with a hard deadline, while the secondary goal (supported through CollabReview) was to obtain high quality source code. These two goals are conflicting to some degree in the short term. In particular, we find that a developer's contribution quality and quantity were correlated at intervention start ($r = -0.70$) and end ($r = -0.43$), respectively. But although the lab was graded, neither source code quality as determined by CollabReview nor the quantity of contributions would affect a student's lab grade. Consequently, even if improving quality would cost developers their time, this would not affect their grade in neither a good nor a bad way. Developers were free to invest (almost) any amount of time into writing Javadoc comments and they were expected to document public API entities well. The only trade off between implementing and documenting was that of having less time for other implementation activities when documenting, just as in any real-world software project.

2.5 Observations during and notes on the conduction

During the daily standup meetings shortly after the CollabReview intervention had begun, the students expressed some confusion about how their reputation scores were computed. Also, in the working time further murmur on the scores was observed by the teaching staff. Still not many explicit questions were asked by the students about the scores.

A bug in the rank computation algorithm led to wrong reputation scores being published via email. Some users had actually a different rank than the one published in the daily email. Luckily, however, this mishap did not affect the winner of the prize, who was Millhouse in both cases. In fact, the mishap later turned out to be fortunate when interpreting the results.

3 Effects of the intervention

This section analyzes the reputation data that was collected during the field experiment.

3.1 Contributed code quantity

Figure 1 shows how the amount of code contributed by individual developers evolved over time; measured in 100 non-empty lines of source code (HSLOC). The figure starts at day -11, which was the first day of the project. More developers appear during the early days of the project when making their first commit to the code base. In general, the figure does not contain any surprising results: As more and more code is developed, the developers' contributions increase.

The figure does not count days on weekends. Therefore, day 0, which marks the start of the intervention, is after a little more than two weeks into the project. The project ended at day 7, one and a half weeks after the intervention start. There is a sudden increase to a high peak and then a rapid decrease to a normal level for Sanjay. This phenomenon results from copying a huge amount of code into the code base, which is later removed or edited by other developers.

3.2 Quality reputation scores

Figure 2 is more important with regard to our study. It shows how the developers' individual reputation scores for quality evolved over time. Especially in the beginning, these scores jumped up and down. The reason is that initially the developers have contributed only few code, while reputation scores are computed from the average quality. When the amount of code authored by a developer increases, each line will have less influence on the average. The smooth line shows the average score of all developers with scores weighted by their developers' quantity, i.e. a developer's score has a higher weight when that developer has contributed more code. The average score is therefore equivalent to the code base's overall quality. A larger code base also

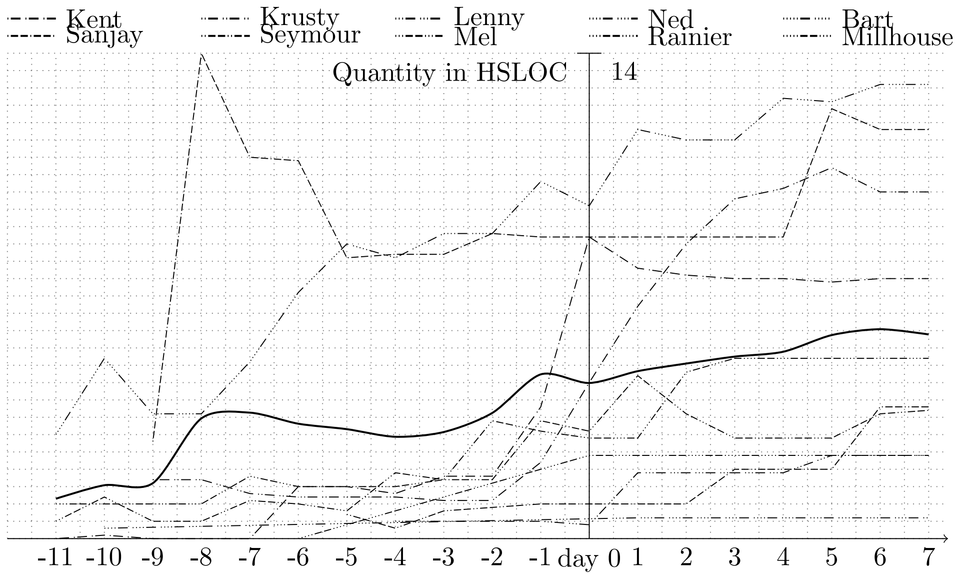


Figure 1: Amount of code contributed by the developers

leads to a stabilizing average score. The simple reason is here, too, that with more code a single change will have less effect on the average.

But the figure also shows that the intervention (starting at day 0) had only a small influence on code quality. The intended effect of increasing the code quality is not supported by our data. In fact, there is actually a small decrease in quality because day 0 coincided with a quality peak. We therefore conclude that the intervention did not have an adverse effect, too. It just did not show any significant effect. One could argue that quality scores are confounded by an end-of-project deadline. It may be that developers would normally have sacrificed quality in favor of functionality to complete all their tasks before the software is delivered. Our intervention could then have prevented that from happening. But this interpretation remains speculative.

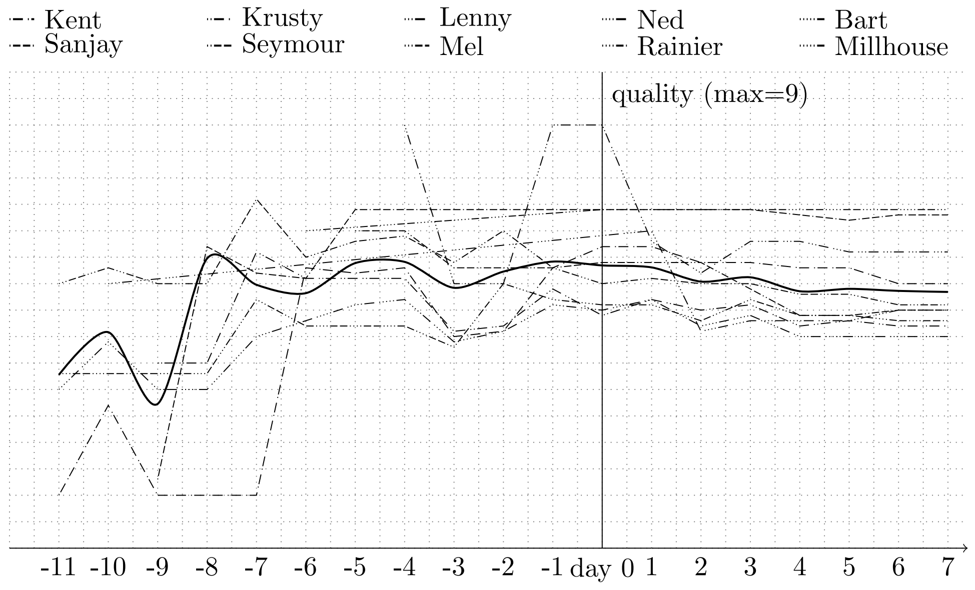


Figure 2: Average quality of authored code for each developer

3.3 Reputation scores and Subversion commit messages

We wanted to know if there is a correlation between the quality of a developer's Subversion commit messages and his quality reputation. The reasoning is that a developer who does write few documentation inside code will also neglect documentation in commit logs.

We went through all revisions created during the experiment and their according log messages. 471 revisions were created by the 10 developers who were part of the experiment (another ≈ 300 revisions were committed by other developers working on other sub-projects); some 60 revisions were not counted because they were submitted within seconds of an earlier commit with an identical log message, hinting that the developer just split his commit into parts. Each commit message was classified as bad (completely useless, e.g. empty, “commit”, ...), ok (some effort made but not much information payload, e.g. “implemented the concepts”), or good. According to this classification, 342 commits were considered as good, 83 as ok, and 46 as bad.

We found a Spearman correlation of $r_s = 0.55$ between a developer's final quality score and the amount of non-bad commit messages among all his messages. It may be possible that not all developers were familiar with the commit message concept from the start of the project. Therefore we also looked at the messages that were created during the second half of the project only, when they had had time to learn the concept. Here the correlation is $r_s = 0.61$, which is statistically significant for $N = 10$ at $p < 0.05$. We infer that a developer's tendency to provide good commit messages hints at how likely he is to provide Javadoc in his code.

3.4 Summary

We see that the amount of code authored by the individual developers increases during the project. At the same time, there is some fluctuation in their quality scores which stabilizes over time as more and more code is available. However, the quality graph does not reveal any significant change due to our intervention. It seems that the intervention was not appealing enough to alter developer behavior. In the next section, we investigate what the reasons for this ineffectiveness could be. We could show that quality reputation scores seem to be related to a developer's willingness to provide Subversion commit messages. As this is as expected, it indicates that our reputation scores are sound values.

4 Developer feedback

The purpose of the final interviews was to learn for the future design of similar interventions when it became foreseeable that the experiment would fail. We wanted to identify and understand critical aspects. The interview was based on a questionnaire with free text questions to qualitatively into reasons (see Section 4.1), and Likert items for quantitatively capturing the importance of certain factors (see Section 4.2). We investigated eight human “feelings” that we and colleagues deemed important to learn whether our intervention was

- present — How present were the reputation scores in your consciousness?
- fair — How fair were reputation scores suited to your work?
- important — How important were the reputation scores for you?
- understandable — How well did you understand the way reputation scores were computed?
- likable — How much did you like that reputation scores were computed at all?
- interesting — How interesting was it for you that there were reputation scores?
- fitting — How well do you think reputation scores fitted into the lab situation?
- acceptable — How acceptable are reputation scores for you?

4.1 Qualitative results

Some developers perceived the presence of reputation scores as high due to their competitive effect, and due to the daily emails sent out to developers. However, some developers noted that this single email created ups and downs in presence. It could even happen that — because emails were sent in the evening — developers had forgotten about it until the next day. Another one ignored the emails because he did not consider his work (external API) documentation relevant!

Fairness was one of the more problematic aspects of our intervention. One developer stated that his score changed when he added a comment, and so he thinks it was fair. Others, however, criticized that the algorithm was strange and that they did not know how they could achieve good ranks. And it was mentioned that by contributing to files without writing comments but where others contributed them, one could gain reputation, it could create an impression of arbitrariness. Additionally, fairness was negatively affected by the pair programming because someone else might commit on one's account, and although two people are responsible for the code, only one is credited for it. The reality in the lab was not reflected very well, including the lab ideology that instead "the naming should be expressive" making description unnecessary.

Regarding importance, one problem was that reputation scores showed up late in the lab. Of course, the reason for this delay was the research methodology, but it made the scores come in as a surprise. Again one developer mentioned that for his kind of work, Javadoc was not important. So he considered the reputation scores as pointless and unimportant. Also reputation scores were perceived as being in conflict with "too much other stuff", and especially the priority "to implement a running application." But one developer noticed this conflict could be "just a lame excuse of not adding comments." Among the aspects that made reputation scores important were "for improvement of my work" and again the competition with other developers.

Understanding of the reputation scores was rated as the severest problem in the trial, and at the same time had a relatively high influence on the ranking. Only one developer thought that "the explanation was clear", but even he only expressed the hope that it was correct. Although developers "knew the idea of comments", it was the score that caused problems with understanding. More detailed information on the algorithm was missing and "the process of calculating the score was not clear" to some. One developer just "did not care about it" and so did not try to understand.

Developers liked the reputation comparably well, although there was some concern that it was not relevant. Other developers felt "bad to be second not first", "did not care about it", were not "following the score that keenly", or just had a diffuse dislike for it. Still the idea itself was not seen as a problem and liked "as a motivation for writing good documented code".

The perceived importance of reputation scores was negatively impacted by feelings of meaninglessness and priority conflicts. But the competition made reputation scores highly interesting.

The reputation concept was considered as fitting comparably poor into the evaluation environment. Although it was not precluded that it still "might fit", "applying it to our lab was not beneficial because of our lab nature." A major problem was pair programming because of group effects of pair work and the loss of contributor information in commits. Also Feature Driven Development was perceived as being in conflict with reputation scores. One developer mentioned that he noticed low acceptance among his peers which led to few commenting, and another said that he does not like the tendency in modern life to measure everything. However, others are "in agreement with what my score is & how it was computed", or found that it is a "nice idea as long as it is only for us, not grades/payment/other assessment of work done".

4.2 Quantitative results

Likert-scaled responses are presented in Table 1 (very low = 1, ..., very high = 5). The overall reception of our intervention reveals potential for improvement but also shows that it is not

	present	fair	important	under-standable	likable	interesting	fitting	acceptable	Average
Bart	very low	medium	medium	very low	medium	very low	very low	medium	2.0
Kent	low	low	high	low	low	medium	medium	low	2.5
Krusty	high	low	high	low	low	high	medium	medium	3.0
Lenny	high	medium	medium	low	medium	medium	low	medium	2.9
Mel	high	high	low	medium	medium	medium	low	high	3.1
Millhouse	medium	medium	low	medium	medium	medium	medium	medium	2.9
Ned	(no feedback received)								
Reinier	low	low	very low	very low	very low	very low	very low	very low	1.3
Sanjay	medium	very low	medium	very low	high	low	medium	very low	2.3
Seymour	medium	very low	very low	high	medium	medium	high	medium	2.8
Average	2.89	2.33	2.56	2.11	2.67	2.56	2.44	2.56	2.5
$r_s(mail)$	0.53	-0.05	-0.23	0.36	0.58	0.27	0.41	0.13	.35
$r_s(Q_l^{end})$	0.28	0.04	-0.01	0.28	0.49	0.18	0.38	-0.10	.25
$r_s(Q_l^{start})$	0.20	-0.01	-0.04	0.20	0.52	0.06	0.28	-0.15	.08
$r_s(Q_t^{end})$	-0.50	-0.42	0.44	-0.67	-0.03	-0.39	-0.11	-0.41	-.53
$r_s(Q_t^{start})$	-0.40	-0.14	0.41	-0.13	-0.37	0.01	-0.02	0.06	-.18

Table 1: Likert-scaled opinions about the intervention and correlation with score ranks

rejected outright. Fitting (2.44), fair (2.33) and especially understandable (2.11) are the most problematic areas, while likable (2.67) and present (2.89) are less problematic.

Unless stated otherwise, all correlation coefficients r_s denote Spearman rank correlations. Most correlations are not significant, which is probably due to the small sample size. That does not necessarily mean that detected correlations do not exist but the chances are non-negligible. As chances increase with lower coefficients, only correlations that have at least a medium strength (around $r \approx 0.3$ in the classification of de Vaus (2002)) are discussed below.

The average personal opinions correlate weakly with quality at the end of the evaluation period. The relationship with ranks published through email ($r_s(mail) = .35$) is a bit stronger. There is a chance that this increased strength might be attributed to the effect that profiteering leads to a better reception, or that a better overall reception of reputation leads to more investing in quality. Yet the difference is not significant and could be coincidence.

There is a strong relationship between published ranks and the perceived presence of reputation scores ($r_s(mail) = .53$). One explanation could be that those who feel that reputation has a high presence are more interested in achieving high reputation scores and ranks themselves, and therefore write higher quality code. But the following reasoning suggests that it is the other way around; that being higher in the ranks leads to a stronger perceived presence of reputation. If presence leads to investing in quality, regardless of the actually published scores, then the correct ranks Q_l^{end} at the end of the trial should show a similarly strong or even stronger correlation $r_s(Q_l^{end}) = 0.28 \not> 0.53$. Our data does not support this. Instead, future designs should take into account that having a low quality rank leads to low perceived presence of reputation. Cognitive dissonance theory can explain this observation[2] (Festinger, 1957).

The perception of fairness seems unrelated to actual reputation ranking. Similarly, we found no relationship for acceptability. Those who understood well how reputation scores were computed had a slightly better chance to achieve higher ranks ($r_s(mail) = .36$).

Developers who achieve high reputation ranks have a substantial tendency to like the computation of reputation scores ($r_s(mail) = .58$). It may be that profiteers plainly enjoy receiving the benefit of reputation (we describe above that we do not see such evidence). But it may also be that developers who care for code quality — and who therefore have high reputation scores — welcome any tool that supports code quality.

[2] Humans want to always maintain a positive self-image. If they make observations, which are not congruent with positive self-image, then such observations are ignored.

The factors that are most strongly correlating with high reputation ranks are the liking of having reputation computed, the presence of reputation, and the fitting of reputation into the XP environment. The only negatively correlated factor is the perceived importance of reputation.

For work quantity, most correlations are the inverse of the correlations for quality, and are in many cases stronger. We found a major negative relationship between quantity and average opinion $r_s(Q_t^{end}) = -0.53$. A simple explanation is the medium negative relationship between quality and quantity (see Section 2.4). More than that, mass contributors have a fairness problem, think that it is unfair ($r_s(Q_t^{end}) = -0.42$), have problems understanding how scores are computed ($r_s(Q_t^{end}) = -.67$), and do not feel that reputation is present ($r_s(Q_t^{end}) = -0.50$), acceptable ($r_s(Q_t^{end}) = -0.41$) or interesting ($r_s(Q_t^{end}) = -0.39$). But reputation scores often mattered to them ($r_s(Q_t^{end}) = 0.44$).

5 Threats to validity

Our field study included only a few developers, which makes correlation results hardly statistically significant. Statistical significance (5% threshold) for $N = 9$ is reached at $r \approx 0.6$. While some results are almost significant, most of them need to be treated with care. The reason why results are not statistically significant is probably due to the sample size, not because of the strength of the correlation. If the team had been bigger, more of our correlations would have been significant. Still, many results reveal a moderate strength, and the team size of about ten developers is the normal size for an agile software project or sub-project.

While it is true that the environment of the field experiment presented here is artificial to some degree, it is not set up for the experiment specifically but, as part of teaching activities, tries to resemble a natural setting as closely as possible. For example, while the subject pool features students, it is graduate students with development experience working in the domain of their education (Harrison and List, 2004). It is natural that experimental realism comes at the cost of experimental control.

Therefore some environmental influences were sub-optimal for our study. For instance the experiment might have been to short. Perhaps more time would have been needed for CollabReview to unfold its full effect. Yet we could not influence the length of the project. Another sub-optimal factor is that developers did pair programming. But these are the normal difficulties of experiments that are not conducted under fully controlled laboratory conditions. Researchers have to live with what is there. At the same time, this adds to the realism of field studies. To ensure that our results can be generalized to other software projects, more and larger studies in industrial software projects have to be conducted.

6 Related work

The question whether a correlation coefficient of $r = 0.5$ is weak, moderate or strong is difficult to answer. The answer is to some extent relative. When humans are involved, most outcomes have many causes so that no two variables alone are likely to be very strongly related. Here a correlation of 0.30 might be regarded as relatively strong. For describing correlation strengths involving human factors, this paper follows the recommendations of de Vaus (2002).

An experiment where CollabReview was successfully deployed to improve contribution to a work group's wiki is described in Dencheva et al. (2011). The main differences compared to the earlier study are the (i) domain (source code vs. wiki), (ii) social team structure (short-time lab vs. long-term collaboration), (iii) rewards (money vs. display), and (iv) pair development resulting in a loss of responsibility fidelity (pair vs. solitary contributing). Hoisl et al. (2007) present a similar study in a wiki using a different reputation system. Singer and Schneider (2012) gave points for committing frequently to a revision repository, emailed the scores in a weekly digest, and thereby could successfully alter developer behavior.

Checkstyle is a well-known tool for finding potential coding style problems in source code (Smart, 2008). Several authors have used Checkstyle and static analysis for grading and assessment of student programming assignments (e.g. Edwards, 2003; Smith, 2005; Loveland, 2009). However, we do not know of instances where automated assessments are combined with fine-grained responsibility information to form reputation scores.

7 Lessons learned and future work

This section summarizes and recaps the lessons we learned from conducting the field experiment.

7.1 Lessons regarding the development environment

The presented experiment was the first field experiment with CollabReview in a programming environment. This encounter with reality has taught us several lessons. Firstly, CollabReview is not very well suited for pair programming because there is only one committer recorded in the revision control system while code is written by two developers. The social dynamics that are in effect between the changing pairs of developers, and which are not available to the reputation system, should not be underestimated. Also, the metrics were computed only for a part of the whole code base, which was assigned to the project. The analyzed code did not include Prolog and some Java parts. For example, it seems that Ned contributed only very little, which is probably wrong. This blurs contribution and responsibility mappings and probably leads to incorrect or at least skewed reputation scores. In turn, this reduces the understanding and perceptions of fairness in the team.

Extreme programming favors expressive naming of identifiers over comments. Much in line with this philosophy, project management had ambivalent opinions regarding comments and Javadoc during our field test. Some students perceived Javadoc as senseless. Management must clarify that there is no goal conflict, and that indeed naming and comments are important (cf. Raskin, 2005). Its full support for the employed measurements is necessary.

7.2 Lessons regarding the effectiveness of the reputation system

Classical management states that "you get what you measure". A metric might not measure the right things, leading to undesired behavior, but one should get what one measures (Hauser and Katz, 1998). Consequently, we should have gotten a lot of (potentially senseless) Javadoc comments. But we did not. The question is why?

Money seemed adequate for such a short, one-time project. Possibly winning the one prize was too risky or too much out of control for the individual (Hauser and Katz, 1998). Perhaps there should be more prizes. In previous work, we have been experimenting with other rewards as well (cf. Dencheva et al., 2011; Prause et al., 2010). For some developers, competition is an important motivator. For instance, Krusty mentioned competition as important factor and enjoyed it. According to our data, something with a high presence is needed. And in general, a better overall reception with regard to the different feelings leads to more investment in scores.

The reputation idea was mostly accepted. But developers expressed that acceptance depends on how scores are used. Especially developers who already care for quality, welcome reputation scores. However, there was no indication that it is profiteering which leads to a better reception.

Major contributors have a stronger influence on total code quality because they contribute more code. It is therefore of high importance to reach them with the intervention. But acceptance of the intervention among them is especially low. They might feel treated unfair because they give the software a lot of functionality, and might feel that this contribution is not valued enough. For a minor contributor (like Millhouse) it is much easier to achieve very high or very low reputation scores. It must be explained to major contributors that the purpose of the intervention is not assessing their performance. Instead, the goal is to improve the quality of the project, and here they have an even higher responsibility than minor contributors.

Low ranks result in higher perceived importance of reputation scores. This is a danger for team motivation! Even more so, as while being second is good, it "feels bad to be second not first". Additionally, low ranks lead to low perceived presence, so especially poor-scoring developers need more presence of reputation and must understand that scores measure something that is important to them. Contrariwise, major contributors have the worst feelings about quality scores. But it is them who are very important because they influence the code base the most.

Create understanding of how scores are computed! Understanding has one of the highest correlations with the email rank. This is congruent with literature (cf. Hauser and Katz, 1998). Perhaps developers would just have needed more time to get acquainted to CollabReview. But also in the short time, the understanding of reputation could have been improved by providing more up-front training, and more immediate and elaborate feedback on developer actions. An alternative is to simplify the algorithm for measuring quality to make it easier to understand.

We are still convinced that reputation gaming will work when the experimental setup is amended in such a way that the relationship between investing in code quality and a desirable reward is firmer as, e.g. in the VIE theory by Vroom (1964); that means that

- developers need a better understanding of how to affect their score,
- responsibility assignments are less fuzzy by avoiding pair programming and scores are more under control of the individual (i.e. "Expectancy": effort in code ⇒ better score)
- a high score leads to a prize with a higher probability, e.g. by not only rewarding the first place (i.e. "Instrumentality": better score ⇒ prize), and
- the not necessarily monetary prize is worthwhile to achieve (i.e. "Valence", the prize).

8 Conclusion

We have conducted a field experiment with the CollabReview reputation system for source code in an agile software project. The goal was to affect ongoing development in such way that developers write more Javadoc. The reputation scores were not meant as a performance measure of a developer's productivity but as a compensation for the "endured pain" of doing what developers do not like to do: to write documentation (Selic, 2009).

The CollabReview reputation system was brought into the project after an initial phase where comparison data was collected. Developers started receiving a daily digest of their reputation scores, and a prize was announced for the winner. While the intervention probably had had some effect, no measurable quality improvement occurred. However, the experiment does not evidence that a reputation system is unsuitable for improving code quality. Instead, it shows that integration into development is difficult, and must be done right. CollabReview was previously deployed successfully in a wiki, so it was a surprise that this experiment "failed".

The major contribution of our work is therefore the lessons that we learned: Measurements must be implemented carefully to measure the right things, and to not endanger team spirit. A low rank leads to low perceived presence but high importance, i.e. weak positive but strong negative effects. Already the second place leads to a bad perception. Furthermore, developers who care for quality welcome a tool like CollabReview that supports them. Yet there is no indication that it is the profiteering that leads to a better overall reception. But a better overall perception leads to more investing in quality. Of all factors, understanding seems to be the most essential factor for performance improvement. Especially mass contributors had bad feelings about scores but they mattered to them the most, creating feelings of unfairness. However, there was few rejection and acceptance mostly depends on how scores are used.

With our work we guide the design of similar systems, and further the understanding of the social dynamics that reputation systems cause in software projects. We also found that developers with high reputation scores (resulting from Javadoc) are likely to write commit messages. The reason is perhaps a documentation-affirming character trait. This finding suggests that reputation scores do not capture an arbitrary value but are a sound measure.

In the future, we want to repeat the field experiment in a more suitable environment without pair programming, and carefully consider the lessons learned. Ideally, the later experiment has more participants, and can run for a longer duration. At the same time, we want to write a daily log of project events to be able to relate events to observations of reputation scores.

Acknowledgments

We are grateful to the teachers of the XP Lab 2011 — Armin B. Cremers, Daniel Speicher, Paul Imhoff — who kindly allowed us to conduct our studies, and the participants for being our guinea pigs. Furthermore, we like to thank Markus Eisenhauer, Uwe Kirschenmann, René Reiners, Gabriel Bonerewitz and the PPIG reviewers for their advice on the experimental setup, designing the interview, their opinions on the obtained results, and the valuable hints for improvement.

Bibliography

de Vaus, D. A. (2002). *Surveys in Social Research.* Routledge, fifth edition.

Dencheva, S., Prause, C. R., and Prinz, W. (2011). Dynamic self-moderation in a corporate wiki to improve participation and contribution quality. ECSCW, pages 1–20. Springer.

Dubochet, G. (2009). Computer code as a medium for human communication: Are programming languages improving? In 21st *Annual Workshop of PPIG*, PPIG.

Edwards, S. H. (2003). Teaching software testing: Automatic grading meets test-first coding. In *OOPLSA Companion*, pages 318–319. ACM.

Festinger, L. (1957). *A Theory of Cognitive Dissonance.* Stanford University Press.

Harrison, G. W. and List, J. A. (2004). Field experiments. *J. of Econ. Lit.*, XLII:1009–1055.

Hauser, J. and Katz, G. (1998). Metrics: You are what you measure! *EMJ*, 16(5):517–528.

Hoisl, B., Aigner, W., and Miksch, S. (2007). Social rewarding in wiki systems — motivating the community. In *Online Communities and Social Computing.* Springer.

ISO 9126-1 (2001). ISO 9126-1: Software engineering - product quality: Part 1: Quality model.

Jøsang, A., Ismail, R., and Boyd, C. (2007). A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644.

King, P., Naughton, DeMoney, Kanerva, Walrath, Hommel, et al. (1997). Java code conventions.

Loveland, S. (2009). Using open source tools to prevent write-only code. In *Conference on Information Technology: New Generations.* IEEE CS.

Prause, C. R. (2011). Reputation-based self-management of software process artifact quality in consortium research projects. In *ESEC/FSE*, pages 380–384. ACM Press.

Prause, C. R. and Apelt, S. (2008). An approach for continuous inspection of source code. In 6th *International Workshop on Software quality*, WoSQ, New York, NY, USA. ACM.

Prause, C. R. and Durdik, Z. (2012). Architectural design and documentation: Waste in agile development? In *International Conference on Software and System Process.* IEEE CS.

Prause, C. R. and Eisenhauer, M. (2012). First results from an investigation into the validity of developer reputation derived from wiki articles and source code. In *CHASE.* ACM.

Prause, C. R., Reiners, R., Dencheva, S., and Zimmermann, A. (2010). Incentives for maintaining high-quality source code. In *Psychology of Programming Interest Group Work-in-Progress.*

Raskin, J. (2005). Comments are more important than code. *ACM Queue*, 3(2):64–62 (sic!).

Seibel, P. (2009). *Coders at Work: Reflections on the Craft of Programming.* Apress.

Selic, B. (2009). Agile documentation, anyone? *IEEE Software*, 26(6):11–12.

Singer, L. and Schneider, K. (2012). It was a bit of a race: Gamification of version control. In 2nd *International Workshop on Games and Software Engineering.*

Smart, J. F. (2008). *Java Power Tools.* O'Reilly Media, first edition.

Smith, D. (2005). Gide: An integrated development environment focusing on agile programming methodologies and student feedback. In *WCCCE.*

Spinellis, D. (2011). elyts edoc. *IEEE Software*, 28:104–103.

Vroom, V. H. (1964). *Work and Motivation.* Wiley.

PPIG 2012 Doctoral Consortium

Teaching Novices Programming Using a Robot Simulator

Louis Major
School of Computing and Mathematics
Keele University, UK
l.major@keele.ac.uk

This research investigates the effectiveness of using simulated robots as tools to teach introductory programming. It has been influenced by the results of a Mapping Study and Systematic Literature Review which indicated that such a study would be valuable. A robot simulator has been developed after reviewing educational software guidelines. This has been implemented in a number of programming workshops. A range of participants have been involved including novice programmers in addition to trainee and in-service high school teachers. By its conclusion, over 95 participants will have taken part. The case study methodology has been used and this will help to ensure reliable and rigorous empirical research is undertaken. This project will contribute to knowledge by addressing the findings of the SLR, specifically the need to investigate the use of simulated robots as tools to teach programming.

Novel Interaction Designs for a New Novice Programming Editor

Fraser McKay
School of Computing
University of Kent
fm98@kent.ac.uk

This project concerns the design of interaction and presentation elements in a new programming editor, designed for learner programmers. Its aim is to avoid both the viscosity that is often part of visual programs, and the error-proneness of traditional text-based programming languages like Java. This extended abstract presents some prototype interactions, and our observations about them thus far. A stand-alone editor using these interactions has been presented elsewhere. CogTool models have also been used to compare the new editing style to those found in existing systems. This paper also briefly describes new usability heuristics that we have proposed, and begun to evaluate, for systems in this specific domain. Our evaluations of existing systems (including Scratch, Alice, Greenfoot, and others) have led to the development of these heuristics.

An empirical study on program comprehension using eye tracking and think aloud

Teresa Busjahn
Department of Computer Science
Freie Universität Berlin
teresa.busjahn@fu-berlin.de

The main focus of my thesis is on code reading and program comprehension. I use a combination of eye tracking and retrospective think aloud in order to gain insight into cognitive processes during code reading and understanding.

Programming, Professionalism and Pedagogy

Melanie Coles
Bournemouth University
mcoles@bournemouth.ac.uk

This research aims to explore professionalism in programming, what it means to be a programming professional and what skills and attributes this should include. The research will explore what a variety of stakeholders consider to be the professional attributes of a programmer, covering both documentary evidence and opinion of stakeholders. The documentary evidence will be investigated to identify common themes and trends, this will then be followed up by detailed interviews to gather further data and develop the themes from a variety of stakeholders. Then the opinion of a broader range of stakeholders on these themes will be gathered using a survey/questionnaire. The findings of this research will be used to develop a model of professional programming, possibly developing a model for different domains of programming. This model can then be evaluated by stakeholders and applied in a variety of domains to assess its usefulness.

The design and implementation of a notional machine for teaching introductory programming

Michael Berry
School of Computing
University of Kent
mjrb5@kent.ac.uk

Several studies have shown that students find programming hard to grasp, with many computing courses showing significant drop out and failure rates. This work aims to help with this problem by contributing a new notional machine that is helpful for the teaching of introductory programming, and is currently in the start of its second year. An evaluation of existing notional machines and their corresponding implementations has been performed thus far, and a prototype is currently being developed in BlueJ that should form the basis of the implementation of the new notional machine. The intended outcome of this project is to contribute a new notional machine which students should find helpful for learning object oriented programming, specially in Java.

Eliciting peer definitions of a 'good' programmer

Gail Ollis
1st year PhD researcher
School of Design, Engineering and Computing
Bournemouth University
gollis@bournemouth.ac.uk

This research, currently in its first year, investigates individual differences between experienced programmers varying in aptitude. The topic of aptitude calls for a clear definition of 'good programmer', a concept for which multiple and potentially conflicting criteria exist. In this research, the chosen criteria will be those identified by experienced software developers as affecting their productivity when working with existing code. The tasks of fixing, maintaining or adapting a program represent a large part of software development effort, so the effect of implementation decisions by the original programmer can be significant. A later phase of the research will explore the role of personality in those decisions. The first phase, though, is to establish characteristic behaviours which are commonly considered a help or a hindrance by peers. Discussion is invited on the proposed methods for eliciting their opinions.

Supporting developer to apply Trust Management

Mark Vinkovits
User Centred Ubiquitous Computing
Fraunhofer FIT
mark.vinkovits@fit.fraunhofer.de

Software security has become a demanded functionality in the last decade. As more and more assets are stored digitally and organizations enter e-commerce, security has become an unavoidable issue. Although the obvious need for proper software security, managers and developers ignore the proper handling of the problem and apply best effort solutions based on their available expertise. This work aims to understand the reason for the neglect, adapt the amount of security to the real needs and support non-security specialists to integrate software security into their products. The area of interest is Trust Management, as one of the promising solutions for distributed systems, to which we apply model-driven security as way to achieve the desired usability.

Dual Eye Tracking for Teaching Debugging

Kshitij Sharma
CRAFT, ´ Ecole Polytechnique F´ed´erale de Lausanne
kshitij.sharma@epfl.ch

We study Dual Eye Tracking and Collaborative Pair Programming with two major goals. The first goal is to consolidate and deepen our understanding of the mechanisms that make collaborative interaction productive and to model how gaze reflects to activity that is pursued by the collaborators and their levels of expertise. The second goal of the project builds on our previous findings, and consists of developing and testing a gaze-awareness tool that informs collaborators about the convergence of their gaze. Such an indication should result in a lower grounding effort since speakers can “see” whether their partners are following them and better tune the effort that is required to reach shared understanding.

Getting at Ephemeral Flaws

Tamara Lopez
Centre for Research in Computing, The Open University
t.lopez@open.ac.uk

Software rarely works as intended when it is initially written. Things go wrong, and developers are commonly understood to form theories and strategies to deal with them. Much of this knowledge relates to ephemeral flaws rather than reported bugs, and is not captured in the software record. As a result, these flaws and understanding about them are neglected in software engineering research. In this paper we describe a study designed to elicit stories from software developers about problems they encounter in their daily work. We also offer preliminary thoughts about the utility of retrospective interviewing in getting at information about ephemeral flaws.

Studying the utility of Natural Language Descriptions as a support for novices in the early stage of learning programming

Edgar Cambranes
Human-Centred Technology Group
School of Engineering and Informatics.
University of Sussex.
Brighton, United Kingdom.
E.Cambranes-Martinez@sussex.ac.uk

Many novice programmers find the programming process problematic, in part because they have to use unfamiliar elements to create programs. Some environments developed for novice programmers, such as Scratch or Alice, have considered, for example, motivation or error-free syntax as key points to support programmers. However, these environments still fail to express programs in a way familiar to novice programmers, thus the user cannot easily verify whether the solution program matches with their intention. This research proposes to explore the usefulness of providing a second representation (in addition to a flowchart representation) to support a visual language designed for novice programmers. In particular, the second representation uses Natural Language to describe the current solution. The aim of the research is to assess the beneficial effects (if any) of a second representation on the comprehension, creation and debugging of programs. Specifically, the research will try to answer to what extent does the second representation affect these different aspects of the programming process? This paper describes an experiment that has compared the use of natural language as a secondary representation to a flowchart language with the use of pseudo-code as a secondary representation to the same flowchart language.

The Development of an Intelligent Simulation Framework to Optimize the Production, Design and Development in the Cameroon Development Corporation (C.D.C) Palm Oil Industry

Cosmas A. Fonche
Faculty of Life Sciences and Computing
School of Computing
London Metropolitan University
Caf0233@londonmet.ac.uk

This project focuses on the use of simulation to optimize production design and development within industries. Phase one of the research work is to define a process of imitating the operations of a real world process or system over time, view and evaluate the systems drawbacks and bottlenecks and finding the best input variable or solution, from among all possibilities. The objective of simulation experiments is to understand the behaviour of the system and to test possible changes. The purpose of this research is to explore the influence of simulation and develop an intelligent framework to optimize the production design and development in the Cameroon Development Corporation (C.D.C) Palm Oil Industry.

Investigation Leading to Behaviour-Based Hybrid Intrusion Detection System for Mobile Devices

Khurram Majeed
School of Computing
London Metropolitan University
k.majeed@londonmet.ac.uk

Smartphones nowadays have become immensely popular because they provide All-In-One expediency by integrating traditional mobile phones with hand-held computing devices making them more open and general purpose. However this flexibility leaves the Smartphones prone to attacks by malicious hackers. These malware not only poses a threat to mobile system data confidentiality, availability and integrity but can result in unwanted billing, depletion of battery power and denial-of-service (hereafter DOS) attack by generating malicious traffic hence seriously crippling the mobile network and service capacity. Current Smartphones malware detection and prevention techniques are limited to Signature-Based antivirus scanners (hereafter SIDS). These can efficiently detect malware with a known signature, but they have serious shortcomings with new and unknown malware creating a window of opportunity for attackers. A framework for Behaviour-Based Hybrid Intrusion Detection System is proposed to circumvent these shortcomings. This framework aims to provide protection against physical misuse using Machine Learning technique and detection of malicious applications using Knowledge Based Temporal Abstraction method. This research will be among the first to combine these two methods. Being platform independent is another novelty of the framework. A prototype has been partially implemented on Google Android and tested on emulators. Further validation will be performed on Smartphones to benchmark this framework.