HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments

Volume 4, Number 1, May 2008

SPECIAL ISSUE ON PSYCHOLOGY OF PROGRAMMIING

Jorma Sajaniemi, Guest Editor

Pertti Saariluoma, Editor-in-Chief

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments Volume 4, Number 1, May 2008

Contents

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments

Human Technology is an interdisciplinary, scholarly journal that presents innovative, peer-reviewed articles exploring the issues and challenges surrounding human-technology interaction and the human role in all areas of our ICT-infused societies.

Human Technology is published by the Agora Center, University of Jyväskylä and distributed without a charge online.

ISSN: 1795-6889
Submissions and contact: humantechnology@jyu.fi
Managing Editor: Barbara Crawford

www.humantechnology.jyu.fi

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 1–3

From the Editor in Chief

THE PROBLEMS OF PROFESSIONALS

Pertti Saariluoma
Cognitive Science, Department of Computer Science and Information Systems
University of Jyväskylä, Finland

When we discuss interaction and communication technology (ICT) usability, the images of ordinary users facing difficulties in getting things to work come easily to mind. People who struggle to use digital applications or find mobile services, or feel lost or frustrated when trying to use all of the features of a remote controller seem to form the very stereotype of users that interaction research should help. How my auntie, elderly neighbor, or disabled brother could survive in an ICT-infused world is a recognized problem today, although not that long ago, their problems were not a priority. The main focus of the research had been on early middle-aged families with Western backgrounds (Czaja, 1997; Newell & Gregor, 1997).

Of course, concentration on the "ordinary" people is acceptable on several grounds. Consumers form the widest audience and markets for new ICT products. They also may have the least amount of time for learning new environments and gadgets. Finally, they often possess the lowest level of computing skills or technical know-how. This is why emphasis on the usability of applications is understandably of great importance among interaction researchers. However, it would be a mistake to think that usability should be directed only toward solving the challenges of ordinary people.

The interaction challenges of the professional are a much less clearly recognized problem than the concerns for everyday consumer interaction. One might think that technology professionals easily understand what other professionals need and that professionals in general do not make similar mistakes in using technologies or solving interaction problems via ICTs as do to individuals with everyday technology use. Moreover, professionals appear to have the time and the skills, are often within an appropriate age group to be familiar with various types of technologies, have expertise in what they should do with computing or other ICT devices, and are usually willing to invest the time and energy to learn new devices or applications. They are generally educated and experienced enough so that they can be systematically trained to use new software or technologies, and it is easier for and more typical of them to seek and obtain the needed support when they confront interaction problems.

URN:NBNfi:jyu-200804151348

While many of these assumptions may be true, nevertheless these arguments miss one important point: The tasks of professionals via ICTs are far more complex and critical than the tasks in everyday computing and ICT use.

Take, for instance, surgeons or other medical professionals, for whom losing time or information as a consequence of poorly constructed interaction systems is safety critical. Similarly, officers at the helm of a huge ship or, perhaps more importantly, workers at a nuclear power plant, still must be thoroughly familiar with—and have easy access to and use of—all essential, even if even rarely used, features of their complex, contemporary technologies. Because professional ICT-facilitated interaction is constantly increasing and becoming more ubiquitous, it is essential that interaction researchers give specific attention to how experts use technology. While human factors researchers have done much work in this area, much still remains to be done. For example, when it can take a business professional untold hours over the course of a year to complete and submit travel expense documents, or when someone is expected to read a hundred-page-long users' instruction manual in order to store a couple of numbers in a computing program, it is easy to see that this area is underestimated and underresearched. These interaction realities are complex issues.

A classic example of this interaction complexity challenge is computer programming. This complex task requires immense cognitive energy and skills. This reality has not been lost on a community of researchers who initiated some of the earliest attempts to make technologies simpler for users, and the professionals who create the technologies. The first programming languages were designed to help programmers remember the code. Similarly programming paradigms, such as structured programming or object-oriented programming, were intended to make programs easier for the programmers to comprehend and remember. Thus psychology was employed to help improve the work conditions for professional programmers, and thus opening a new field of research: the psychology of programming. The very foundation of this work points to the need to observe and address the challenges of professionals—whether they are computer programmers, or medical or business professionals. The outcomes of the field of computer programming reflect the emphasis on good expert-computer interaction. Furthermore, the rich tradition of this field offers a large body of knowledge that can be transferred to other professional fields.

We are pleased to have a special issue on the psychology of programming in *Human Technology* because few areas of professional interaction research have equally developed the practices of discourse, analysis, and developmental design. The papers of this issue reflect not only concerns about computer programming, but also topics that can provide the foundation for exploring many other aspects of expert-technology interaction. From this research foundation, knowledge about the psychology of expert interaction with technology can feed ongoing expert-technology research in more diverse fields, such as medicine, education, aeronautics, business, energy, and transportation.

REFERENCES

Czaja, S. (1997). Computer technology and the older adult. In M. Helander, T. Landauer, & P. Prabhu (Eds.), Handbook of human–computer interaction (pp. 797–812). Amsterdam: Elsevier.

Newell, A. F., & Gregor, P. (1997). Human computer interfaces for people with disabilities. Handbook of human–computer interaction (pp. 813–824). Amsterdam: Elsevier.

All correspondence should be addressed to:
Pertti Saariluoma
University of Jyväskylä
Cognitive Science, Department of Computer Science and Information Systems
P.O. Box 35
FIN-40014 University of Jyväskylä, FINLAND
psa@it.jyu.fi

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 4–8

Guest Editor's Introduction

PSYCHOLOGY OF PROGRAMMING: LOOKING INTO PROGRAMMERS' HEADS

Jorma Sajaniemi
Department of Computer Science and Statistics
University of Joensuu, Finland

Psychology of programming (PoP) is an interdisciplinary area that covers research into computer programmers' cognition; tools and methods for programming related activities; and programming education. The origins of PoP date back to late 1970s and early 1980s, when researchers realized that programming tools and technologies should not be evaluated based on their computational power only, but also on their usability from the human point of view, that is, based on their cognitive effects. The hope of such a new approach was that programmers would make fewer errors, produce better software, and work more efficiently. In the first Workshop on Empirical Studies of Programmers, Ben Shneiderman listed "several important destinations for researchers: refining the use of current languages, improving present and future languages, developing special purpose languages, and improving tools and methods" (Shneiderman, 1986, p. 1). During the past two decades, the flow of new languages, tools, and methods has increased rapidly, the scope of programming work has expanded, and research interests have extended to cover group activities. Yet the main goal of PoP—to assist programmers through the benefits of cognitive research—has remained.

The PoP research community consists of cognitive psychologists and computer scientists. The main motivation for computer scientists is the improvement of current tools and the development of new ones, as well as the discovery of general principles concerning humans in the context of programming tasks. Psychologists are interested in new theories of human cognition applicable in other domains too. For them programming—a highly complicated task—provides good opportunities to study high-level cognitive processes in a complex setting. This dual character of PoP manifests itself also in the skills required from researchers: a good knowledge of both cognitive psychology and programming or, better still, psychology, social sciences, and software engineering.

On the other hand, PoP research results are not necessarily limited to the programming domain, but can be applied in other domains that involve design activities in a formal environment. As an example, consider cognitive dimensions (CDs), which were introduced by Green (1989) to describe, compare and control how programming language features affect program design strategies. The dimension role-expressiveness, for example, relates to how well a piece of program code (e.g., "+") reveals its meaning without a need to study the context

URN:NBNfi:jyu-200804151349

of the piece (addition, string catenation, etc.). Later, CDs were developed further and applied to many types of cognitive artifacts, such as educational theorem provers (Kadoda, Stone, & Diaper, 1999), prototyping techniques (Dearden, Siddiqi, & Naghsh, 2003), and music notations (Blackwell & Green, 2003).

Even though the area of PoP seems to be quite narrow—computer programming—it covers a large variety of phenomena, from novices' problems to experts' tacit knowledge, from program design to testing and maintenance, and from short individual programs to huge software systems. Consequently, research methods vary as well. Most often, research methods have been adopted from cognitive psychology (e.g., controlled experiments run in laboratory settings) or social sciences (e.g., field studies with qualitative analysis techniques), but it seems that in many subareas appropriate research methods are yet to be discovered. As many researchers are also computer science educators, they have instant access to novices and, therefore, studies on novices' problems and programming education are frequent. A new source of research materials is provided by various open source communities that make their program code, change logs, and discussions among program developers freely available on the net. These materials represent expert programming in state-of-the-art contexts.

During the past two decades, two important workshop series have been fully devoted to PoP: the Workshop on Empirical Studies of Programmers (ESP), based primarily in the USA, and the Psychology of Programming Interest Group Workshop (PPIG), having a European character. The first ESP workshop was held in 1986 in Washington, DC, the eighth and last one in 1999. Later, this workshop series was incorporated into the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), which, however, has a broader scope than pure PoP and includes implementation aspects and the like. The European conference series, PPIG, started in 1989, and continues to be organized annually. It is more informal in nature than ESP; in addition to fully developed research papers, PPIG proceedings include position papers and suggestions for individual studies. Many of the best papers have later been published in more formal conferences and journals.

The organization behind PPIG workshops, the Psychology of Programming Interest Group, was established in 1987 and—just like the workshop—is informal in nature. For instance, there is no formal committee: Decisions are discussed in an open business meeting held during every workshop. The interest group publishes an electronic newsletter and hosts two mailing lists, a low-volume announcements list plus another list for discussions. In essence, the interest group is an informal collection of people who are enthusiastic about psychological aspects of programming and software engineering[1].

The latest PPIG workshop was held in Joensuu, Finland in July 2007. The scientific program consisted of four half- or full-day tutorials, a doctoral consortium, two keynote addresses, 18 technical presentations, and two discussion sessions. All paper submissions were reviewed by at least two—usually three—anonymous reviewers and papers were accepted in two categories: Full Papers and Work in Progress Reports, as decided by the Program Committee. This special issue of *Human Technology* contains five of those papers, selected based on the reviewers' statements. The papers were re-reviewed and improved for publication in this journal. These papers demonstrate the variability in themes and research methodologies of PPIG workshops.

The first two papers deal with research methodology. In the article "A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming," Stephan Salinger, Laura Plonka, and Lutz Prechelt consider the analysis of

rich video data that is typical for programming protocols. They have used grounded theory (Strauss & Corbin, 1990), in which the whole coding is based totally on protocol data, and developed a specific coding scheme to be used in the context of pair programming. The article provides guidance for the use of grounded theory in the analysis of rich protocol data when the purpose of a study is to understand cognitive phenomena within a design process. The principles described in the paper apply to domains outside programming, as well.

Rozilawati Razali, Colin Snook, Michael Poppleton, and Paul Garrat have used two methods to evaluate the usability of a semiformal notation that combines UML (Object Management Group, 2007) with B (Abrial, 1996), the latter being a formal notation for describing semantics. The evaluation methods include CDs and the results were analyzed using grounded theory. This paper, "Usability Assessment of a UML-based Formal Modeling Method Using a Cognitive Dimensions Framework," thus demonstrates how one can use several research methods for the usability analysis of tools within formal domains that involve design activities.

The next two papers concentrate on specific details within programming. Sue Jones and Gary Burnett tackle a popular problem: how to predict students' success in learning programming. Earlier work on this area has looked at correlation between programming success and some other property, for example, field dependence (e.g., Mancy & Reid, 2004), inclination to systematic behavior (e.g., Dehnadi, 2006), or self-efficacy (e.g., Wiedenbeck, LaBelle, & Kain, 2004). Jones and Burnett study spatial ability and find a positive correlation between mental rotation ability and programming success in their paper "Spatial Ability and Learning to Program."

Juha Sorva looks at variable-oriented programming paradigm (Sajaniemi & Niemeläinen, 1989) and combines it with the notion of roles of variables (Sajaniemi, 2002). This results in a data-flow description of programs that explicitly classifies variables using a fixed set of categories found in expert programmers' tacit knowledge (Sajaniemi & Navarro Prieto, 2005). The article, "A Roles-Based Approach to Variable-Oriented Programming," also demonstrates how the new notation can be used for mental exercises even without a fully functional implementation.

The final paper, "From Procedures to Objects: A Research Agenda for the Psychology of Object-Oriented Programming Education" by Jorma Sajaniemi and Marja Kuittinen, presents an overview of PoP research in novice education and debates whether existing research literature, which deals mostly with procedural programming, can be applied to current educational practice that is based on object-oriented programming (de Raadt, Watson, & Toleman, 2002). The authors point out fundamental differences that make the use of existing research results in the current context dubious and suggest areas that should be studied if programming education is to be based on research results rather than intuition.

The five papers included in this special issue of *Human Technology* represent studies in research methodology and in small scale programming. Programming in the large, that is, production of complex software systems, is not represented in this set. The reason is simple: There were very few papers on that area in the 2007 PPIG workshop. This is also typical for PoP research in general. Research into the construction of large systems, although highly important, is very expensive and industry partners willing to use their time for such research are hard to find.

There is still a long way to go before PoP can provide an extensive picture of programming and software engineering in general.

ENDNOTE

1. For more information on the Psychology of Programming Interest Group, see http://www.ppig.org

REFERENCES

Abrial, J. R. (1996). *The B-Method: Assigning programs to meanings*. Cambridge, UK: Cambridge University Press.

Blackwell, A., & Green, T. (2003). Notational systems: The cognitive dimensions of notations framework. In J. M. Carroll (Ed.), *HCI models, theories, and frameworks: Toward a multidisciplinary science* (pp. 103–133). San Francisco: Morgan Kaufmann Publishers.

Dearden, A., Siddiqi, J., & Naghsh, A. (2003, April). *Using cognitive dimensions to compare prototyping techniques*. Paper presented at the 15th Annual Workshop of the Psychology of Programming Interest Group, Keele, UK.

Dehnadi, S. (2006). Testing programming aptitude. In P. Romero, J. Good, E. A. Chaparro, & S. Bryant (Eds.), *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group* (PPIG '06; pp. 22–37). Brighton, UK: University of Sussex.

de Raadt, M., Watson, R., & Toleman, M. (2002). Language trends in introductory programming courses. In E. Cohen & E. Boyd (Eds.), *Proceedings of Informing Science and IT Education Conference* (InSITE '02; pp. 329–337). Santa Rosa, CA, USA: Informing Science Institute.

Green, T. R. G. (1989). Cognitive dimensions of notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V* (pp. 443–460). Cambridge, UK: Cambridge University Press.

Kadoda, G., Stone, R., & Diaper, D. (1999, January). *Desirable features of educational theorem provers: A cognitive dimensions viewpoint*. Paper presented at the 11th Annual Workshop of the Psychology of Programming Interest Group, Leeds, UK.

Mancy, R., & Reid, N. (2004). Aspects of cognitive style and programming. In E. Dunican & T. Green (Eds.), *Proceedings of the Sixteenth Annual Workshop of the Psychology of Programming Interest Group* (PPIG '04; pp. 1–9). Carlow, Ireland: Institute of Technology.

Object Management Group (2007). *Introduction to OMG's Unified Modeling Language (UML)*. Retrieved April 11, 2008, from http://www.omg.org/gettingstarted/what_is_uml.htm

Sajaniemi, J. (2002). Visualizing roles of variables to novice programmers. In J. Kuljis, L. Baldwin, & R. Scoble (Eds.), *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group* (PPIG '02; pp. 111–127). Uxbridge, UK: Brunel University.

Sajaniemi, J., & Navarro Prieto, R. (2005). Roles of variables in experts' programming knowledge. In P. Romero, J. Good, S. Bryant, & E. A. Chaparro (Eds.), *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group* (pp. 145–159). Brighton, UK: University of Sussex.

Sajaniemi, J., & Niemeläinen, A. (1989). Program editing based on variable plans: A cognitive approach to program manipulation. In *Proceedings of the Third International Conference on Human-Computer Interaction on Designing and Using Human-Computer Interfaces and Knowledge Based Systems* (2nd ed.; pp. 66–73). New York: Elsevier Science Inc.

Shneiderman, B. (1986). Empirical studies of programmers: The territory, paths, and destinations. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 1–12). Norwood, NJ, USA: Ablex Publishing Co.

Strauss, A., & Corbin, J. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. London: Sage Publications, Inc.

Wiedenbeck, S., LaBelle, D., & Kain, V. N. R. (2004). Factors affecting course outcomes in introductory programming. In E. Dunican & T. Green (Eds.), *Proceedings of the Sixteenth Annual Workshop of the Psychology of Programming Interest Group* (PPIG '04; pp. 97–110). Carlow, Ireland: Institute of Technology.

Author's Note

I am grateful to Pablo Romero who organized the reviewing process of the paper that I have coauthored.

All correspondence should be addressed to:
Jorma Sajaniemi
University of Joensuu
P.O.Box 111
FI-80101 Joensuu
Finland
saja@cs.joensuu.fi

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 9–25

A CODING SCHEME DEVELOPMENT METHODOLOGY USING GROUNDED THEORY FOR QUALITATIVE ANALYSIS OF PAIR PROGRAMMING

Stephan Salinger
Institut für Informatik
Freie Universität Berlin
Germany

Laura Plonka
Institut für Informatik
Freie Universität Berlin
Germany

Lutz Prechelt
Institut für Informatik
Freie Universität Berlin
Germany

Abstract: *A number of quantitative studies of pair programming (the practice of two programmers working together using just one computer) have partially conflicting results. Qualitative studies are needed to explain what is really going on. We support such studies by taking a grounded theory (GT) approach for deriving a coding scheme for the objective conceptual description of specific pair programming sessions independent of a particular research goal. The present article explains why our initial attempts at using GT failed and describes how to avoid these difficulties by a predetermined perspective on the data, concept naming rules, an analysis results metamodel, and pair coding. These practices may be helpful in all GT situations, particularly those involving very rich data such as video data. We illustrate the operation and usefulness of these practices by real examples derived from our coding work and present a few preliminary hypotheses regarding pair programming that have surfaced.*

Keywords: *pair programming, grounded theory, coding scheme development, qualitative data analysis, video data.*

INTRODUCTION

During the last few years, pair programming, as it is known from extreme programming (Beck, 2004), has been the subject of many empirical investigations. This research focused mainly on the measurement of bottom-line pair programming effects, whereas the underlying process of pair programming has been regarded as a kind of black box, the output of which is analyzed quantitatively with respect to its performance, error rate, programmer satisfaction, and so forth.

URN:NBNfi:jyu-200804151350

Unfortunately, the results of this research are often contradictory. For instance, regarding total effort (measured in person-hours of developers' work time), Williams (2001) found that pair programming results in a 15% increase compared to solo programming, Lui and Chan (2003) found 21%, and Nawrocki, Jasiński, Olek, and Lange (2005) found 48%. Most likely these differences are caused by differences in moderator variables, such as programmer and pair experience, type of task, and so on, but we do not know the complete set of relevant moderator variables nor the nature and mechanism of their influence.

Our goal as software engineering researchers is to understand pair programming in such a way that we can advise practitioners how to use it most efficiently. We propose that the only way to obtain such understanding is to understand the mechanisms at work in the actual pair programming process. Obviously, this understanding must first be gained in qualitative form before we can start quantifying and, since we do not know much yet, the investigation has to start in an exploratory fashion.

We have started such an investigation based on the grounded theory (GT) methodology (Strauss & Corbin, 1990) and working from rich sets of data (full-length audio, programmer video, and screen video of pair programming sessions). The current article presents a number of important methodological insights gained during this research and a few initial results. Its contributions are the following:

- a description of stumbling blocks for a GT-based analysis in this area;
- a set of practices that extend the plain GT method and help overcome obstacles;
- a sketch of a pair programming process coding scheme.

In subsequent research, the coding scheme is intended to form the basis for more detailed conceptual descriptions of the pair programming process. It also should support the proposition of hypotheses and theory construction.

We will first give a short introduction to GT and describe the nature and origin of our raw data. The heart of the article describes how and why plain traditional GT does not work well under these constraints and which practices help it work better. Thereafter we will present the application of the modified GT process and a few of its initial results, namely excerpts of a coding scheme for describing the activities occurring during pair programming. We close by outlining related works and offering a summary and outlook. This article is an improved and slightly extended version of Salinger, Plonka and Prechelt (2007) and focuses on research method, not on research results. The results primarily serve to illustrate the method.

THE GROUNDED THEORY METHODOLOGY

Selecting Among Qualitative Research Methods

We have already argued why we believe that it is time to study pair programming in an exploratory manner. We want to avoid posing specific hypotheses and generally make as few assumptions as possible. Using predefined coding schemes (see Hughes & Parkes, 2003, for a list) implies making such assumptions and hence should be avoided. Considerations like these quite naturally lead to using GT as the research method, because GT is an approach that makes the fewest number of assumptions.

Alternative methods, such as protocol analysis (Ericsson & Simon, 1993) or verbal analysis (Chi, 1997), appear less suitable because they start from at least partially predefined coding schemes or theoretical models. They are also more specialized than appropriate: They were designed for investigating cognitive processes.

Verbal analysis aims at the ability to quantify qualitative data, which could be an advantage. Unfortunately, such quantification requires a well-defined granularity of segmentation, so making such decisions at the start of the analysis prematurely structures the exploration space and prevents a completely open exploratory approach.

The Basic Ideas of Grounded Theory

GT, first described in Glaser and Strauss (1967), is a data analysis approach that is largely data driven and aims at producing a theory that describes interesting relationships between things, situations, events, and activities (together called *phenomena*) reflected in the data by means of abstract *concepts*. The term *grounded* indicates that this theory will contain only statements derived from actual observations in a manner that can be traced back to these data: The theory is grounded in the data.

We use the variant of GT described by Strauss and Corbin (1990), who suggest three (partially parallel) activities for a GT-based data analysis:

1. *Open coding* describes the data by means of conceptual (rather than merely descriptive) codes, which are derived directly from the data.
2. *Axial coding* identifies relationships between the concepts described by these codes. Strauss and Corbin (1990) suggest a concrete set of relationships to check for (in particular: *causal conditions* leading to phenomena that exist in a *context* featuring *intervening conditions* and leading to *participant's strategies* that create certain *consequences*). These relationships (plus the slightly fuzzy notion of forming *categories*) they call a *paradigmatic model*, a term we will use further below.
3. *Selective coding* extracts a subset of the concepts and relationships found and formulates them into a coherent theory. Selective coding is not relevant for the development of a coding scheme and thus will not be discussed in the present article.

Strauss considered the following three aspects to be the core of the GT method, saying in an interview that only these are required in order to call something GT (Legewie & Schervier-Legewie, 1995):

- *Theoretical coding:* Codes are theoretical, not just descriptive. They reflect concepts that have potential explanatory value for the phenomena described.
- *Theoretical sampling:* The selection of the material to be analyzed is made incrementally during the course of the analysis, based on what is expected to be most relevant for the theory under development.
- *Constant comparison:* Observed phenomena (and their contexts) are compared many times in order to create codes that are precise and consistent.

Theoretical sampling is of less interest in the present article, but theoretical coding and constant comparison are of vital importance to understand the discussion.

DATA USED FOR THE ANALYSIS OF PAIR PROGRAMMING

In the following subsections, we describe our observation context (programmers and task). We also describe the data capturing method used.

Observation Context: The Origin of Our Data

We observed (in the manner described below) seven pairs of graduate students who all worked on the same task. Six of them had worked together as pairs previously. The average work time (which was not limited) was 3.8 hours. The students were all participants of a highly technical course on enterprise information systems and the Java2 Enterprise Edition (J2EE) architecture and technologies. The specific task called for an extension of an existing Web shop application. The task required broad passive J2EE knowledge for analyzing and understanding the existing system and specific operational knowledge about Java Message Service (JMS), Java Naming and Directory Interface (JNDI), and the JBoss application server[1] for programming, configuring, and testing the actual extension. The task was not easy; only three of the pairs were completely successful. The other four pairs terminated their work before it was completely finished. They did not believe it to be possible to solve the remaining problems in an acceptable time frame.

For the analysis described in the present article, we used the session of one of the successful pairs only. This session ran 2 hours and 58 minutes.

Observation Method: Data Capturing Procedure

Since we did not know in advance what would or would not be important, we needed to start from a rather rich data set. We used three different data sources:

- An audio recording captured verbal communication between the participants, as well as other noises, vocal or other, that may have helped with the interpretation of the data.
- A frontal-perspective video of the programmers (shot from above and behind the screen and reaching down to about waist level) captured aspects of facial expression, gestures, posture, direction of attention, and, most relevantly, who was operating mouse and keyboard at any given time.
- A full-resolution screen recording captured almost all computer activities of the programmers on a fairly fine-grained level.

All three recordings were made simultaneously using Camtasia Studio[2] and unified into a single, fully synchronized video file in which the camera video was superimposed semitransparently onto a corner of the screen video. In this way, all data was visible at once (multidimensional video).

The session was recorded in an otherwise silent office. Combined with the high audio quality of a high-end webcam[3], this arrangement provided good acoustical playback conditions.

PROBLEMS OF A PLAIN GROUNDED THEORY DATA ANALYSIS APPROACH

Attempting GT-style exploratory analysis of the rich data set described above (actually a precursor study, but very similar in all respects), we quickly recognized that transcription was not practical. Too much relevant information in the screen recording—source code fragment input, used features of the development environment (such as browsing across different files or positions within files), pointing with the mouse during discussion with the partner, and so on—proved unclear in how to go about, or impractical in the effort of, transcribing.

This is why we decided to work on the raw video directly. We chose the qualitative data analysis software ATLAS.ti[4] for achieving this task, which is one of the few products that allows direct annotation to video.

One of us, Stephan Salinger, started open coding in the manner suggested by Strauss and Corbin (1990). The short-term goal was to characterize the activities occurring during pair programming; the long-term goal was to identify recurring behavioral patterns and classify them as helpful, hampering, ambivalent, or neutral.

This approach generated as many as 194 distinct concepts and almost complete confusion and despair in the course of a few days of analysis due to the following problems:

- No predefined focus: We had no criteria for selecting which observations (verbal interaction, facial expressions, gestures, posture, directions of gaze, subverbal vocal noises, nervous tics, computer input, input methods, computer output, etc.) to code and which to ignore, and consequently were overwhelmed by the data.
- No predefined granularity: We made no prior decision regarding the level of detail worth coding. As a result, we produced codes on different levels of detail (e.g., coarse ones such as *handle problem* and finer ones such as *test defect fix*), which were difficult to delineate against one another subsequently.
- No predefined level of acceptable subjectivity: The nature of the codes chosen in GT can be anywhere on the spectrum, ranging from codes that reflect observations that any observer could agree with to codes that interpret the observation to a degree that could be called wishful thinking. GT as such does not provide a criterion for deciding where "grounded in data" ends and wishful thinking begins. As a consequence, we mixed objective–descriptive and subjective–evaluative attitudes for selecting codes. This led to codes of different nature (e.g., descriptive ones such as *uses documentation* and assumption-bearing ones such as *gains knowledge of detail*) existing side-by-side, which made it harder to decide which code to use in a particular case.
- Too many topics: The codes described too many different topics of interest, making it impossible to properly focus on anything. None of the resulting collections of information ever reached a useful degree of completeness.
- Lack of concept grouping: The diversity of topics also distracted from forming what GT calls categories: a few large groups of heavily interrelated concepts, say, human-human interaction (HHI) or human-computer interaction (HCI).
- Importance misjudgments: The high attention to a broad set of concepts overtaxed our ability to judge their importance so that, because of the large number of concepts we introduced, we completely overlooked a number of important ones.

After we had noticed and gradually understood a number of these problems, we stopped this mode of investigation completely. We restarted the complete analysis from scratch (but very slowly and carefully, and with considerable backtracking) and concurrently redesigned the coding procedure. The result of this redesign was a number of heuristic practices described below that help using the GT analysis process.

PRACTICES SUPPORTING THE ANALYSIS OF COMPLEX VIDEO DATA

The methodological heuristics presented here form the heart of the present article. These intertwined practices serve to reduce or solve the problems described in the previous section. After introducing them, we will present an application that shows how they work together and mutually support one another.

Practice 1: Perspective on the Data

Strauss and Corbin (1990) suggest that the start of selective coding (that is, after open coding and axial coding have been going on for quite some time) is the time when you should begin to decide what is important and what is less so. As described above, we found that this is not practical when working with rich video data. There are three reasons why a perspective used for the analysis should be defined before starting:

- to avoid drowning in detail;
- to provide consistency in the criteria used for creating and assigning concepts;
- to focus attention on the most relevant aspects.

This perspective can be defined by formulating answers to the following questions. These answers should be reviewed (and perhaps revised) several times in the course of the analysis:

1. In which respects do you expect the data to provide insight?
2. What kinds of phenomena do the researchers allow themselves to identify in the data?
3. What type of result do you want the analysis to bring forth?

Question 1 does not ask what you expect to find, only in what respects you expect to find *something.* The answer acts as a filter that tells you which phenomena should receive more attention than others. Furthermore, constantly rechecking and adjusting the answer to this question helps in deciding when to stop the analysis, when to modify (or even replace) your research question, and when to obtain further or different raw data. In our case, the expectation was that the data could help understand what activities dominate the pair programming process and how they relate.

Answer 2 provides the mechanism for systematically bounding the nature and amount of subjectivity to be found in the conceptualizations of the data. The strongest restriction would be to allow only concepts that express directly observable phenomena, resulting in a behaviorist (stimulus/response) research perspective. Weaker restrictions might also allow concepts referring to unobservable processes (such as attitudes or thinking processes of actors), concepts that involve predictions (such as “helpful for reaching goal X”), and/or

concepts expressing moral judgment (good, bad). We were convinced that, in our case, only the behaviorist perspective would enable us to trust our own results.

Finally, the result type is the standard used for deciding how much attention to invest in which kinds of phenomena when the analysis resources begin to get scarce (which very quickly they will). It helps to stay on track. Do we want to produce a full conceptual theory, just a conceptual structure (system of categories) for the data, or even just a coding scheme? In our case, the goal was just to produce a coding scheme, because we felt we knew so little about the internals of pair programming that we should not yet decide on an actual engineering research question.

Practice 2: Concept Name Syntax Rules

Choosing concept names is another area where we found that giving up some of the freedom postulated by plain GT is beneficial. We found that our initial freely chosen concept names turned out to be highly variable and hence difficult to understand, remember, and compare.

As a remedy, we developed a structured naming scheme. Within the confines we set for ourselves by Practice 1, that is, describing directly observable activities of the pair programmers, the scheme does not predetermine anything with respect to the meaning of a concept: It only prescribes the shape of its name. When working with this scheme, we observed the following benefits:

- A concept will be better understood right at introduction time.
- A naming scheme facilitates managing a large set of concepts consistently.
- Some relationships between concepts are implicitly recorded as well, which greatly simplifies axial coding and the forming of categories.
- A concept name explicitly represents several aspects at once, which simplifies the fundamental GT practice of constant comparison.
- It becomes easier to understand where difficulties in delineating one concept against another arise, and correspondingly easier to obtain insights into the weaknesses of the overall conceptual description in practice.

In our case, the concepts needed to describe individual activities by one or both of the pair members, although for other domains of analysis different code naming structures might be preferable. Our concept name was structured like a complete sentence:

code = <actor>.<description>
actor = P1 | P2 | P
description = <verb>_<object>[_<criterion>]

Examples for such concept names are P1.ask_knowledge and P2.explain_knowledge. The criterion element of the structure can be used for additional specialization where needed. Given such codes, subsequent analysis can very easily abstract, for instance, the verb element (to compare contexts of objects) or the object element (to compare the variants of action types). Without such complex codes, the same situation would probably be modeled by a tuple of codes with relationships. So while finding relationships in plain GT involves axial coding, in our case recording at least some relationships became a fringe benefit of open coding.

Practice 3: Analysis Results Metamodel

When we started practicing GT, we found some of the terminology and concepts confusing. First, where GT talks about phenomena, conceptualization, concepts, properties, categories, and relationships, our analysis software (ATLAS.ti) talks about quotations, annotation, concepts, concepts, families, and relationships, respectively—and even the term *relationships* denotes two different notions.

Second, even after the initial learning phase, some of the differences were subtle enough that we misapplied them every once in a while. As a result, we became confused when trying to reconstruct what we had meant to express.

Third, when decisions regarding the introduction or demarcation of codes became difficult (which they often did), we realized we needed guidance for systematically applying the ideas of GT to break out of the situation in an appropriate way. (An example of this will be given in the section presenting the practices' application.)

Fourth, we extended the terminological framework with additional ideas related to the nature of our data, in particular the notion of a Track for partitioning data in order to support data visualization for a better overview of nested and parallel activities.

Together, these issues prompted us to formulate an explicit analysis results metamodel, that is, a model of the concepts that describe the structure of an analysis result. We formulated this metamodel as a UML class model (Rumbaugh, Jacobson, & Booch, 2005), which is shown in Figure 1.

Here is a very short description of the model's elements: a *Quotation* defines a fragment of the data (a scene of the video) the analysis refers to. An *Annotation* connects Quotations with a *Concept*. Concepts can be grouped into a *ConceptClass*; a single Concept can be a member of many ConceptClasses.

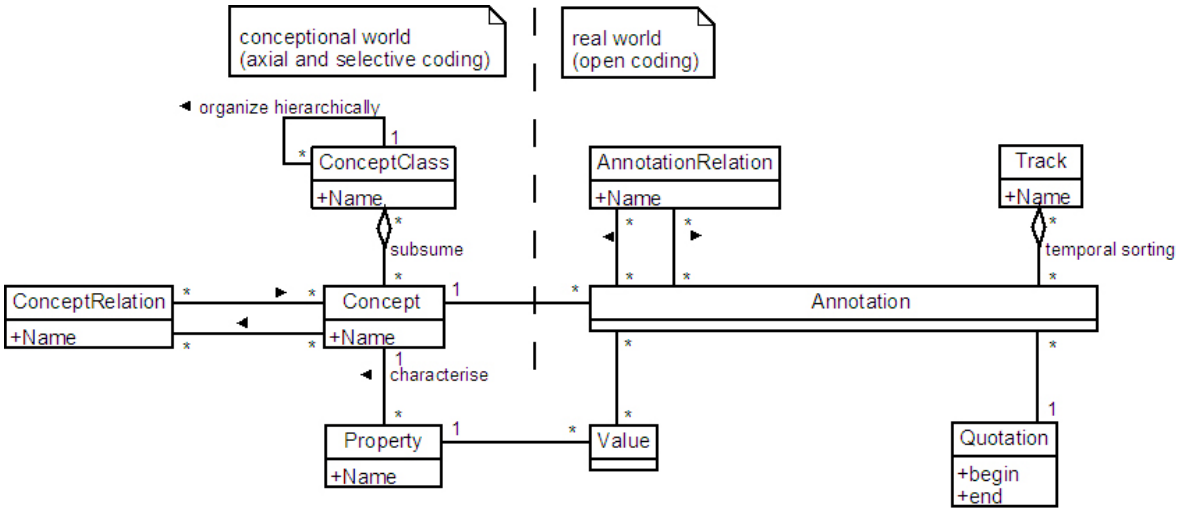

Figure 1. Complete metamodel of analysis results formulated as a UML class model. Boxes denote the various different kinds of elements occurring in our GT analysis results and the lines describe the relationships between them.

In order to further differentiate Concepts, they can be attributed with *Properties* that have *Values*. This allows developing concepts in a data-driven manner during axial coding and is helpful for identifying relationships between concepts (Strauss & Corbin, 1990).

A *ConceptRelation* is used to describe a relationship between Concepts, for instance according to the paradigmatic model. In many cases, such a relationship is not valid for all pairs of Annotations that use these Concepts; it can then be expressed individually by using *AnnotationRelation*. A *Track* allows for defining subsets of annotations that help identify various kinds of recurring relationships on the concept level, typically by means of appropriate visualization, as shown in Figure 2.

In addition to describing the structure of analysis *results* (to avoid terminological confusion), the metamodel also acts as a repository of ideas for the analysis *process*. For instance, when one is unsure whether a certain ConceptRelation will always hold, the metamodel suggests initial annotation of the currently known *instances* only (AnnotationRelation) and deferring the creation of the more general ConceptRelation until sufficient evidence is available.

Note that the metamodel is meant to be used throughout all phases of the GT research process. Some of its elements (e.g., Tracks) are used only rarely during the development of a coding scheme, as described in this article.

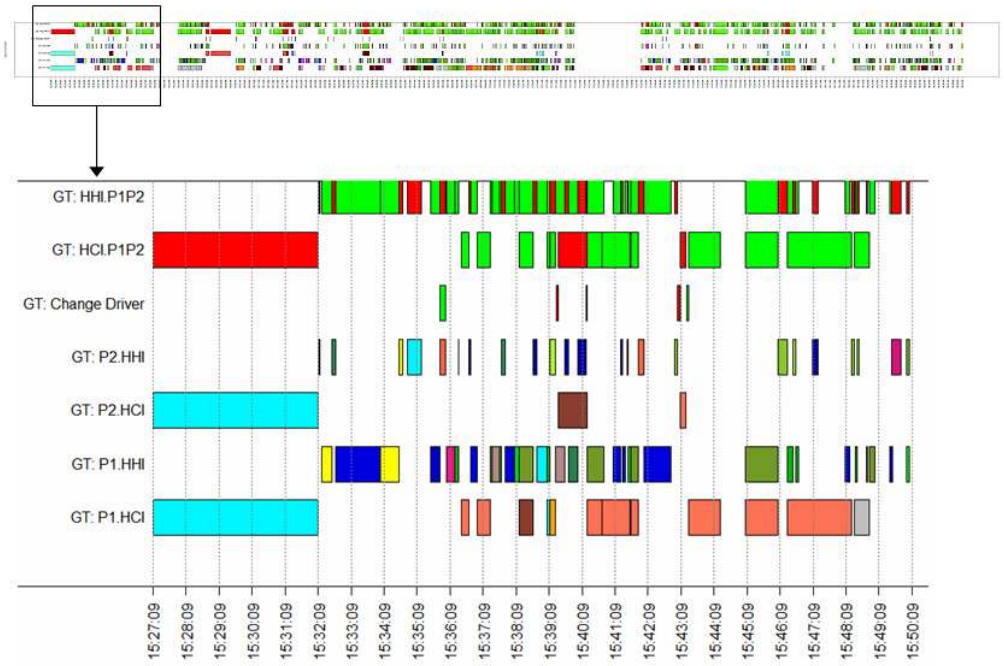

Figure 2. An example of a visualization of Tracks: The upper part shows a heavily scaled-down, automatically generated visualization of the GT annotations for a full pair programming session of 2 hours and 58 minutes. The lower part shows a magnified excerpt containing in particular the following four tracks: Track HHI.P1P2 represents the HHI activities of P1 (green) and P2 (red); HCI.P1P2 is the corresponding view of the HCI activities. Track P1.HHI represents each type of HHI activity performed by P1 in a different color; P1.HCI is the corresponding view of the HCI activities.

Practice 4: Pair Coding

The central and most important practice is pair coding. *Pair coding* means that all coding work is done by two people working together at one computer (much like pair programming, but that is just a coincidence). The key idea of pair coding is to require a consensus of two people for all important decisions: Which phenomena found in the data to single out for coding; where in time such a phenomenon starts and ends; which existing concept to use for coding this phenomenon; when to create a new concept; how to name that concept.

We found a number of benefits associated with pair coding as compared to a single researcher, some of them very important for successful GT work:

- Concept definitions become more exact, because they are scrutinized more closely upon their introduction. This effect is further supported by the structured naming scheme (Practice 2).
- The differentiation between similar concepts also becomes more precise, due not just to better definitions but also because a pair is less likely to let a concept slip in that is on a much different level of granularity than the others (and hence likely to have big overlaps with one or more existing concepts).
- Remaining concept differentiation problems will not be ignored but rather discussed. If they can be resolved, this will happen at an earlier point in time, leading to fewer incorrect concept assignments and therefore less rework. If it is impossible to fully resolve them (a not uncommon situation), the discussion will help understanding why, leading to a better understanding of the concepts involved.
- The perspective on the data (Practice 1) is maintained more consistently.
- The perspective on the data is refined more regularly and more thoroughly.
- A larger number of relevant phenomena are detected and encoded.

These results are in tune with psychological research suggesting that groups will often produce better decisions than isolated individuals (Shaw, 1981). Under adverse circumstances, *groupthink* (i.e., excessive concurrence seeking in groups) may make group decisions worse (t'Hart, 1988). But there is hardly any danger that this will happen in our setting: Groupthink is most likely in cohesive groups with a dominant leader, where the group is sharing common stereotypes and producing group pressures towards conformity (Janis, 1982). Since it is one of the routine tasks of any pair coder to challenge stereotypes used by the partner and to strive towards identifying possible different viewpoints, only a dominant person can pose any danger of groupthink in a pair-coding context. If the coders are equals, groupthink will be highly unlikely to happen.

Taken together, these four practices provided a quantum leap in the usefulness of our analysis results. The next section will illustrate this with a number of examples that will also show how the practices complement one another.

APPLICATION OF THE PRACTICES AND SOME RESULTS

This section will present a few fragments from the analysis process that used the practices described above and that led to our coding scheme for pair programming. We present these

examples to make the practices clearer, to explain how they interact, and to make it more credible that they help vitally.

We first introduce four concepts from our coding scheme and then present some episodes from the process in which we created them. Finally, we state a few hypotheses about pair programming that we have derived based on our coding scheme.

An Extract from the Coding Scheme

Our current version of the coding scheme (which ignores the subject part of the concept names) contains about 50 different concepts, clustered into about 20 overlapping ConceptClasses, with most concepts being members of either two or three of them. As an illustrative example we present the four concepts of the *ThinkAloud* ConceptClass. They are shown in Table 1; the descriptions are heavily summarized.

Use of the Practices: A Few Examples

Early during the coding process we recognized that the so-called driver (Williams, Kessler, Cunningham, & Jeffries, 2000) frequently verbalized what he was doing on the computer. Based on this observation, we made two decisions. First, we developed two ConceptClasses (see Practice 3) called HCI (human–computer interaction) and HHI (human–human interaction) for separating the computer-operating aspect from the verbalization aspect. These were ConceptClasses rather than individual concepts because the same separation would obviously be relevant in many other cases as well. Second, we postulated a new concept, ThinkAloud_Activity. By virtue of the concept naming syntax structure (Practice 2), this one concept immediately generated a whole ConceptClass (although having only one member at first) based on the verb *to think aloud*. This effect led to extended differentiation of concepts where needed but incurs only little additional complexity for the coding scheme.

We introduced ThinkAloud_Finding as the second member of this class, when we found a phenomenon that was obviously thinking aloud but did not explain computer activity. The demarcation appeared to be relatively clear. In the discussion of the pair coders (Practice 4), we agreed that ThinkAloud_Activity can be used only for the driver and that it has priority where ThinkAloud_Finding might also be applicable.

Table 1. The Concepts of the ThinkAloud ConceptClass.

Concept name	Description
ThinkAloud_Activity	Explains a current computer-operating activity
ThinkAloud_Finding	States a newly won insight (e.g., that some prior action was a mistake)
ThinkAloud_State	Reflects on the current state of work with respect to the current strategy and goal
ThinkAloud_Completion	States that a simple work step has been completed

Soon thereafter we encountered a programmer's explanation of the state of affairs and recognized it could be annotated as ThinkAloud_State, thus creating the third member of this set of concepts. But we soon found ThinkAloud_State to exhibit two problems. First, we had a case where it collided with ThinkAloud_Finding, because the finding concerned the state of work. Second, it designated statements on rather different levels of abstraction and granularity.

We solved both problems by using the metamodel (Practice 3), specifically by introducing the ConceptRelation "is-precondition-of" from the existing concepts *Propose_Step* (suggesting the next step) and *Propose_Strategy* (suggesting an approach for choosing many future steps). We postulated that ThinkAloud_State had to refer to a previous Propose_Strategy and introduced a new concept ThinkAloud_Completion that would refer to a previous Propose_Step. This solved both problems at once: We could now discriminate large and small granularity (strategic and tactical) and gained a criterion for when not to use ThinkAloud_Finding, which provided the demarcation to the other two.

This illustrates how open coding naturally leads into axial coding and how the combination of the paradigmatic model with the concept naming syntax (Practice 2) can show a way back into open coding, thus keeping the complexity of the resulting annotations down.

We are convinced that this route worked only because of the pair coding constellation (Practice 4), since both coders initially suggested encodings based on the existing codes and only the nonacceptance of these suggestions (and their supporting arguments) by the other led to the discovery of the "is-precondition-of" relationship and the fourth code ThinkAloud_Completion.

Some Hypotheses Based on the Coding Scheme

Although we have not yet started the analysis of the actual pair programming process as such, a number of phenomena recurred so consistently that we already call them hypotheses:

- We have found no evidence that the driver and the observer do indeed work on different levels of abstraction, as claimed in the pair programming literature (Williams et al., 2000). Similar results have been reported for pair programmer discussions by Bryant, Romero and du Boulay (in press), Freudenberg (née Bryant), Romero and du Boulay (2007; based on quantitative–qualitative work), and by Chong and Hurlbutt (2007).
- We have observed what we call *pair phases*, characterized by a high density of communication acts referring to just one narrow issue. They look a lot like what descriptions of pair programming suggest as the normal pair programming process, but we realized they are all of short duration (usually under 3 minutes).
- We believe that pair programming is not driven by strategic planning and monitoring. Rather, the plan is quite often only one step long: A single step is suggested, possibly discussed, decided (or revised), and immediately executed.
- Besides the unavoidable roles of driver and observer, pair programming sessions apparently tend toward implicitly producing a leader role as well. The leader is the person more skilled for the given task and influences speed and direction of the process much more strongly than the pair partner, no matter which role the leader is taking.

We expect that valuable insight about pair programming can be gained by investigating the reasons, consequences, and typical context conditions of the above trends. For instance,

we expect to find that pair phases are episodes of super-high productivity; it would be helpful to understand when and why they occur.

RELATED WORK

Qualitative Analysis of Pair Programming

We know of no other work analyzing the process of pair programming that uses a real GT approach: Most similar works use at least partially predefined coding schemes and most perform quantitative–qualitative analyses by means of protocol analysis or verbal analysis. We are also not aware of any work that is using video data directly in the analysis process.

Wake (2002) presented a list of typical pair programmer activities, but provided little information on how it was derived. Bryant (2004) studied the difference in interaction type and frequency in novice versus expert pair programmers. In a pilot study, she first refined Wake's list into a table of 11 behavior and interaction types. In the actual study, she then recorded the sequence of events in real time according to this schema and analyzed these data in a mostly quantitative way.

Such real-time categorization is obviously a good precondition for analyzing a large number of sessions, which is a positive approach. On the other hand, the simplicity of the categorization that is needed to make it possible also restricts the results to analyzing in terms of the rather simple concepts already presented in the predefined list. Neither subtle discriminations nor surprising new insights appear likely from this approach: It is applicable only in narrowly scoped investigations using predefined hypotheses.

Bryant et al. (in press) investigated behavior related to the driver and observer roles. They started from audio recordings, transcribed them, and annotated exactly each sentence with one out of the six predefined codes. The coding scheme is based on Pennington (1987) and characterizes the abstraction level. The analysis is mainly quantitative. This research aims at confirming or rejecting a conventional wisdom and is thus rather more hypothesis-driven than exploratory. A similar assessment applies to Freudenberg et al. (2007).

Cao and Xu (2005) investigated the activity patterns of pair programming. Pair working sessions were videotaped and then transcribed. The analysis used a coding scheme based on a combination of the schemes from Lim, Ward and Benbasat (1997) and Okada and Simon (1997). Then, during the analysis of the data, a new schema was developed in a manner not described. This work shares our behaviorist observation attitude; unlike our approach, however, it ignored all information contained in the computer interaction even though it was still grounded in only objectively observable communication acts.

In contrast, Xu and Rajlich (2005) used the dialog-based protocol in order to analyze the cognitive activities in pair programming, which involves a far greater amount of either subjectivity or generalized assumption. The coding scheme involved classification heuristics derived from a theory on self-directed learning (Xu, Rajlich, & Marcus, 2005). Xu and Rajlich proposed to do the coding assignment by two or more coders. In contrast to our approach, the coders worked separately and compared the results afterwards. This approach is sensible only with a fixed coding scheme; a GT-like generation of concepts would be very inefficient in this manner. Immediate discussion, as in pair coding (Practice 4), is much more efficient.

It is obvious that all five studies use rather predefined concepts during the analysis than concepts grounded only in the data. We fear that such approaches will be much more likely to fall prey to unwarranted assumptions according to conventional wisdom, such as the presumed driver/observer role differences, and so on.

Grounded Theory Work Using Rich Video Data

Even in the broader GT-related literature, examples of studies using video during the analysis (rather than transcripts of videos only) are rare. We found one such example in medicine that studied medical team leadership behavior (Xiao, Seagull, Mackenzie, & Klein, 2004). The video was recorded with four cameras from different angles. The analysis involved four analysts and three steps: (a) One analyst identified video segments with interesting verbal or nonverbal team interactions; (b) Two analysts created conceptual descriptions of the segments by consensus; and (3) Taxonomies for leadership actions from the conceptual descriptions were developed. This approach resembles our pair coding practice, at least in Step 2. If different people performed Steps 1, 2, 3 (the article is very unclear in this respect), we consider this a problematic procedure: It is almost antithetical to the GT philosophy, because it partially prohibits constant comparison and fully prohibits the intertwining of open coding (Steps 1 and 2) and axial coding (Step 3).

CONCLUSION AND FURTHER WORK

We have described why a straightforward application of the standard GT method on multidimensional video data of pair programming sessions is not likely to be successful. Furthermore, we presented and illustrated a set of four analysis practices that provide a systematic way to hold the analysis problems at bay:

- *Perspective on the data* helps avoid drowning in detail.
- *Concept name syntax rules* help create useful and consistent concept names.
- *An analysis results metamodel* helps keep the analysis process systematic and the results well structured.
- *Pair coding* mitigates the effects of limited or distorted perception.

We have used these practices to generate a general-purpose coding scheme of pair programming activities, of which we presented a small excerpt. In the future, we will proceed with the following steps:

- Validation of the coding scheme. We will encode sessions that have very different properties with respect to participants, task, and setting.
- Qualitative and quantitative evaluation of the coding process itself, based on its results, intermediate results, and process monitoring information (in particular timestamps) recorded by ATLAS.ti.
- Refinement of the coding scheme with respect to particular research applications, in particular by adding properties according to the metamodel.
- Application of the coding scheme to produce actual grounded theories of several aspects of the pair programming process. This will require selective coding through

which we expect to exercise even those parts of the metamodel not discussed in the present article.

Just like the four practices mutually support one another, these tasks will also exhibit synergy and so will be performed partially in parallel.

ENDNOTES

1. See http://labs.jboss.com/
2. A product of the TechSmith Corporation, http://www.techsmith.com
3. Logitech 5000 webcam
4. See http://www.atlasti.com/

REFERENCES

Beck, K. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Boston: Addison-Wesley Professional.

Bryant, S. (2004). Double trouble: Mixing qualitative and quantitative methods in the study of extreme programmers. In *Proceedings of the 2004 IEEE Symposium on Visual Languages: Human Centric Computing* (VL/HCC '04; pp. 55–61). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/VLHCC.2004.20

Bryant, S., Romero, P., & du Boulay, B. (in press). Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies.*

Cao, L., & Xu, P. (2005). Activity patterns of pair programming. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* (HICSS '05; p. 88a). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/HICSS.2005.66

Chi, M. T. H. (1997). Quantifying qualitative analyses of verbal data: A practical guide. *Journal of Learning Sciences, 6*, 271–315.

Chong, J., & Hurlbutt, T. (2007). The social dynamics of pair programming. In *Proceedings of the 29th International Conference on Software Engineering* (ICSE '07; pp. 354–363). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/ICSE.2007.87

Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis: Verbal reports as data*. Cambridge, MA, USA: MIT Press.

Freudenberg, S. (née Bryant), Romero, P., & du Boulay, B. (2007). "Talking the talk": Is intermediate-level conversation the key to the pair programming success story? In *AGILE 2007* (pp. 84–91). Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/AGILE.2007.1

Glaser, B. G., & Strauss, A. L. (1967). *The discovery of grounded theory: Strategies for qualitative research*. New York: Aldine de Gruyter.

Hughes, J., & Parkes, S. (2003). Trends in the use of verbal protocol analysis in software engineering research. *Behaviour and Information Technology, 22*, 127–140.

Janis, I. L. (1982). *Groupthink* (2nd ed.). Boston: Houghton Mifflin Company.

Legewie, H., & Schervier-Legewie, B. (1995). Im Gespräch: Anselm Strauss [An interview of Anselm Strauss]. *Journal für Psychologie, 3*, 64–75.

Lim, K., Ward, L., & Benbasat, I. (1997). An empirical study of computer system learning: Comparison of co-discovery and self-discovery methods. *Information Systems Research, 8*, 254–272.

Lui, K. M., & Chan, K. C. (2003). When does a pair outperform two individuals? In M. Marchesi & G. Succi (Eds.), *Extreme programming and agile processes in software engineering* (Lecture Notes in Computer Science 2675, pp. 225–233). Berlin, Germany: Springer.

Nawrocki, J. R., Jasiński, M., Olek, Ł., & Lange, B. (2005). Pair programming vs. side-by-side programming. In I. Richardson, P. Abrahamsson, & R. Messnarz (Eds.), *Software process improvement* (Lecture Notes in Computer Science 3792, pp. 28–38). Berlin, Germany: Springer.

Okada, T., & Simon, H. (1997). Collaborative discovery in a scientific domain. *Cognitive Science, 21*, 109–146.

Pennington, N. (1987). Comprehension strategies in programming. In G. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Corp.

Rumbaugh, J., Jacobson, I., & Booch, G. (2005). *The unified modeling language reference manual* (2nd ed.). Boston: Addison-Wesley Professional.

Salinger, S., Plonka, L., & Prechelt, L. (2007). A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. In J. Sajaniemi, M. Tukiainen, R. Bednarik, & S. Nevalainen (Eds.), *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group* (pp. 144–157). Joensuu, Finland: Department of Computer Science and Statistics, University of Joensuu. Also available at http://www.ppig.org/papers/19th-Salinger.pdf

Shaw, M. E. (1981). *Group dynamics: The psychology of small group behavior*. New York: McGraw Hill.

Strauss, A., & Corbin, J. (1990). *Basics of qualitative research: Grounded theory procedures and techniques*. London: Sage Publications, Inc.

t'Hart, P. (1988, July). *Groupthink: Observations toward a theory*. Paper presented at the meeting of the International Society of Political Psychology, Meadowlands, NJ, USA.

Wake, W. (2002). *Extreme programming explored*. Boston: Addison-Wesley.

Williams, L. (2001). Integrating pair programming into a software development process. In *Proceedings of the 14th Conference on Software Engineering Education and Training* (CSEET '01; pp. 27–36*)*. Washington, DC, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/CSEE.2001.913816

Williams, L., Kessler, R. R., Cunningham, W., & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE Software*, *17*(4), 19–25.

Xiao, Y., Seagull, F., Mackenzie, C., & Klein, K. (2004). Adaptive leadership in trauma resuscitation teams: A grounded theory approach to video analysis. *Cognition, Technology & Work, 6*, 158–164.

Xu, S., & Rajlich, V. (2005). Dialog-based protocol: An empirical research method for cognitive activities in software engineering. In *International Symposium on Empirical Software Engineering* (ISESE 2005; pp. 383–392). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.ieeecomputersociety.org/10.1109/ISESE.2005.1541848

Xu, S., Rajlich, V., & Marcus, A. (2005). An empirical study of programmer learning during incremental software development. In *Fourth IEEE Conference on Cognitive Informatics* (ICCI 2005; pp. 340–349). Los Alamitos, CA, USA: IEEE Computer Society. Retrieved April 11, 2008, from http://doi.acm.org/10.1145/1145287.1145289

Author's Note

All correspondence should be addressed to:
Stephan Salinger
Institut für Informatik
Freie Universität Berlin
Takustr. 9
14195 Berlin
Germany
salinger@inf.fu-berlin.de

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 26–46

USABILITY ASSESSMENT OF A UML-BASED FORMAL MODELING METHOD USING A COGNITIVE DIMENSIONS FRAMEWORK

Rozilawati Razali
Dependable Systems and Software Engineering, School of Electronics and Computer Science, University of Southampton, UK

Colin Snook
Dependable Systems and Software Engineering, School of Electronics and Computer Science, University of Southampton, UK

Michael Poppleton
Dependable Systems and Software Engineering, School of Electronics and Computer Science, University of Southampton, UK

Paul Garratt
Dependable Systems and Software Engineering, School of Electronics and Computer Science, University of Southampton, UK

Abstract: *Conceptual models communicate the important aspects of a problem domain to stakeholders. The quality of the models is highly dependent on the usability of the modeling method used. This paper presents a survey conducted on a method that integrates the use of a semiformal notation, namely the Unified Modeling Language (UML) and a formal notation, namely B. The survey assessed the usability of the method by using the grounded theory, the Cognitive Dimensions of Notations (CD) framework, and several criteria suggested by the International Organization for Standardization (ISO). Ten participants responded to the survey. The results suggest that the method is accessible to users when the principles and roles of each notation are obvious and well understood, and when there is strong support from the environment. Supported by the findings, a usability profile based on CD for designing a method that integrates semiformal and formal notations is proposed.*

Keywords: *empirical assessment, semiformal and formal notations, cognitive dimensions (CD), grounded theory, usability.*

INTRODUCTION

Modeling is vital in the development and maintenance of software systems. It allows the characteristics of the existing and future systems to be captured and understood. The modeling process produces models where the requirement specification is one of them. Software requirement specification is a conceptual model that establishes the connection between the user's needs of a system and the software solution to meet them. It is an abstract,

clear, precise, and unambiguous conception of a system, which is developed by using the appropriate notations. Some examples of notations used in conceptual modeling include semiformal notations such as entity-relationship diagram (ERD; Chen, 1976) and Unified Modeling Language (UML; Object Management Group [OMG], 2008), and formal notations such as Z (Spivey, 1992) and B (Abrial, 1996). In addition, there are also notations that integrate both semiformal and formal, such as UML and Z (Martin, 2003).

Formal notations such as Z and B use mathematical symbols to describe a system. The notations have three components: rules for determining the grammatical well-formedness of sentences (syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (semantics); and rules for inferring useful information (proof theory), which provides the basis for automated analysis of a model (van Lamsweerde, 2000). Formal notations therefore have the ability to increase a model's precision and consistency, which is necessary especially for critical systems (Hinchey, 2002). However, the notations are regarded as being difficult to comprehend, due to the usage of unfamiliar symbols and underlying rules of interpretation that are not apparent to many practitioners (Carew, Exton, & Buckley, 2005). On the other hand, semiformal notations such as ERD and UML provide abstract graphical representations for illustrating system elements. They are semiformal because, although they possess some formal aspects such as support the iterative refinement process, they cannot be used to verify or predict the vast majority of system characteristics (Alexander, 1996). As a result, an accurate and consistent model cannot be guaranteed. Nonetheless, the notations are perceived as more accessible, since it is easier to visualize the mapping of graphical symbols to the real-world objects they represent (Bauer & Johnson-Laird, 1993).

By integrating formal and semiformal notations, it may be that practitioners can produce a model that is accurate, consistent, and more accessible to them. One possible approach to this integration is to combine the formal notation of B and the semiformal notation of UML. A method called UML-B (Snook & Butler, 2006) is one such product. The rationale of this integration is that B has strong industrial supporting tools, such as Atelier-B (ClearSy Systems Engineering [ClearSy], n.d.) and B-Toolkit (B-Core Limited [B-Core], 2002), and UML has become the de facto standard for system development (Pender, 2003).

This paper presents an investigation into the usability of UML-B. Usability in this context means the understandability/comprehensibility, learnability, operability, and attractiveness of the method. The assessment was conducted by using the grounded theory and a usability evaluation framework, namely the Cognitive Dimensions of Notations (CD; Green, 1989; Green & Petre, 1996), with several usability criteria suggested by the International Organization for Standardization (ISO, 2003, 2004). The following section provides the background of the paper, which includes a brief description of CD and UML-B. Later, the survey is presented. The final section concludes the paper with a summary of the main findings and future work.

BACKGROUND

Cognitive Dimensions

The CD framework provides a comprehensive vocabulary for discussing the usability of programming languages, tools, and environments. It was originally proposed as a broad-brush

discussion tool, offering a vocabulary to discuss the usability tradeoffs that occur when designing programming environments (Green, 1989; Green & Petre, 1996). Nevertheless, it is also applicable beyond the programming environment. Since its proposal, the CD framework has been used as a basis of usability evaluation for several notations, such as UML (Cox, 2000; Kutar, Britton, & Barker, 2002), C# (Microsoft Corporation [Microsoft], 2008) programming language (Clarke, 2001), spreadsheet application (Tukiainen, 2001), and Z notation and tools (Triffitt & Khazaei, 2002).

The framework is generally seen as a tool that aids the usability evaluation of information-based artifacts (Green & Blackwell, 1998). The aim of the framework is to provide general guidelines that can be used to evaluate the usability and suitability of an artifact for a particular setting. An artifact is analyzed based on a usability profile that contains a CD set. The profile guides the evaluation of the artifact for a particular user activity. The framework distinguishes six main types of user activity (Blackwell & Green, 2003): incrementation, transcription, modification, exploratory design, searching and exploratory understanding. Each of these user activities is supported by a specific usability profile.

Table 1 provides the 14 dimensions in the CD framework, with summarized descriptions. Although the dimensions are conceptually independent, many of the dimensions are pairwise interdependent (Green & Blackwell, 1998). This means although any given pair can be treated as independent, a change in one dimension usually requires a change in some other dimension. For example, reducing a notation's viscosity may not affect its closeness of mapping, but it is likely to affect other dimensions, such as increasing the abstraction gradient. The framework considers this situation a matter of making compromises or tradeoffs in artifact designs.

Table 1. The CD Framework (drawn from Green, 1989).

Dimension	Description
Abstraction gradient	Level of grouping mechanism enforced by the notation
Closeness of mapping	Mapping between the notation and the problem domain
Consistency	Similar semantics are presented in a similar syntactic manner
Diffuseness	Complexity or verbosity of the notation to express a meaning
Error-proneness	Tendency of the notation to induce mistakes
Hard mental operations	Degree of mental processes required for users to understand the notation and to keep track of what is happening
Hidden dependencies	Relationship between two entities such that one of them is dependent on the other but the dependency is not fully visible
Premature commitment	Enforcement of decisions prior to information needed and task ordering constraints
Progressive evaluation	Ability to evaluate own work in progress at any time
Provisionality	Flexibility of the notation for users to play with ideas
Role-expressiveness	Purpose of an entity and how it relates to the whole component is obvious and can be directly implied
Secondary notation	Ability to use notations other than the official semantics to express extra information or meaning
Viscosity	Degree of effort required to perform a change
Visibility/Juxtaposibility	Ability to view every component simultaneously or view two related components side by side at a time

In essence, CD provides a framework for assessing the usability of building and modifying information structures. Because usability depends on the structure of the notation and the supporting tools provided by the environment, the dimensions are indeed applicable to the whole system.

UML-B

UML-B (Snook & Butler, 2006) is a graphical formal modeling notation and method based on UML (OMG, 2008) and B (Abrial, 1996). It uses UML's *Class* and *Statechart* diagrams as the graphical representation of its model. The Class diagram shows the structure and the relationships between system entities. The Statechart diagrams are attached to classes to describe their behavior. A notation, μB (micro B) that is based on B notation, is used for textual constraints and actions for the diagrams. μB has an object-oriented style dot notation that is used to show ownership of entities, namely attributes and operations by classes. The modeling environment of UML-B includes Rational Rose (IBM Software [IBM], n.d.) and a translator called U2B (Snook & Butler, 2006). Rational Rose provides the environment for the UML-B model development while U2B is a tool that translates a UML-B model to a B model so that it can be verified by B tools, such as Atelier-B (ClearSy, n.d.) and B-Toolkit (B-Core, 2002).

Figures 1 and 2 illustrate examples of a Class diagram and a Statechart diagram of a UML-B model, respectively. The Class diagram shows the entities and relationships involved in an Auction System. Two main classes, namely *USER* and *AUCTION*, are connected through *seller* and *highest_bidder* relationships. The Statechart diagram shows the states and transitions (operations) of the AUCTION class with the respective textual constraints specified using μB.

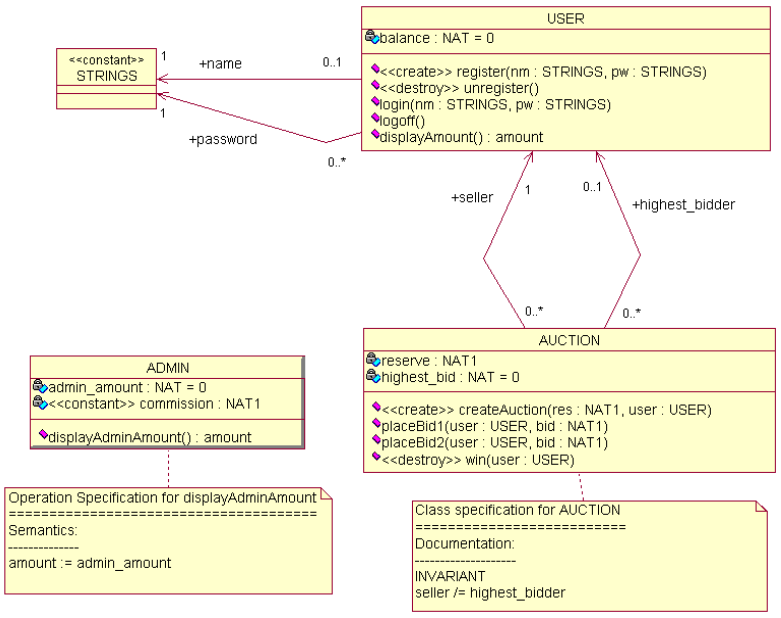

Figure 1. An example of a Class diagram of UML-B.

Figure 2. An example of a Statechart diagram of UML-B.

The comprehensibility of the notation used in a UML-B model has been assessed in previous work (Razali, Snook, Poppleton, Garratt, & Walters, 2007). The assessment was conducted as a controlled experiment that compared a UML-B model and a B model for model interpretation task. The measure of interest used in the experiment was efficiency in performing the task, that is, accuracy over time. The results suggest with 95% confidence that a UML-B model could be up to 16% (overall comprehension) and 50% (comprehension for modification task) easier to understand than the corresponding B model. The subjects commented that the UML-B model made it easier and quicker to understand the scenario and the relationships between operations; easy to develop, especially on computers; and more logical to developers. Nevertheless, the model was said to be useful only with good tool support. The UML-B model was also regarded as being quite "messy," since the information was scattered around the Class and Statechart diagrams.

SURVEY

The controlled experiment described briefly in the previous section evaluated the notation comprehensibility in terms of how easy it is to understand a UML-B model from the perspective of users who interpret the model. The results of the experiment suggest that the UML-B model is more comprehensible than the B model. The findings however cannot suggest by any means that the notation is also usable from the perspective of developers who use UML-B for modeling. Neither could they determine whether or not the notation suits the developers' common needs and expectations.

The following subsections present a further survey conducted on UML-B. The survey assessed the usability of the notation used in UML-B from developers' perspectives, especially from the point of view of users who have only recently started to use it. Since

usability depends on the notation and its environment, the evaluation included the tools that accompany the method, namely Rational Rose (IBM, n.d.) and U2B (Snook & Butler, 2006), whenever appropriate.

Objectives and Methods

The survey was qualitative in nature. Despite the fact that some of the data were quantified using an ordinal scale, the bulk of the analysis was interpretative. This type of analysis was carried out due to the problem at hand, that is, the survey attempted to understand the nature of experience of using UML-B. Since little is known about the UML-B method, the survey aimed to explore and gain novel understandings of its use through qualitative data and analysis. The analysis allows the intricate details about the phenomena, such as feelings, emotions, and thoughts to be extracted and analyzed.

Many different approaches to qualitative data are employed in the social sciences (Cassell & Symon, 1994; Denzin & Lincoln, 1994; Westbrook, 1994). We adopted one approach, namely the grounded theory (Glaser & Strauss, 1967; Strauss & Corbin, 1998). There are two variations in the approach, which are based on different directions taken by its originators, namely Glaser (1992) and Strauss and Corbin (1998). This survey employed Strauss' approach because it is more systematic and directive. In particular, it contains more formal models and procedures to generate theories. It also encourages a qualitative study to have a research question so that the researcher can stay focused amid the masses of data. In a qualitative study, the research question should be broad and open-ended.

The theory in the grounded theory approach is derived from data, systematically gathered and analyzed through the process. This approach was chosen because, unlike the controlled experiment conducted previously, this survey was not based on any specific theory. The grounded theory approach allows the study to be initiated without a preconceived theory in mind: The researcher can start with a phenomenon and allow the theory to emerge from the collected data. Because the theory is drawn from data, it is likely to offer insight, enhance understanding, and provide a meaningful guide to action (Strauss & Corbin, 1998). It is believed that the theory generated from this approach is more likely to resemble the reality, as compared to theory derived by merging concepts based on how one thinks things ought to work.

The survey aimed to formulate tentative theories of the usability of integrated methods, (combined semiformal and formal notations) such as UML-B, based on the understanding obtained from the qualitative analysis using the grounded theory approach. While a single study can never embrace all possible situations, the survey sought to provide some preliminary evidence of the integrated method's likely strengths and weaknesses when used under certain defined conditions. It was also intended to identify any threats that could hinder the method's usability and any opportunities that could improve the method further. The tentative theories could act as a basis for further investigation and analysis.

One of the subjective comments obtained from the earlier controlled experiment was that UML-B was seen as easy to develop, particularly on computers. The method also was deemed to be useful only with good tool support. These hypotheses were given by subjects who dealt with the already-developed UML-B model, not the process of modeling. This could suggest, therefore, that the hypotheses might not be true from developers' perspectives

for modeling purposes. As a result, the survey included these hypotheses in its investigation of the phenomenon through the following broad research questions:

- Do individuals who develop a model using the UML-B method perceive them (i.e., the method and the model) as usable (easy to understand, easy to learn, easy to operate, and attractive)?
- What are the characteristics of the UML-B method and UML-B model that affect their usability from the modeling perspective?

Materials

The survey instrument was developed based on the ideas proposed in the CD usability framework (Green, 1989). The framework was adopted because it captures a significant number of psychology and human–computer interaction (HCI) aspects that focus particularly on the notational design. The framework comprises 14 dimensions (see Table 1), which acted as the response variables in the survey.

The questions for the survey were constructed by following the proposed CD questionnaire (Blackwell & Green, 2000). The advantage of using a standard instrumentation, as proposed by the CD questionnaire, is that it has been assessed for validity and reliability by the authors. The CD framework is widely used by other researchers investigating the usability of notations, such as UML diagrams (Kutar et al., 2002) and Z (Triffitt & Khazaei, 2002), and so it provides a mechanism to compare the results of this survey with the results of other similar studies.

The CD questionnaire is intended to present the dimensions in general terms, applicable to all information artifacts, rather than presenting descriptions specialized to a specific system under consideration. The questionnaire was therefore tailored and modified slightly to reflect the characteristics of UML-B. Moreover, the questions for the survey were designed to include a set of answers using an ordinal scale together with the open-ended questions. This approach allowed the survey to obtain some quantitative measures rather than exclusively qualitative measures.

In addition to the CD framework, the questions on the survey were also constructed based on the usability criteria proposed by the International Organization for Standardization (ISO, 2003, 2004): understandability, learnability, operability, and attractiveness. There were 20 questions on the survey: 14 reflecting the dimensions of the CD framework, 5 representing the ISO's usability criteria, and 1 designed to gather suggestions for improvement. The 14 questions on CD were also mapped to at least one usability criterion of ISO. The mapping was based on the definition stated in the standard. The questions on the survey were presented in random order without following a specific sequence of dimensions. To ensure the questions were purposeful and concrete, the general guidelines on survey question construction were followed (Kitchenham & Pfleeger, 2002).

The questions used an ordinal scale that provided the respondents with five potential levels of agreement, from *–2* (*very difficult)* to *2* (*very easy).* An uneven number of levels were used because, by allowing for a neutral opinion, uneven numbers contribute to the achievement of better results (Bonissone, 1982). In addition to the selection on the scale, justification for the answer given was also required through open-ended questions, such as Why? or Which part? This acted as the qualitative data, which were used together with the quantitative data on the scale for the analysis. There were also questions that required an answer of Yes, No or Not sure.

The survey questions and raw data can be found in Razali (2007). As an overview of the questions, Figure 3 provides some examples of the survey questions. The first question concerns the visibility and juxtaposability dimension, which also relates to the operability/attractiveness criteria of the ISO. The second question involves the hard mental operations dimension that also implies the ISO's understandability/learnability criteria.

The CD framework describes the necessary conditions for usability based on the structural properties of a notation, the properties and resources of an environment, and the type of user activity: incrementation, transcription, modification, exploratory design, searching and exploratory understanding (Blackwell & Green, 2003). In particular, it addresses whether the users' intended activities are adequately supported by the structure of the notation used and its environment. For the survey, the identified users' intended activity was exploratory design, in which the users employed UML-B (notation and environment) to design a conceptual model. The survey questions and analysis therefore were tailored towards this aspect.

The survey questions were reviewed by a focus group prior to distribution. There were four people involved in the process. The purpose of the review was to identify any missing or unnecessary questions as well as to identify any ambiguous questions and instructions.

Participants

Ten participants responded to the survey. They were master's students of a software engineering program at the University of Southampton, who registered for the Critical Systems[1] course in spring 2006. They were chosen due to their potential contribution towards the development of usability theory for integrated methods such as UML-B. Specifically, they were selected because they received formal training on B (9 hours) and UML-B (1 hour) during the course. They also had completed courses on the object-oriented technology and formal methods of developing at some points in their studies. Basic knowledge of those aspects is necessary to develop a UML-B model. Moreover, the participants had some practical experience in using UML-B and its tools before participating in the survey. In particular, they used the method to develop a model of a system as part of their coursework towards the end of the Critical Systems course.

If you need to compare different parts of your UML-B model (e.g., between diagrams or windows of different operations, etc.), how easy is it to view them at the same time in Rational Rose?

Very difficult Very Easy
-2 -1 0 1 2

Why?

Do you find any complex or difficult tasks to work out in your head when modeling your UML-B model?

No Not Sure Yes

If Yes, what are they? If No or Not Sure, why?

Figure 3. Examples of the survey questions from Razali (2007).

The survey adhered to the university's ethical policies and guidance for conducting research involving human participants. The participants were aware that the survey was intended for research purposes. They were motivated to participate as it helped them in exploring the method in addition to providing a space for reflection on their learning prior to their course examination.

The subjects were in the final semester of their master's program. They therefore had a reasonable amount of experience and knowledge in software development. Some of them had some professional work experience in this area. They are the next generation of professionals, thus they represented closely the population under study: software developers who are new users of the UML-B method.

Results and Analysis

The survey adopted the grounded theory approach for the data analysis. In addition to capturing the informants' experiences of using UML-B, the survey aimed to formulate tentative theories on the usability of such integrated methods in general. The theory in the approach denotes a set of discrete categories that are systematically connected through statements of relationship. The categories in essence are abstract concepts that describe the phenomenon under study, whereas the statements of relationship are the interrelated properties of those categories.

Employing the grounded theory approach entails a number of coding and analysis processes. The first one applied was *open coding* where the responses were examined for objects of interest based on the stated research questions. The technique used was *microanalysis* (Strauss & Corbin, 1998). The analysis focused on identifying major themes or categories and how often they emerged in the data under varying conditions. The idea was to form a theoretical framework, thus the analysis involved the formulation of general categories rather than ones specific to any individual cases. For example, issues of using Rational Rose (IBM, n.d.) and running U2B (Snook & Butler, 2006) were conceptualized as Availability and Usefulness of Supporting Tools. The analysis did not intend to specifically delineate every single limitation of the tools. Rather, the objective was to identify and propose a set of categories that can be used as a basis for examining the usability of other similar methods in future.

After completing open coding, an axial coding process was conducted. *Axial coding* involves moving to a higher level of abstraction by identifying relationships between categories based on their properties. This forms the basis for the theory construction. The properties for the categories were derived by having queries such as *what*, *why*, *how* and *when* during the analysis process. For example, respondents mentioned the issue of learning UML and B several times in their answers. Therefore, Learnability of Notations and Tools was recognized as one of the categories. On the other hand, it is necessary to know what aspect of the notations and their tools was easy or difficult to learn, when and why they happened, in order to understand the phenomenon. To answer the queries, evidence was obtained and accumulated from various parts of the questionnaire. This included both the quantitative (ordinal scale) and qualitative (subjective) data. The use of CD framework and ISO's usability criteria that shaped the dimensions of usability investigation facilitated the identification of the categories and properties.

The following paragraphs list the categories and elaborate their properties. The properties (reasoning based on CD and ISO usability criteria) that support the statements are stated in the parentheses in the paragraphs. The properties were grouped into categories based on the respondents' qualitative answers and data on the ordinal scales (for details, see Razali, 2007).

Category 1: Model Structure and Organization. The UML portion of UML-B allows the system properties and behaviors to be illustrated using the Class and Statechart diagrams. Each diagram represents the system from a specific perspective. For example, the Class diagram shows the attributes and relationships between entities in the system while the Statechart diagram delineates the states and transitions involved in the system operations. In modeling a UML-B model, the users employ the diagrams to illustrate the system properties from these perspectives.

The diagrams are equipped with formal semantics, where the characteristics and behaviors of the systems are more precisely specified. Formal semantics in the form of B syntax are added at different parts of the diagrams so that the diagrams and semantics can be transformed to a B model. For example, the global variables and invariants are placed at the Class diagram level while the conditions and effects of the behaviors are placed at the Statechart diagram level. Despite being scattered throughout several parts of the model, the method has the ability to transform the diagrams and consolidate the semantics as a single B model through its tool, namely U2B.

Despite being logical, having the formal semantics at different parts of the model causes an accessibility issue for the users. They need to switch to different parts of the model to specify the formal semantics. Rational Rose supports the display of multiple windows at one time. However, having to deal with several displayed windows simultaneously in Rational Rose seems to be a problem (Property: visibility and juxtaposibility dimension). The users have to view not only the windows that display the Class and Statechart diagrams but also the pop-up windows that carry the semantics for each of the diagrams. In fact, some of these windows have to be on top of each other due to limited screen space. This leads the users to overlook certain aspects of the model and to become prone to errors (Property: error proneness dimension). The users can view and subsequently check the model using B tools by translating it to a B model using U2B at any modeling stage they like (Property: progressive evaluation dimension). However, having to transform the model, particularly while formulating and synthesizing ideas, has been regarded as a "noise." In addition, model transformation at early stages, where many aspects have yet to be carefully thought through, will generate error messages in B tools. And starting modeling with many generated errors can be a daunting experience, especially to new users.

This finding supports the comment obtained from the controlled experiment where the UML-B model had been regarded as messy. The messiness is caused not only by the scattered information but also the display of multiple windows simultaneously. The structure of the model does affect its accessibility for both model reading and development, even on the computer screen. The cognitive psychology theory that underpins this phenomenon is that humans have a limited amount of information that can be processed at one time. The way material is organized and presented has an effect (Chandler & Sweller, 1992). When the related information is separated on the page or screen, users have to use cognitive resources to search and integrate it. Users are less able to hold the separated information in working

memory simultaneously, especially if the information has a high intrinsic cognitive load (Sweller & Chandler, 1994). In general, a formal notation such as B syntax is high in intrinsic cognitive load because it involves concurrent interactions between its syntactical and semantic characteristics.

Because a UML-B model always involves the use of more than one UML diagram that carries the respective B syntax, the issue of scattered information is seen as unavoidable. However, the effect of split-attention can be reduced if the modeling tool allows more convenient and less distracting switching to and viewing different parts of the model.

Category 2: Availability and Usefulness of Supporting Tools. Rational Rose and U2B are the main supporting tools in UML-B. These tools have been useful in some aspects (Property: consistency dimension; secondary notation dimension; Learnability and Utility of U2B). On the other hand, several problems in user-friendliness were discovered by the users. For example, Rational Rose does not support some changes automatically, which causes the modification process to be unnecessarily tedious (Property: viscosity dimension). If a variable name is changed in the Class diagram, the change is not reflected in other parts, such as in the Statechart diagram or in the semantics where the variable name is used. A similar situation applies to variable deletion. Thus, the changes have to be done manually by visiting the respective parts of the model.

U2B in general has received a fairly good acceptance among the users. This is due to its obvious role, that is, to transform a UML-B model into a B model. By executing several simple steps, the users can generate a B model and execute the verification task using B tools (Property: progressive evaluation dimension). This is the reason why the tool is seen as easy to learn and use (Property: Learnability and Utility of U2B). The automatic transformation has alleviated some pains that would occur when modeling a B model from scratch. At the very least, it provides basic structures for the B model, which the users could extend further by adding more details. However, in order to keep the U2B simple, it does not contain a verification feature; the user would need to return to the B tools to achieve verification. As a result, no matter how simple to use, U2B, or even Rational Rose, does not support any type of checking. This means users have to transform the UML-B model to a B model and run it in B tools each time they change an idea, even if it involves only a minor change. Otherwise, there is no way to be sure whether or not the change is acceptable. The generated B model will contain numerous types of errors from the simplest to the most complex, which can only be recognized during model verification using B tools. Because of this reason, users feel that the method is less supportive for experimenting with ideas (Property: provisionality dimension). Users would benefit from having some simple checking abilities, such as unused variables and typing errors of B syntax at the modeling and transformation levels. This could act as the frontline checking to eliminate minor errors before pursuing more extensive verification in B tools. Rather than introducing all types of errors at once, evolutionary phases of checking could make the verification task less daunting and troublesome for the users. Because the tool currently lacks these elements, it does not fully meet the users' expectation (Property: Learnability and Utility of U2B).

This finding supports the comment obtained from the controlled experiment where several subjects in the experiment believed that the method is useful only with good tool support. Although the necessary tools are available, there are several aspects that should be improved in order to increase their utility (Property: Future Improvement). Perhaps a more

seamless modeling environment should be created so that users do not have to perform several individual and intricate steps during modeling.

Category 3: Learnability of Notations and Tools. The successful use of UML-B relies on the fact that users have to be familiar with UML and B. Otherwise, the integration of both notations could not be understood or valued. From the results of the survey (Razali, 2007), it has been found that it is difficult if not impossible to obtain the understanding of the notations used in both UML and B at the same time (Property: Learnability of UML-B). Even though the users have been exposed to UML and B for some time, a level of mental burden still occurs during the process (Property: hard mental operations dimension). Having to think, integrate, and harmonize two styles of modeling from two different methods seems to be problematic.

The model transformation provided by U2B also requires some learning (Property: Learnability of UML-B). A UML-B model, in essence, carries two types of semantics: explicit B syntax specified by the users in the UML diagrams that U2B transforms as it is in the B model, and implicit B syntax that U2B implies and generates automatically from the diagrams. For example, behaviors of the operations have to be specified by the users using the B syntax in the UML diagrams whereas classes and associations in the diagrams are translated automatically as the respective sets and variables in the B model. Users have to understand these transformations and why they are accomplished in such ways (Property: Learnability and Utility of U2B; hidden dependencies dimension), since it affects the way they should do the modeling (Property: closeness of mapping dimension). Moreover, learning of how to do modeling in Rational Rose is also required (Property: Learnability of UML-B).

Modeling the UML diagrams is regarded as quite straightforward (Property: role expressiveness-diagram dimension; error proneness-diagram dimension), which eases the process of describing what is intended (Property: diffuseness dimension; closeness of mapping dimension). Despite the fact that B modeling imposes some task ordering and requires users to define and group things beforehand, the diagrams have somehow diluted the effects (Property: premature commitment dimension; abstraction gradient dimension). Perhaps these factors help to explain why a UML-B model is seen as more approachable than a B model and, thus, UML-B is preferred for formal modeling (Property: Operability and Attractiveness of UML-B).

On the other hand, specifying the UML diagrams with the correct formal semantics is perceived as difficult and error-prone (Property: error proneness-syntax dimension; hard mental operations dimension). Shallow understanding of how the formal semantics should work with the UML diagrams, lack of comprehensive documentation on the method (Property: Usefulness of Documentation), and the need to grasp the underlying principles of the employed methods and tools mentioned above have downgraded the operability of the method (Property: Operability and Attractiveness of UML-B). To attract new users to the method, a more comprehensive documentation should be readily available (Property: Future Improvement). The documentation should cover more of the practical aspects of the method and its tools (Property: Usefulness of Documentation), rather than just theory. Currently, the available documentation on the method is not helping the users much in this aspect (Property: Accessibility of UML-B)

Category 4: Functionality of Notations. Rational Rose provides specification windows in each diagram for specifying the semantics. There are two types of diagrams involved in

UML-B, thus the users are provided with two types of specification windows. One is in the Class diagram and the other is in the Statechart diagram. Regardless of the location, U2B is able to extract the semantics and treat them accordingly as a B model.

The semantics in the Statechart diagram are transformed as a nested condition under the primary condition, which is obtained from the Class diagram. In many cases, the semantics of the Statechart diagram can also be placed directly in the specification windows of the Class diagram. If the users know the states and transitions involved in the operations, they can specify it literally as a series of conditions in the specification windows of the Class diagram. Despite providing an alternative in modeling, the flexibility somehow has made the role of the semantics in the Statechart diagram, or even the Statechart diagram itself, unclear to some users (Property: role expressiveness-diagram dimension; role expressiveness-syntax dimension). The users seem to prefer specifying the full semantics in the Class diagram, since it is more obvious and straightforward. Such a process could also reduce the mental burden of having to work with two different diagrams at the same time (Property: visibility and juxtaposibility dimension; hard mental operations dimension). Moreover, the generated nested conditions from the Statechart diagram tend to complicate the B model. Because the only end product that actually matters is the transformed B model, users prefer to have a simple and quick solution to achieve it.

More clear roles and boundaries should be set between the formal semantics of the Class diagram and the Statechart diagram. The explanation of the roles and responsibilities of each part of the diagrams and semantics should be stated succinctly in the documentation, which is currently lacking in the method (Property: Usefulness of Documentation). It may be better if some principles and controls can be placed on how a UML-B model should be modeled. Although it may reduce the flexibility in modeling, it could at least guide the users based on what should and should not be done. It can also avoid redundancy. This is particularly true for new users, who often have no idea how to start and pursue the modeling. Furthermore, the transformation of formal semantics from the Statechart diagram to a B model could be smoothed further so that no unnecessary complication is introduced to users.

Discussion

The data from the survey suggest that UML-B is appealing to users who opt into B modeling while yet prefer working with standard development style of UML. This is particularly true when users are familiar with UML and have the capacity to appreciate what formal notations, such as B, could offer. The graphical modeling environment alleviates the difficulty of developing a formal model from scratch by stimulating the formulation of ideas through the use of visual objects at the abstraction level. On the other hand, users are faced with the challenge of having to grasp the underlying principles of each unique notation, as well as to understand how both notations work together to achieve the integration objectives. Each notation's roles and functionality at different parts of a model should be understood, which can easily be achieved only if the distinction between them is clear. Users are also required to learn and become familiar with the individual tools that accompany each notation, which in general should provide the necessary support.

Based on the findings, the survey generated the following tentative theories of the usability of integrated methods that combine semiformal and formal notations. The categories that contribute to the formulation of the theories are stated in the parentheses.

> **Theory 1:** The integration of semiformal and formal notations requires the understanding of principles and roles of both notations as well as the rules of the integration. The principles, roles, and rules ought to be obvious to users (Categories 3 and 4).
>
> **Theory 2:** The integration of semiformal and formal notations requires strong support from the environment. Supporting tools and comprehensive documentation should be not only available but also useful, easy-to-learn, and easy-to-use (Categories 1, 2, and 3).

Unlike the other categories, Category 1: Model Structure and Organization is not explicitly stated in the theories, although it is included. It is indirectly implied in Theory 2 with a similar effect as Category 2: Availability and Usefulness of Supporting Tools. This is because the incident may depend on the environment by which the method is supported (Rational Rose). Perhaps only the current environment has the problem of managing scattered information and multiple windows. As the data are quite limited, more observation is required on this aspect, particularly within different environments.

In terms of the CD framework, goals for designing integrated methods such as UML-B were identified. The design goals were proposed based on the nature of semiformal and formal notations, and the motivation behind the integration. The individual notations (semiformal and formal) have their own strengths and weaknesses, which are enhanced through the integration effort. In addition, the design goals were based on the common types of user activity involved in using such methods. In general, there are two major user activities: exploratory design, where users implement such methods to create a new model, and modification, where users use the methods to make changes and enhancements to an existing model.

Table 2 illustrates the recommended CD profile for designing methods that combine semiformal and formal notations. The profile proposes the desired level for each dimension that integrated methods and their notations (a combination of semiformal and formal) should aim to achieve after the integration. The *High* and *Low* indicate whether the dimension should be increased or reduced respectively, when such methods are designed. For example, method designers are recommended to aim at increasing progressive evaluation and reducing hidden dependencies. The *Moderate* indicates that although the dimension is desired at a certain level (High or Low), it may be traded off to suit more important dimensions or the two user activities. For instance, secondary notation is very useful for a Modification activity since it provides users with additional informal information. It thus may be needed (High) to improve the model comprehensibility, especially for formal (mathematical) models. However, secondary notation may cause exploratory design activity to be a bit cumbersome, because users are obliged to provide informal information about the elements in the model in addition to the official notation. Moreover, the two user activities require a model to be less resistant to change (low viscosity). By having secondary notation, any alterations to the model can be difficult because the changes are also required for the additional information. Therefore, secondary notation may be traded off (Moderate instead of High) for achieving low viscosity and facilitating the two activities. Diffuseness may need to be traded off (Moderate instead of Low) for achieving low premature commitment. Premature commitment is one dimension that designers may aim to reduce because it can be problematic for both exploratory design and

Table 2. Proposed CD Profile for Designing Integrated Methods of Semiformal and Formal Notations.

Dimension	Desired Level
Abstraction gradient	Low*
Closeness of mapping	High*
Consistency	High**
Diffuseness	Moderate (instead of Low)*
Error-proneness	Low*
Hard mental operations	Low*
Hidden dependencies	Low
Premature commitment	Low*
Progressive evaluation	
Provisionality	High
Role-expressiveness	High*
Secondary notation	Moderate (instead of High)
Viscosity	Low
Visibility/Juxtaposibility	High

Note: High means to increase; Low means to reduce; Moderate suggests a possible trade-off among dimensions;
*Semiformal notations support formal notations to achieve the desired level (otherwise, the level will be opposite);
**Formal notations support semiformal notations to achieve the desired level (otherwise, the level will be opposite).

modification activities. To reduce the need for users to look ahead and make a decision before sufficient information is available during the activities, the notation may need to be verbose, or fuller. It is up to method designers to decide the best compromise based on their methods' context of use and needs.

There are dimensions that specifically affect a particular notation more than the other. By integrating the notation with the other notation, it is believed that its usability can be improved. A single asterisk in Table 2 indicates a dimension that affects formal notations, which semiformal notations help to reduce the effect. On the other hand, two asterisks denote a dimension that semiformal notations lack, which formal notations help to overcome. For example, it is generally known that formal notations such as B syntax involve high, hard mental operations, which causes comprehension difficulties. The use of intuitive graphical symbols in semiformal notations with formal notations often reduces the effect. Similarly, semiformal notations in general lack mechanisms for a systematic progressive evaluation, which formal notations can normally offer. Without such interplay between the two types of notations, the integration is not worth the effort. After all, the motivation of such integrated methods is to allow one notation's limitations to be compensated by the strengths of the other. The following paragraphs elaborate how both notations cooperate to achieve the desired level for dimensions other than those described above.

Abstraction gradient: Formal notations impose abstractions, since users need to define and group elements into logical entities (High). Moreover, to reduce viscosity, users may need to

introduce abstractions so that any changes required would be easier. Integrating the graphical symbols of semiformal notations with formal notations may alleviate the effect, since the grouping of elements becomes more apparent (Low).

Closeness of mapping: The mapping of a problem domain is not quite straightforward using formal notations, due to the notations' unfamiliar symbols and underlying rules of interpretation (Low). The graphical symbols in semiformal notations may however facilitate the mapping, as they generally resemble objects in the real world (High).

Consistency: The formality in formal notations enforces a consistency that semiformal notations solely could not assure (Low). Semiformal notations together with formal notations could enable a consistent graphical formal model to be developed (High).

Diffuseness: The textual aspect of formal notations that is similar to natural language may cause a description to be fuller. In contrast, the graphical symbols in semiformal notations could normally carry meanings in simpler forms. The combination of textual and graphical symbols may enable the description to be short and precise (Low or Moderate).

Error-proneness: The unfamiliar mathematical symbols in formal notations frequently induce mistakes (High). The accessibility of graphical symbols in semiformal notations may reduce the tendency of making errors (Low).

Premature commitment: Formal notations normally require users to look ahead in order to obtain the right abstractions (High). Incorporating the graphical symbols of semiformal notations into formal notations may reduce the effect, since they permit the visualization of possible interacting entities (Low).

Role-expressiveness: The roles of mathematical symbols in formal notations are not so obvious to many users due to their complex interpretation rules (Low). On the other hand, the graphical symbols in semiformal notations are mainly intuitive. By combining the graphical symbols together with the mathematical symbols, users may be helped to grasp the roles of the latter (High).

The remaining dimensions without a single or double asterisk in Table 2 involve factors other than the notations used. The dimensions are provisionality, hidden dependencies, secondary notation, viscosity and visibility/juxtaposibility. Based on the findings of the survey, it is believed that the environment in which the notations reside plays a major role in achieving the desired levels for these dimensions. This environment includes the structure of the model and the tools that support the notations. This claim is worth investigating in future.

The tentative theories and the proposed CD profile may not be conclusive, and they should be validated and refined further in future investigations. However, they can act as the first step in understanding the nature of integrated methods such as UML-B and provide a meaningful guide to better design.

Validity

Threats to validity are influences that may limit the ability to draw conclusions from the data. The following paragraphs discuss some threats of this survey.

Selection of Respondents. The respondents were students in the university where the research was conducted. Therefore, their answers might have been biased (positively or

negatively). On the other hand, the respondents were considered the most appropriate candidates for this study because they have been trained on B and UML-B. This knowledge is necessary for using UML-B. In fact, the participants also had some experience in using UML-B and thus were able to contribute more fully to the survey. Moreover, they were independent users, who had no personal interest with the technologies involved or direct contact with the research. To reduce the threat, the subjects were advised to give opinions and comments as sincerely as possible.

Students as Respondents. The respondents of this survey were students. They may have not represented software developers, since they are less experienced. However, the respondents were in the final semester of their master's program and had a reasonable amount of experience and knowledge of software development. Half of the students had some professional working experience. Thus they were seen as valid respondents for the survey as new users with developer's experience.

Sample Size and Response Rate. The survey questionnaire was distributed to all 14 master's students of software engineering at the University of Southampton who registered for the Critical Systems course in spring 2006. Thirteen students responded to the survey. Due to a technical problem, only 10 responses were considered for analysis. Although the number was quite small, a response rate of 70% was considered appropriate for an initial attempt. Moreover, as a qualitative study, the quality of the data is the focus, rather than strictly the quantity. Brief identity screening was done on the four students who were not included. No particular pattern was identified that could have potentially biased the results.

Non-committal Responses. Using an uneven number of levels for the ordinal scale leaves open the possibility of noncommittal responses, with the medians representing "neither –nor" or "not sure." Although such incidents could be seen in the data, they did not happen often and no pattern was detected in either the questions or by respondents.

Toy Problem. Due to time and resource constraints, the modeling task given to the respondents was not large and may have not represented real software systems. However, the task was believed to be sufficient for the respondents to experience modeling using UML-B. In fact, the task required the respondents to explore most of the functionality provided by the method.

Analysis Process. The grounded theory approach encourages the gathering of further data after analyzing the first gathered data. In fact, data collection and analysis should be repeated several times so that more incidents are captured and validated until the theory saturates (Strauss & Corbin, 1998). Due to time and resources constraints, the data collection and analysis for the survey were conducted only once, and the findings presented here reflect one set of data. However, the survey will be repeated in the future.

Nature of Study. Surveys and qualitative measures by their nature are retrospective. Therefore, there was a risk that the respondents reported based on what they thought they did rather than what they actually did. Advising the respondents to complete the survey questionnaire as soon as they completed the modeling task could have reduced this threat, because the respondents would have had a clearer memory of what they found during the task. The respondents submitted the questionnaire together with their completed models at the end of the course.

Heterogeneity of Respondents. The respondents might have different abilities and experiences. Thus, there was a risk that the results might have been affected by individual differences. As a qualitative study, the variation however could provide richer data for the analysis.

Familiarity of Respondents. The respondents were taught formally on B for about 9 hours and on UML-B for 1 hour. They were then required to complete a modeling task using UML-B within a month period. The results may have been different if the respondents were given more time and training. The aim of the survey was to capture the experience of using UML-B from new users' perspectives. Therefore, the allocated time frame and training were seen as adequate and realistic for the purpose of this research. The results may also have been influenced by the respondents' knowledge of UML obtained from their previous working experience and studies, which varied considerably.

CONCLUSION

This paper has presented a survey conducted on a method that integrates the use of semiformal and formal notations, namely UML-B. The survey assessed the usability of the notation used in the method and its modeling environment by using the CD framework with several usability criteria suggested by the ISO. The data analysis was conducted using the grounded theory approach. The findings indicated that the dual characteristics of the method bring to users several implications, both positive and negative. Combining semiformal and formal notations allows the potential of individual notation to be strengthened, while each notation's limitations can be compensated by the other. However, the integration, in essence, brings to the designers the loads of two individual notations, which are actually quite different in many ways. Users therefore need strong support from the environment to lessen the burden that lies beneath the integration effort. The support involves not only the tools that aid the modeling process but also resources for learning the method. Based on the findings, we proposed a usability profile based on CD for designing integrated methods such as UML-B.

Some of the findings of the investigation are now being fed into the next generation of UML-B development[2]. The findings of the survey can be improved further by extending the survey to a large number of users. This will help enhance the current understanding of the method and discovering other factors that might affect its use. The tentative theories and the proposed CD profile of integrated methods (combined semiformal and formal notations) discussed in this paper can also be validated and refined further by applying them to examine other similar methods. This allows the derivation of more concrete theories and guidelines that can be used to design and improve the usability of such methods in future.

ENDNOTES

1. Electronics Computer Science (ECS), COMP3011 Critical Systems, http://www.ecs.soton.ac.uk/syllabus/COMP3011.html
2. EU Framework VI project: Rigorous Open Development Environment for Complex Systems (RODIN) http://rodin.cs.ncl.ac.uk/

REFERENCES

Abrial, J. R. (1996). *The B-Method: Assigning programs to meanings*. Cambridge, UK: Cambridge University Press.

Alexander, P. (1996). Best of both worlds (formal and semi-formal software engineering). *IEEE Potentials*, *14*, 29–32.

Bauer, M., & Johnson-Laird, P. (1993). How diagrams can improve reasoning. *Psychological Science*, *4*, 372–378.

B-Core Limited [B-Core]. (2002). The *B-Toolkit*. Retrieved April 18, 2008, from http://www.b-core.com/ONLINEDOC/BToolkit.html

Blackwell, A. F., & Green, T. R. G. (2000). A cognitive dimensions questionnaire optimised for users. In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group* (PPIG '00; pp. 137–154). Cosenza, Italy: Memoria.

Blackwell, A., & Green. T. (2003). Notational systems: The cognitive dimensions of notations framework. In J. M. Carroll (Ed.), *HCI models, theories and frameworks: Toward a multidisciplinary science* (pp. 103–134). San Francisco: Morgan Kaufmann.

Bonissone, P. (1982). A fuzzy sets based linguistic approach: Theory and application. In M. Gupta & E. Sanchez (Eds.), *Approximate reasoning in decision analysis* (pp. 329–339). New York: North-Holland Publishing Company.

Carew, D., Exton, C., & Buckley, J. (2005). An empirical investigation of the comprehensibility of requirements specifications. In G. Kadoda (Ed.), *Proceedings of the 4th International Symposium on Empirical Software Engineering* (ISESE '05; pp. 256–266). Noosa Heads, Australia: IEEE Computer Society.

Cassell, C., & Symon, G. (1994). *Qualitative methods in organizational research.* Thousand Oaks, CA, USA: Sage.

Chandler, P., & Sweller, J. (1992). The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology*, *62*, 233–246.

Chen, P. (1976). The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, *1*, 9–37.

Clarke, S. (2001). Evaluating a new programming language. In G. Kadoda (Ed.), *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group* (PPIG '01; pp. 275–289). Bournemouth, UK: Bournemouth University.

ClearSy Systems Engineering [ClearSy]. (n.d.). *Atelier B, the industrial tool to efficiently deploy the B Method.* Retrieved April 18, 2008, from http://www.atelierb.eu/index_en.html

Cox, K. (2000). Cognitive dimensions of use cases: Feedback from a student questionnaire. In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group* (PPIG '00; pp. 99–122). Cosenza, Italy: Memoria.

Denzin, N., & Lincoln, Y. (1994). *Handbook of qualitative research.* Thousand Oaks, CA, USA: Sage.

Glaser, B. (1992). *Basics of grounded theory analysis: Emergence vs. forcing*. Mill Valley, CA, USA: Sociology Press.

Glaser, B. G., & Strauss, A. L. (1967). *The discovery of grounded theory: Strategies for qualitative research*. London, UK: Weidenfeld and Nicolson.

Green, T. R. G. (1989). Cognitive dimensions of notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and computers V* (pp. 443–460). Cambridge, UK: Cambridge University Press.

Green, T. R. G., & Blackwell, A. F. (1998, September). Design for usability using cognitive dimensions. Tutorial session at the *British Computer Society Conference on Human Computer Interaction* (BCS-HCI '98). Sheffield, UK.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A cognitive dimensions framework. *Journal of Visual Languages and Computing*, *7*, 131–174.

Hinchey, M. G. (2002). Confessions of a formal methodist. In P. A. Lindsay (Ed.), *Proceedings of the 7th Australian Workshop on Safety-Related Programmable Systems* (SCS '02; pp. 17–20). Adelaide, Australia: Australian Computer Society.

IBM Software [IBM]. (n.d.). *Rational Rose*. Retrieved April 18, 2008, from http://www-306.ibm.com/software/awdtools/developer/rose/index.html

International Organization for Standardization [ISO]. (2003, July). *Software engineering, product quality—Part 3: Internal metrics* (Standard No. 9126-3). Geneva, Switzerland: ISO.

International Organization for Standardization [ISO]. (2004, March). *Software engineering, product quality—Part 4: Quality in use metrics* (Standards No. 9126-4). Geneva, Switzerland: ISO.

Kitchenham, B. A. & Pfleeger, S. L. (2002). Principles of survey research: Part 3: Constructing a survey instrument. *SIGSOFT Software Engineering Notes, 27*(2), 20–24.

Kutar, M., Britton, C., & Barker, T. (2002). A comparison of empirical study and cognitive dimensions analysis in the evaluation of UML diagrams. In J. Kuljis, L. Baldwin, & R. Scoble (Eds.), *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group* (PPIG '02; pp. 1–14). Brunel, UK: Brunel University College.

Martin, S. (2003). The best of both worlds integrating UML with Z for software specifications, *Journal of Computing and Control Engineering, 14,* 8–11.

Microsoft Corporation [Microsoft]. (2008). Visual C# Developer Center. Retrieved April 18, 2008, from http://msdn.microsoft.com/vcsharp/

Object Management Group [OMG]. (2008). *Introduction to OMG's unified modeling language (UML).* Retrieved April 18, 2008, from http://www.omg.org/gettingstarted/what_is_uml.htm

Pender, T. (2003). *UML Bible*. Indianapolis, IN, USA: Wiley.

Razali, R. (2007). *UML-B Survey questionnaires and responses*. (Electronics and Computer Science, University of Southampton Tech. Rep., ID code 13322). Retrieved April 18, 2008, from http://eprints.ecs.soton.ac.uk/13322

Razali, R., Snook, C. F., Poppleton, M. R., Garratt, P. W., & Walters, R. J. (2007). Experimental comparison of the comprehensibility of a UML-based formal specification versus a textual one. In B. Kitchenham, P. Brereton, & M. Turner (Eds.), *Proceedings of the 11th International Conference on Evaluation and Assessment in Software Engineering* (EASE '07; pp. 1–11). Keele, UK: British Computer Society.

Snook, C., & Butler, M. (2006). UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, *15*(1), 92–122.

Spivey, J. M. (1992). *The Z notation: A reference manual* (2nd ed.). Englewood Cliffs, NJ, USA: Prentice-Hall.

Strauss, A. L., & Corbin, J. (1998). *Basics of qualitative research: Techniques and procedures for developing grounded theory* (2nd ed.). Thousand Oaks, CA, USA: Sage.

Sweller, J., & Chandler, P. (1994). Why some material is difficult to learn. *Cognition and Instruction, 12,* 185–233.

Triffitt, E., & Khazaei, B. (2002). A study of usability of Z formalism based on cognitive dimensions. In J. Kuljis, L. Baldwin, & R. Scoble (Eds.), *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group* (PPIG '02; pp. 15–28). Brunel, UK: Brunel University College.

Tukiainen, M. (2001). Evaluation of the cognitive dimensions questionnaire and some thoughts about the cognitive dimensions of spreadsheet calculation. In G. Kadoda (Ed.), *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group* (PPIG '01; pp. 291–301). Bournemouth, UK: Bournemouth University.

van Lamsweerde, A. (2000). Formal specification: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 147–159). New York: ACM Press.

Westbrook, L. (1994). Qualitative research methods: A review of major stages, data analysis techniques, and quality controls. *Library and Information Science Research, 16,* 241–245.

Authors' Note

The authors gratefully acknowledge the COMP3011 (spring 2006) students who participated in this study.

All correspondence should be addressed to:
Rozilawati Razali or Colin Snook
Dependable Systems and Software Engineering Group (DSSE)
School of Electronics and Computer Science (ECS)
University of Southampton
SO17 1BJ United Kingdom
rr04r@ecs.soton.ac.uk or rozila_razali@yahoo.co.uk or cfs@ecs.soton.ac.uk

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 47–61

SPATIAL ABILITY AND LEARNING TO PROGRAM

Sue Jones
School of Computer Science
University of Nottingham
Nottingham, UK

Gary Burnett
School of Computer Science
University of Nottingham
Nottingham, UK

Abstract: *Results in introductory computer programming modules are often disappointing, and various individual differences have been found to be relevant. This paper reviews work in this area, with particular reference to the effect of a student's spatial ability. Data is presented on a cohort of 49 students enrolled on an MSc in Information Technology course at a university in the UK. A measure was taken of their mental rotation ability, and a questionnaire administered that focused on their previous academic experience, and expectations relating to the introductory computer programming module they were studying. The results showed a positive correlation between mental rotation ability and success in the module (*r *= 0.48). Other factors, such as confidence level, expected success, and programming experience, were also found to be important. These results are discussed in relation to the accessibility of programming to learners with low spatial ability.*

Keywords: *spatial skills, programming ability, individual differences.*

INTRODUCTION

Results in introductory computer programming modules are often disappointing (Mancy & Reid, 2004), with reports of up to 30% of students failing to complete them (Guzdial & Soloway, 2002). Students who perform well in other subjects may not achieve equivalent success in programming tasks (Byrne & Lyons, 2001; Fincher et al., 2005), lose confidence, and give up computer science courses. Irrespective of experience, some programmers appear to be more skilled than others. Curtis (1981) found a range of performance scores of 23 to 1 in a debugging exercise, and Shneiderman (1980) reported differences in performance of 100 to 1 among programmers of similar programming experience.

Previous research has focused on why some students underperform in programming. Various individual differences have been implicated in programming success, with debate over the relative importance of each of these factors. This paper will focus on spatial ability,

URN:NBNfi:jyu-200804151352

but also considers how some of the other individual differences are related in their influence on learning to program. The next two sections will focus on a literature review of the work in this area. Then follows an analysis of data collected in a study carried out over the academic year 2005/06 at the University of Nottingham. Finally, the results will be discussed in relation to programming achievement.

SPATIAL ABILITY AND MENTAL MODELS

One individual difference considered to have some relevance to programming aptitude is spatial ability. This is a heterogeneous cluster of skills considered to be a dimension of intelligence distinct from verbal, mathematical, and reasoning skills. Halpern (2000) defines spatial ability as a cognitive characteristic that gives a measure of the ability to conceptualize the spatial relations between objects. As this is a broad cognitive concept, various categorizations have been suggested to help organize our understanding of this area, one of these being mental rotation. Mental rotation is the capacity to accurately picture the rotation of two- or three-dimensional objects in the mind, and some researchers believe that it is a good measure of a general spatial reasoning ability (Halpern, 2000). The Vandenberg and Kuse (1978) Mental Rotation Test is the standard test for this skill.

Spatial ability has been shown to be important for navigation in the real world, and in an abstract information space such as hypertext (Jones & Burnett, 2007b). It is also considered by some to be an important determinant in program comprehension, due in part to source code being likened to a multidimensional virtual space that requires similar skills for navigation as those utilized in a real environment (Cox, Fisher, & O'Brien, 2005). However, there are few studies looking at relations between spatial ability and individual programming performance. Mayer, Dyck, and Vilberg (1986) showed that success in learning Basic was related to spatial ability ($r = 0.31$, $p < 0.05$). Fincher et al. (2005) showed an overall small positive correlation between performance in a spatial visualization test and marks achieved in the introductory programming courses at 11 institutions ($r = 0.17$, $p = 0.047$). Both of these studies used versions of the Paper Folding Test, designed to measure spatial visualization, one of the various types of task included in the broad cognitive category of spatial ability (Halpern, 2000). The Paper Folding Test requires subjects to imagine the result after folding a paper object, but this process does not require mental rotation (Velez, Silver, & Tremaine, 2005).

Webb (1984), studying children between the ages of 11 and 14, found a relationship between spatial ability and various programming components of a short course in Logo programming, with an average correlation of 0.63 ($p < .001$). The only pretest measure to give a stronger correlation was the score on a mathematics reasoning test. Webb utilized three measures of spatial ability, one of which was the Paper Folding Test, but the others requiring mental rotation of figures. Fisher, Cox, and Zhao (2006) used the Vandenberg and Kuse (1978) Mental Rotation Test to study correlations with a software maintenance task for a short Java program. While they found a high correlation for the men ($r = 0.63$, $p = .012$), this was not reflected in the women's results.

A related factor in the role of spatial ability to programming success is the development of mental models. Mental models are variously defined, but in the context of this paper are considered as predictive representations or abstractions of a program. In recent work, Jones

and Burnett (2007a) demonstrated differences in the navigation of source code in a code comprehension exercise, with individuals with higher spatial ability jumping between functions more frequently and making more interclass jumps (moving between files). The authors speculate that this style of navigation may allow a better mental model of the program to be formed, thus aiding comprehension. Mental models of spatial information are called cognitive maps (Downs & Stea, 1973), and people build these while familiarizing themselves with an environment. They become disorientated if this internal map does not correspond to the physical representation of the environment (Westerman & Cribbin, 1999). In relation to cognitive maps of program code, Fisher et al. (2006, p. 1) use the term "codespace," and define it as "a programmer's mental model of source code with respect to the perceived spatial attributes of entities identified within the code." Hence the mental model is an abstraction of the program formed from piecing together various kinds of information extracted while navigating the source code. Wiedenbeck, LaBelle, and Kain (2004) stress the importance of a good mental model in program understanding.

It has been argued that because source code is linear, mental rotation may not be as important as other types of spatial ability for navigation in programming environments (Fisher et al., 2006). However, the studies utilizing mental rotation as a measure of spatial ability appear to show stronger relations with measures of programming aptitude than those using the Paper Folding Test as a measure of spatial visualization. Kimura (1999) makes the link between success in mental rotation and the capacity to build a mental model, or cognitive map, of an environment. She suggests that good mental rotation capacity enables us to recognize a scene from different angles, and thus to retrace a route in reverse when returning to a destination, or piece together other bits of information to devise a new route back. There is an increasing recognition of the importance of mental models to learning programming (Wiedenbeck et al., 2004), and perhaps a good mental rotation capacity is allowing formation of more accurate mental models of programs.

One study looked at the map-drawing styles of students to determine if this was a predictor of success in the introductory programming courses they were studying (Tolhurst et al., 2006). Students were required to sketch maps of a given real world environment, and the maps were then grouped according to the landmark, route, survey model for the acquisition of spatial knowledge (Werner et al., 1997). When compared to the marks achieved in the course, they found a trend for the high achievers to draw survey maps, while those who sketched route maps performed less well but better than those who produced landmark maps. The authors speculate that programming ability is related to an ability to navigate through the information space using the same skills as in the real world. Good spatial ability has been related to the development of survey knowledge, equivalent to a well-formed cognitive map of an environment (Cutmore, Hine, Maberly, Langford, & Hawgood, 2000).

RELATION OF SPATIAL ABILITY TO OTHER INDIVIDUAL DIFFERENCES

This section reviews the literature on the impact of various individual differences on programming performance. The main focus will be the interactions with spatial ability.

Gender

Girls are underrepresented in university computer science (CS) courses (Byrne & Lyons, 2001). Suggested reasons for this include sex stereotyping and males' greater exposure to computers and computer games (Mumtaz, 2001; Schumacher & Morahan-Martin, 2001). Boys appear to have a more confident, positive attitude toward computers (Durndell & Haag, 2002). Another reason is that spatial skills are crucial to most video and computer games as well as many computer applications, and repeated practice may actually enhance spatial skills (Subrahmanyam, Greenfield, Kraut, & Gross, 2001). There is a wealth of evidence of gender differences in spatial ability, with females appearing to underperform in certain measures, such as mental rotation (Voyer, Nolan, & Voyer, 2000).

It is difficult to study gender differences in programming style and ability due to the fact that many females, prior to university admission, have already chosen not to take CS courses for the reasons mentioned above (Scragg & Smith, 1998). Byrne and Lyons (2001) found no significant difference in performance between male and female students in a first-year programming module, possibly because the module was part of a Bachelor of Arts honors degree program with a preponderance of female students (61%). However, other studies also have not found the expected gender difference (Bergin & Reilly, 2005; Rountree, Rountree, & Robins 2002), perhaps because group-based differences such as gender have less effect on an individual's performance than individual differences such as spatial ability or cognitive style (Beckwith & Burnett, 2004). Fisher et al. (2006) hypothesize that females prefer a more low-risk, bottom-up approach to program development and comprehension, and males a more high-risk, abstract, top-down approach. Bradley (1985) demonstrated that top-down processing was positively related to Logo programming success.

Self Efficacy/Comfort

Self-efficacy relates to how we estimate our capability to perform well in a certain context (Bandura, 1986). A person with high self-efficacy is more likely to undertake challenging tasks, expend more effort to achieve them, and demonstrate persistence when difficulties arise. Rountree et al. (2002) surveyed students early in a first-year computer science course and found that students' expectation of how they were going to perform was the biggest indicator of success. Surprisingly, students predicted the outcomes very early in the course, and this may contribute to their level of motivation and persistence required to achieve. Closely related to self-efficacy is comfort level, based on our perception of the degree of difficulty of a task, which affects our anxiety levels. Studies have found that comfort level, derived from questionnaires, was strongly predictive of programming performance (Bergin & Reilly, 2005; Wilson & Shrock, 2001). Students with low computer experience are likely to be less confident and more anxious when starting a programming course (Byrne & Lyons, 2001). It has also been suggested that males tend to show greater self-efficacy and lower computer anxiety than females (Beckwith & Burnett, 2004; Durndell & Haag, 2002). Having a good mental model increases self-efficacy by enabling program comprehension (Wiedenbeck et al., 2004).

Previous Academic Exposure

Previous programming experience seems to relate to success in introductory programming courses (Rountree et al., 2002; Wilson & Shrock, 2001), with Wiedenbeck et al. (2004) linking this with self-efficacy. Boys are more likely than girls to have previous programming experience (Bruckman, Jensen, & DeBonte, 2002). Good performance in mathematics is also relevant, and is often an entry requirement for computer science, with the belief that the skills required for solving mathematics problems are similar to those needed for programming tasks (Byrne & Lyons, 2001). Various studies have found relations between mathematics results and success in learning programming (Webb, 1984; Werth, 1986; Wilson & Shrock, 2001). Byrnes and Lyons (2001) found a relationship between results in secondary school mathematics examinations and results from a first-year programming course ($r = 0.353$, $p < 0.01$). The correlation with science results was even stronger ($r = 0.572$, $p < 0.01$). There was no correlation between the English or foreign language results and programming achievement. Others have found a similar relationship with science (Bergin & Reilly, 2005; Werth, 1986).

The ability to succeed in mathematics has been related to spatial ability (Pease & Pease, 2001), and one study showed that males outperformed females in both the Vandenberg and Kuse (1978) test for spatial ability and a mathematics aptitude test, with mental rotation predicting mathematics aptitude for the female samples (Casey, Nuttall, Pezaris, & Benbow, 1995). When mental rotation ability was statistically adjusted for, the gender difference in mathematics achievement was eliminated in most of the groups studied.

Cognitive Style

Cognitive abilities are specific to a particular domain of content or function, such as verbal, numerical, or spatial ability. A measure can be taken of an individual's spatial ability as separate from their verbal reasoning score—one may be high, the other low. In contrast, cognitive styles cut across these domains, and have more to do with organization and control of cognitive processes. Consequently, there appears to be an interaction between cognitive abilities and styles, with field-dependency being the style most associated with spatial ability (Chen, Czerwinski, & Macredie, 2000). McKenna (1984) presents arguments debating whether field dependence, often measured by the Embedded Figures Test (EFT), is a cognitive style or cognitive ability. The EFT requires participants to locate a given simple shape embedded within a larger complex one. Because the EFT is timed, it has been argued that it is more a measure of cognitive ability than style, assessing differences in level of, rather than manner of, performance. McKenna reviews work showing there is a strong relation between the EFT and spatial ability.

Various studies have demonstrated some impact of field dependency on programming achievement. Bishop-Clark (1995) carried out a review of some of the work in this area. She concluded that the results are not consistent (correlations ranging from .08 to .80), but that field independence appeared to be positively related to programming success. Mancy and Reid (2004) compared field dependency and marks in various assessments on an introductory programming course. Field dependency was measured using the EFT, and the results showed positive correlations with marks on the different assessments, with a good correlation ($r = 0.40$) with the final examination.

In summary, individual differences with varying degrees of impact on programming performance have been discussed in relation to spatial ability (see Figure 1). However, there are only a small number of adult studies looking at mental rotation in this context. The following study aimed to gather extra data to supplement the research knowledge regarding any effect of spatial ability, and other interacting factors, on programming performance.

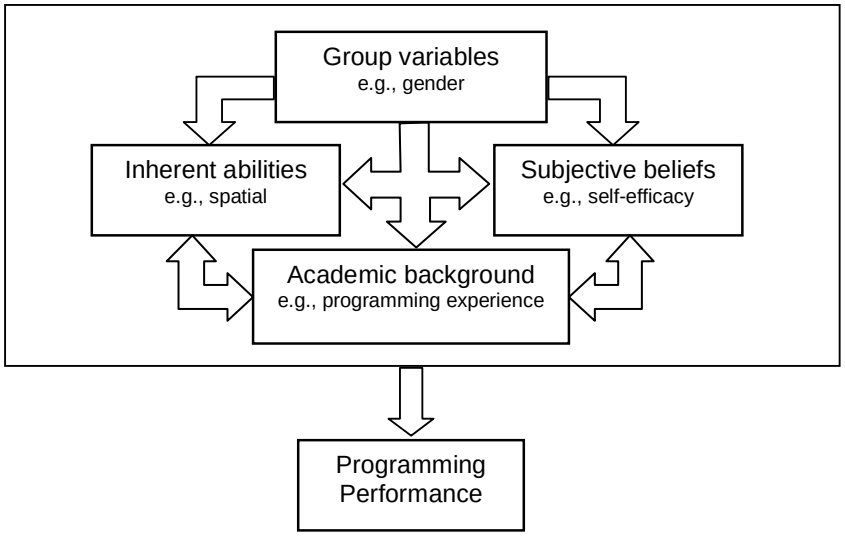

Figure 1. Potential interactions between individual differences and programming performance.

THE STUDY

During the academic year 2005/06, a test of mental rotation ability and a questionnaire were administered to a cohort of university students. This section will focus on the results of our analyses to ascertain any relations between the variables studied.

Data Collection

Participants consisted of 49 volunteers (average age 26 years) from students enrolled in a one-year master's conversion course (meaning students did not have computer science as their first degree) at the University of Nottingham, UK. An introductory programming module (ICP) was compulsory for all students in the first semester. The course consisted of two streams. Students in the MSc in Information Technology (IT, n = 28) had their first degree in a science or engineering subject and would continue with further compulsory programming modules in the second semester. Those studying for the MSc in the Management of Information Technology (MIT, n = 21) had a first degree in a wide range of subjects (including arts and humanities), and were not required to take programming modules after the first semester. The cohort consisted of 39 males and 10 females, with only 2 females in the IT stream.

Data on the participants' academic history and perceived programming experience were collected by questionnaire. At the end of Semester 1 (December 2005), and before taking the introductory programming examination, the students were asked how they rated their

confidence levels in their last (as yet unmarked) programming coursework. They also rated the ICP in comparison to other nonprogramming modules on the parameters of difficulty, workload, and expected success, similar to Rountree et al. (2002). In addition, the final marks in the following modules were collated:

- Semester 1: compulsory modules
  - Introduction to Computer Programming (ICP). The assessment consisted of 50% coursework (2 programming assignments) and 50% examination. The language taught was Java.
  - Introduction to Human Factors (IHF), a nonprogramming module with assessment based on 25% coursework and 75% final examination.
- Semester 2
  - Object Oriented Systems (OOS), with two programming assignments contributing 50% to the final mark, and a final examination. This module was compulsory for the master's in IT. The language taught was C++.
  - Management of IT (MAN), a nonprogramming module with 25% coursework and 75% examination. This module was compulsory for those taking the master's in MIT.

The programming modules involved considerable practical programming assignments, while the nonprogramming subjects focused on issue-based discussion elements.

Individual differences in the participants' spatial skills were measured using a version of the 3D Mental Rotation Test found at Psychlab OnLine[1]. The test is a modified version of the Vandenberg and Kuse (1978) Mental Rotation (MR) test, and was customized for this study by Professor Hay of the University of Wisconsin, Milwaukee.

In the version used, the participants were asked to determine if one shape could be mentally rotated to match the orientation of a second (see Figure 2 for an example). The students were presented with 30 examples to be completed as quickly as possible, with equal emphasis being given to accuracy and speed. This was an on-line test. Once an answer was submitted, there was no recourse for correcting it, and no feedback was provided on the correctness of the answers. At the completion of the test, a file was generated with the number of correct answers, and the total time taken for completion of the 30 questions (mean time = 215s, range 59 to 452 s). A score for spatial ability was derived from the number of correct answers divided by the total time (in seconds) to complete the exercise. The final number was multiplied by 100 to provide a more usable scale.

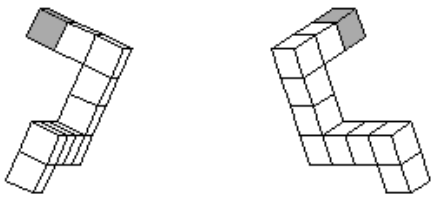

Figure 2. Example of the modified Vandenberg and Kuse (1978) Mental Rotation Test, customized by Professor Hay.

Results

Module marks and questionnaire responses were recorded. Statistical analyses were run to determine any relations between spatial ability and other individual differences.

Module Marks

There was a strong correlation between mental rotation (MR) scores and the participants' grades in the programming modules, but this was not reflected in the nonprogramming modules (see Table 1). There was a stronger correlation for the IT students (those who carried on with programming) in the ICP, and a high correlation between MR scores and results in the more advanced (OOS) programming module for these students. As can be seen in Figure 3, the spread of results in the ICP module was greater for the IT stream (30% to 95%; $M = 66.59$, $SD = 15.43$) than the MIT stream (42% to 78%; $M = 64.90$, $SD = 9.93$), although the difference in means was not statistically significant ($p = 0.66$). Similarly, the MR test scores showed a greater range among the IT students (3.09 to 30.76; $M = 16.47$, $SD = 7.22$) than the MIT students (4.18 to 22.30; $M = 11.73$, $SD = 4.84$), with the IT students scoring significantly higher ($p < 0.05$).

When the results obtained in each of the modules were compared, there was found to be a strong positive correlation between the two programming modules, ICP and OOS, and between the two nonprogramming modules, IHF and MAN. There were no correlations between the programming and nonprogramming modules (see Table 2).

Table 1. Correlation Analysis for Mental Rotation (MR) Scores and Module Marks.

		Programming modules				Nonprogramming modules	
		ICP (All)	ICP (MIT)	ICP (IT)	OOS (IT)	IHF (All)	MAN (MIT)
MR Score	*r*	0.48	0.37	0.57	0.68	0.21	0.10
	p	<0.01	<0.05	<0.01	<0.01	0.16	0.65
	n	49	21	28	27	46	21

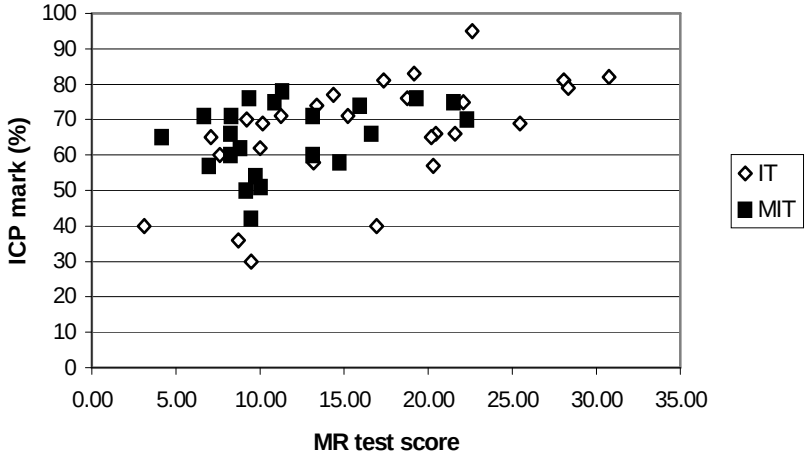

Figure 3. Scatterplot of ICP results as a function of MR test scores for the two master's streams.

Table 2. Pearson Correlation Analysis for Module Marks.

	Programming modules		Nonprogramming modules	
	ICP	OOS	IHF	MAN
ICP				
OOS	0.73 (<0.001)			
IHF	0.28 (0.07)	0.26 (0.26)		
MAN	0.10 (0.76)	N/A	0.69 (<0.005)	

Note: The significance values are in parentheses.

Questionnaire Results

Many of the multiple-choice questions had a choice of 4 or 5 answer categories, and with only 44 students completing the questionnaire, some of the categories had a very small number of respondents. To enable statistical analysis, these categories were collapsed into 2 or 3 items. This allowed *t* tests to be carried out to determine if there were differences in the means for MR test scores and ICP results for the dichotomous measures, and Kruskal-Wallis tests on the trifold measures. The dichotomous variables were also subjected to chi-square analysis to determine if there were any differences in the answers for the two degree streams, and for confidence levels.

Those with greater perceived programming experience had higher spatial scores, and performed significantly better in the ICP results (see Table 3). Similarly, those with higher confidence levels for the ICP coursework performed very well in the ICP module, and there was a trend for them to have higher MR scores (although just failing to reach statistical significance).

When students were asked to compare ICP with other nonprogramming modules, it was found that individuals rating ICP as more difficult tended to have lower spatial scores. Those rating success with ICP (relative to nonprogramming modules) as high were achieving better end results (see Table 4).

There was a nonsignificant trend for those in the IT stream to have more programming experience. Seventy-four percent claimed to be professional/intermediate, compared to only 39% of the MIT students (see Table 5).

Table 3. T-tests for Questionnaire Results.

		MR test score			ICP mark		
Parameter	Categories	Mean	SD	p	Mean	SD	p
First degree	Science	15.71	7.63	0.28	68.00	15.55	0.69
	Non-science	13.29	6.00		66.29	9.76	
Gender	Male	14.67	6.72	0.47	68.51	11.70	0.19
	Female	12.88	5.41		62.22	15.91	
Programming experience	Professional/intermediate	16.65	7.41	<0.05	72.50	9.54	<0.005
	Novice/none	12.31	5.51		60.60	14.34	
Confidence	High/moderate	15.95	7.15	0.09	72.59	8.23	<0.001
	Little/none	12.23	5.85		56.47	14.83	

Table 4. Results of the Kruskall-Wallis Analysis of Student Expectations.

		MR test score			ICP mark		
Parameter	Categories (collapsed)	Mean rank	χ^2	p	Mean rank	χ^2	p
Workload	Much less/less	33.67	4.20	0.12	26.33	2.46	0.29
	The same	26.09			27.09		
	More/much more	20.07			20.43		
Difficulty	Much less/less	38.60	10.50	<0.01	35.50	5.97	0.05
	The same	26.64			22.18		
	More/much more	18.79			20.30		
Success	Much less/less	14.60	5.01	0.08	14.00	6.10	<0.05
	The same	24.11			23.67		
	More/much more	25.63			26.50		

Table 5. Results of Chi-square on Programming Experience and Confidence.

		Masters				Confidence			
Parameter	Categories (collapsed)	IT	MIT	χ^2	p	High/ moderate	Little/ none	χ^2	p
Programming experience	Prof/intermediate	21	8	3.25	0.07	24	2	13.17	<0.001
	Novice/none	7	13			8	15		
Confidence	High/moderate	22	12	1.36	0.24				
	Little/none	6	9						

The impact of programming experience on confidence levels was obvious, with 92% of the more experienced students rating their confidence levels as high or moderate, and only 35% of the less experienced. Although failing to reach statistical significance, a larger number of the IT students (79%) rated their confidence levels for the ICP coursework as high/moderate, compared to only 56% of the MIT students.

DISCUSSION

It is known that students who perform well in other subjects may produce disappointing results in programming. In the current study, there were correlations between the results gained in programming modules and spatial scores, with no correlations being found with the nonprogramming modules. These results suggest that spatial ability, as measured by a mental rotation test, is related to success in the programming subjects, while not appearing to be of relevance in the nonprogramming modules investigated.

When the relationship between spatial ability and ICP results were viewed for the separate master's streams, the correlation was found to be higher for the IT cohort. Some of this variation between the two groups may be accounted for by the larger range of mental rotation scores and ICP results in the IT stream. Additionally, this was a self-selected group of students who have chosen to continue with programming. Although not reaching significance, there was a trend for them to have greater programming experience, so this

variable would have had less of an impact than in the wider master's cohort. Hassell (1982) showed a similar result for second- and final-year students. She looked at correlations between the EFT and measures of programming ability, and found a correlation ($r = 0.5$) for seniors, but a nonsignificant correlation for the second-year students. In the current study, there was a very strong correlation between the results for the IT students in the two programming modules, even though the courses were taught by two different lecturers. There was also a strong correlation between results in the two nonprogramming modules for the MIT students. However, there was no relationship between the programming module (ICP) and the nonprogramming module (IHF) for the group as a whole, nor between ICP and the other nonprogramming module (MAN) for the MIT students. This demonstrates that those who perform well in other subjects may underperform in programming.

As expected, these results suggest that other individual differences may have an impact on the results. Those who considered themselves to be more experienced in programming performed better in the introductory programming module. With this being a master's course, the students were generally older than undergraduate students, and may have had more opportunity for exposure to programming, either within a first-degree course, or from a previous work environment. The more experienced programmers tended to have higher spatial ability and, as expected, admitted to having greater confidence about the coursework. This confidence translated into better performance in the whole ICP module. This is confirmed by the fact that those who expected themselves to be more successful in ICP than nonprogramming modules generally performed better in the final mark. Rountree et al. (2002) found that expecting an A grade was the strongest indicator of success, with expected difficulty and workload making smaller but relevant contributions, trends reflected in the current study. Thus it would appear that self-efficacy is an important contributor to achievement. The data also show that those with low spatial ability were experiencing greater difficulty with ICP compared to the non-programming modules. There was no significant impact of a science background on the results, even though the majority of science graduates were enrolled in the IT stream. There were also no significant differences between males and females in mental rotation test score or ICP mark. This may have occurred because the sample size for the females was too low (20% of the group), a situation reflected in many computer science courses.

One other variable that needs to be considered is the programming activity itself. As shown in this study, the more experienced students had a higher spatial ability, but it is difficult to be sure if this is cause or effect. It is known that high spatial ability predisposes individuals to a choice of spatial subjects (such as engineering) and careers (such as architecture; Quaiser-Pohl & Lehmann, 2002; Smith, 1992). Consequently, it is possible that students with high spatial ability are choosing programming as an option because this skill allows them to excel. Alternatively, the very act of practicing programming may cause an increase in spatial ability. The task of learning to program has been shown to cause students to become more field independent (Cathcart, 1990), and improve their mental rotation ability (Miller, Kelly, & Kelly, 1998). However, these results were found when teaching Logo to schoolchildren; Logo programming requires children to imagine orientating with the turtle, a form of mental rotation (Miller et al., 1998). It would be interesting to give university students a pre- and posttest to see if an intensive programming course resulted in any improvement in mental rotation ability. Additionally, there is a wealth of evidence that training in the use of spatial tasks can improve scores in spatial tests, and this could be an

important exercise for students wishing to improve their inherent programming ability. The authors believe that this training should be incorporated into the school curriculum, perhaps as early as 6 years of age (Jones & Burnett, 2006).

CONCLUSION

From the results of this analysis, there is evidence that spatial ability is important when learning to program. There are also interactions with other factors such as confidence levels, expected success, and programming experience. When the impact of these factors was reduced by focusing on a more advanced group of students, spatial ability was observed to have a stronger effect. Future studies need to be carried out on a larger cohort of students to allow statistical analysis of the relative contribution of each of the variables. It would also be beneficial to have a larger ratio of females in the student group to enable a study of gender effects.

This study provides an important contribution to knowledge about why some students struggle to achieve in introductory computer science courses, resulting in high attrition and failure rates. While spatial ability has been shown to be relevant, we do not feel that mental rotation capacity should be used as a means of predetermining programming aptitude, but should be considered while devising pedagogical interventions. Thought needs to be given to teaching methods and software visualizations that help students with low spatial ability to envisage abstract concepts and build better mental models (Wiedenbeck et al., 2004). The benefits of spatial training intervention also need to be assessed.

ENDNOTE

1. Available at http://www.uwm.edu/~johnchay/

REFERENCES

Bandura, A. (1986). *Social foundations of thought and action.* Englewood Cliffs, NJ, USA: Prentice Hall.

Beckwith, L., & Burnett, M. (2004). Gender: An important factor in end-user programming environments? In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing Languages and Environments* (pp. 107–114). Washington, DC: IEEE Computer Society.

Bergin, S., & Reilly, R. (2005). Programming: Factors that influence success. *SIGCSE Bulletin, 37*(1), 411–415.

Bishop-Clark, C. (1995). Cognitive style, personality, and computer programming. *Computers in Human Behavior, 11,* 241–260.

Bradley, C. A. (1985). The relationship between students' information-processing styles and Logo programming. *Journal of Educational Computing Research, 1,* 427–433.

Bruckman, A., Jensen, C., & DeBonte, A. (2002). Gender and programming achievement in a CSCL environment. In G. Stahl (Ed.), *Proceedings of the Computer Supported Collaborative Learning* (CSCL) *2002 Conference* (pp. 119–127). Hillsdale, NJ, USA: Erlbaum.

Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 49–52). New York: ACM Press.

Casey, M. B., Nuttall, R. L., Pezaris, E., & Benbow, C. P. (1995). The influence of spatial ability on gender differences in mathematics college entrance test scores across diverse samples. *Developmental Psychology, 31*, 697–705.

Cathcart, W. (1990). Effects of Logo instruction on cognitive style. *Journal of Educational Computing Research, 6*, 231–242.

Chen, C., Czerwinski, M., & Macredie, R. (2000). Individual differences in virtual environments: Introduction and overview. *Journal of the American Society for Information Science, 51*, 499–507.

Cox, A., Fisher, M., & O'Brien, P. (2005). Theoretical considerations on navigating codespace with spatial cognition. In P. Romero, J. Good, E. Acosta Chaparro, & S. Bryant (Eds.), *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group* (pp. 92–105). Retrieved May 21, 2006, from http://www.ppig.org/workshops/17th-programme.html

Curtis, B. (1981). Substantiating programmer variability. *Proceedings of the IEEE, 69*, 846.

Cutmore, T. R. H., Hine, T. J., Maberly, K. J., Langford, N. M., & Hawgood, G. (2000). Cognitive and gender factors influencing navigation in a virtual environment. *International Journal of Human-Computer Studies, 53*, 223–249.

Downs, R. M., & Stea, D. (1973). Cognitive maps and spatial behavior: Process and products. In R. M. Downs & D. Stea (Eds.), *Image and environments* (pp. 8–26). London: Edward Arnold.

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.

Fincher, S., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R., & Petre, M. (2005). *Programmed to succeed? A multi-national, multi-institutional study of introductory programming courses* (University of Kent, Computing Laboratory Technical Report 1-05). Retrieved May 21, 2006, from http://www.cs.kent.ac.uk/pubs/2005/2157/

Fisher, M., Cox, A., & Zhao, L. (2006). Using sex differences to link spatial cognition and program comprehension. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (ICSM; pp. 289–298). Washington, DC: IEEE Computer Society.

Guzdial, M., & Soloway, E. (2002). Teaching the Nintendo generation to program. *Communications of the ACM, 45*, 17–21.

Halpern, D. F. (2000). *Sex differences in cognitive abilities*. Mahwah, NJ, USA: Lawrence Erlbaum Associates.

Hassell, J. (1982). Cognitive style and a first course in computer science: A success story. *AEDS Monitor, 21*, 33–35.

Jones, S., & Burnett, G. (2006). Give the girls a chance: Should spatial skills training be incorporated into the curriculum? In K. Morgan, C. A. Brebbia, & J. M. Spector. (Eds.), *The Internet Society II: Advances in education, commerce & governance* (pp. 105–114). Southampton, UK: WITpress.

Jones, S., & Burnett, G. (2007a). Spatial skills and navigation of source code. In *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE '07; pp. 231–235). New York: ACM.

Jones, S., & Burnett, G. E. (2007b). Children's navigation of hyperspace: Are spatial skills important? In *Proceedings of the Sixth IASTED International Conference on Web-Based Education* (pp. 643–648). Anaheim, CA, USA: ACTA Press.

Kimura, D. (1999). *Sex and cognition*. Cambridge, MA, USA: MIT Press.

Mancy, R., & Reid, N. (2004). Aspects of cognitive style and programming. In E. Dunican & T. Green (Eds.), *Proceedings of the Sixteenth Annual Workshop of the Psychology of Programming Interest Group* (PPIG '04; pp. 1–9). Retrieved January 3, 2005, from http://www.ppig.org/workshops/16th-programme.html

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: What's the connection? *Communications of the ACM, 29*, 605–610.

McKenna, F. P. (1984). Measures of field dependence: Cognitive style or cognitive ability? *Journal of Personality and Social Psychology, 47*, 593–603.

Miller, R. B., Kelly, G. N., & Kelly, J. T. (1998). Effects of Logo computer programming experience on problem solving and spatial relations ability. *Contemporary Educational Psychology, 13*, 348–357.

Mumtaz, S. (2001). Children's enjoyment and perception of computer use in the home and the school. *Computers & Education*, *36*, 347–362.

Pease, A., & Pease, B. (2001). *Why men don't listen and women can't read maps*. London: Orion.

Quaiser-Pohl, C., & Lehmann, W. (2002). Girls' spatial abilities: Charting the contributions of experiences and attitudes in different academic groups. *British Journal of Educational Psychology*, *72*, 245–260.

Rountree, N., Rountree, J., & Robins, A. (2002). Predictors of success and failure in a CS1 course. *SIGCSE Bulletin*, *34*, 121–124.

Schumacher, P., & Morahan-Martin, J. (2001). Gender, Internet and computer attitudes and experiences. *Computers in Human Behavior*, *17*, 95–110.

Scragg, G., & Smith, J. (1998). A study of barriers to women in undergraduate computer science. *SIGCSE Bulletin*, *30*, 82–86.

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems*. Cambridge, MA, USA: Winthrop Publishers Inc.

Smith, P. (1992). Spatial ability and its role in United Kingdom education. *Vocational Aspect of Education*, *44*, 103–106.

Subrahmanyam, K., Greenfield, P. M., Kraut, R., & Gross, E. (2001). The impact of computer use on children's and adolescents' development. *Applied Developmental Psychology*, *22*, 7–30.

Tolhurst, D., Baker, B., Hamer, J., Box, I., Lister, R., Cutts, Q., Petre, M., de Raadt, M., Robins, A., Fincher, S., Simon, S., Haden, P., Sutton, K., Hamilton, M., & Tutty, J. (2006). Do map drawing styles of novice programmers predict success in programming? A multi-national, multi-institutional study. In D. Tolhurst & S. Mann (Eds.), *Proceedings of the 8th Australian Conference on Computing Education* (pp. 213–222). Hobart, Australia: Australian Computer Society, Inc.

Vandenberg, S. G., & Kuse, A. R. (1978). Mental rotations, a group test of three-dimensional spatial visualization. *Perceptual and Motor Skills*, *47*, 599–604.

Velez, M., Silver, D., & Tremaine, M. (2005). Understanding visualization through spatial ability differences. In *Proceedings of the IEEE Visualization 2005 Conference* (VIS 2005; pp. 511–518). New York: IEEE.

Voyer, D., Nolan, C., & Voyer, S. (2000). The relation between experience and spatial performance in men and women. *Sex Roles: A Journal of Research*, *43*, 891–915.

Webb, N. M. (1984). Microcomputer learning in small groups: Cognitive requirements and group processes. *Journal of Educational Psychology*, *76*, 1076–1088.

Werner, S. Krieg-Bruckner, B., Mallot, H. A., Schweizer, K., Freksa, C., & Jahrestagung, G. (1997). Spatial cognition: The role of landmark, route and survey knowledge in human and robot navigation. In M. Jarke, K. Pasedach, & K. Pohl (Eds.), *Informatik '97* (pp. 41–50). New York: Springer.

Werth, L. H. (1986). Predicting student performance in a beginning computer science class. In J. C. Little & L. N. Cassel (Eds.), *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '86; pp. 138–143). New York: ACM.

Westerman, S. J., & Cribbin, T. (1999). Navigating virtual information spaces: Individual differences in cognitive maps. In *Proceedings of the UK Virtual Reality Special Interest Group* (UKVRSIG '99), Salford University, UK.

Wiedenbeck, S., LaBelle, D., & Kain, V. N. R. (2004). Factors affecting course outcomes in introductory programming. In *Proceedings of the Sixteenth Annual Workshop of the Psychology of Programming Interest Group* (PPIG '04) Retrieved May 15th 2005 from http://www.ppig.org/workshops/16th-programme.html

Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. *INROADS of SIGCSE, 33*, 184–188.

Authors' Note

The authors wish to thank Professor Emeritus John C. Hay, the University of Wisconsin, Milwaukee, for permitting use of, and customizing, his version of the 3-D Mental Rotation Test.

All correspondence should be addressed to:
Sue Jones
School of Computer Science
University of Nottingham
Nottingham
NG8 1BB
United Kingdom
s.jones@nottingham.ac.uk

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 62–74

A ROLES-BASED APPROACH TO VARIABLE-ORIENTED PROGRAMMING

Juha Sorva
Helsinki University of Technology
Finland

Abstract: *Delocalized variable plans pose problems for novice programmers trying to read and write programs. Variable-oriented programming is a programming paradigm that emphasizes the importance of variable-related plans, and localizes actions pertaining to each variable together in one place in the program code. This paper revisits the idea of variable-oriented programming and shows how it can be founded on roles of variables: stereotypes of variable use suitable for teaching to novices. The paper sketches out how variable-oriented, roles-based programming could be implemented using either a new programming language or a framework built on an existing language. The possible applications, merits, and problems of a roles-based approach, and variable-oriented programming in general, are discussed. This paper points toward possible research directions for the future and provides a basis for further discussions of variable-oriented, roles-based programming.*

Keywords: *roles-based programming, variable-oriented programming, roles of variables, delocalized plans, programming languages.*

INTRODUCTION

It has been widely noted that novice programmers have great difficulty in comprehending and creating computer programs (for recent reports, see Lister et al., 2004; McCracken et al., 2001). A partial explanation for this is provided by the novices' lack of programming-related schemas or plans (Détienne, 1990; Soloway & Ehrlich, 1986). *Schemas* are mental knowledge structures for storing abstract information that can be applied when planning solutions to specific problems that fall within the scope of the schema. An expert in a domain possesses a wide array of rich, domain-specific schemas that reduce cognitive load during problem-solving tasks, such as programming and enable solving more complex problems. An expert's problem-solving process is characterized by planning ahead and forward development (Byckling & Sajaniemi, 2006a; Rist, 1989).

Many schemas in programming are related to the use of variables (Soloway, Ehrlich, Bonar, & Greenspan, 1982). For instance, a basic programming schema could describe how variables can serve as "counters," whose values start at zero and are then repeatedly incremented

URN:NBNfi:jyu-200804151353

by one. Commonly, the ways in which a variable is used in a program are not defined by a single line of code or even by consecutive lines; references to each variable are spread throughout the program code. In the terminology of Soloway, Lampert, Letovsky, Littman, and Pinto (1988), the plan for such a variable is *delocalized*. Delocalization of a plan increases the cognitive load of a programmer trying to comprehend it, since multiple separate units have to be kept in working memory at once in order to figure out the plan. Novice programmers may find coping with this cognitive load very difficult. Delocalized plans can be clarified with documentation (Soloway et al., 1988) or software tools (Sajaniemi & Niemeläinen, 1989). In recent years, *roles of variables* have been introduced as a means to describe, discuss, and think about common stereotypes of variable usage (Sajaniemi, 2002, 2003). Roles of variables have been used to document variable plans and for other purposes in teaching introductory programming (Byckling & Sajaniemi, 2007; Sajaniemi & Kuittinen, 2005; Sorva, Karavirta, & Korhonen, 2007).

This paper presents ongoing work on *variable-oriented programming*, a programming paradigm that places an emphasis on localizing variable-related actions in program code. This work draws on prior work on roles of variables, and uses roles as a basis for creating variable-oriented programs. The paper is structured as follows. The Related Work section describes previous work on roles of variables and variable-oriented programming. The A Roles-Based Approach section introduces a new approach to variable-oriented programming, and discusses how it could be implemented, using either a custom-made programming language or existing programming languages. The Discussion section then takes a look at the possible uses, merits, and downsides of the new approach. The paper concludes with general comments and a look at possible future work.

RELATED WORK

Roles of Variables

Roles of variables are stereotypes of variable use in computer programs (Sajaniemi, 2002). Roles embody expert programmers' tacit knowledge of variable usage patterns, which can be made explicit and taught to students (Sajaniemi & Navarro Prieto, 2005). Roles can help teachers explain delocalized variable-related schemas in programs and assist in the stepwise refinement of pseudocode designs of algorithms (Sorva et al., 2007). Prior research suggests that introductory-level students who are taught programming using roles of variables gain better program comprehension skills than students taught in an otherwise similar way but without using roles (Sajaniemi & Kuittinen, 2005). Moreover, roles-based instruction facilitates the development of program construction skills better than traditional instruction, especially if roles-based visualizations of programs are also used in teaching (Byckling & Sajaniemi, 2006b, 2007).

According to Sajaniemi's (2002) research, the behavior of 99% of variables in novice-level programs can be characterized within a small set of roles. The following list, reprinted from Sorva et al. (2007, p. 410), briefly introduces each variable role. For a fuller introduction to roles of variables, and concrete program examples of each role, see Sajaniemi (2003).

1. A variable has the role *fixed value* if the variable's value is not changed after it is initialized.

2. A variable has the role of *stepper* if it is assigned values in a systematic and predictable order. An example of a stepper is an index counter used when looping through an array of elements.
3. A variable has the role of *most-recent holder* if it holds the latest value in a sequence of unpredictable data values. For instance, a most-recent holder could be used to store the latest element encountered while iterating through a collection of data elements, or the latest value that has been assigned to an object's attribute (i.e., to an instance variable that is a most-recent holder) by a setter method.
4. The role *most-wanted holder* describes variables that hold the "best" value encountered in a sequence of values. Depending on the program and the type of the data, the best value may be the largest, smallest, alphabetically first, or an otherwise most appropriate value.
5. A variable has the role *gatherer* if the variable is used to somehow combine data values that are encountered in a sequence of values, and the variable's value represents this accumulated result. For instance, a variable keeping track of the balance of a bank account (e.g., the sum of deposits and withdrawals) is a gatherer.
6. A *follower* is a variable that always holds the most recent previous value of another variable. Whenever the value of the followed variable changes, the value of the follower is also changed. For example, the "previous node pointer" used when traversing a linked list is a follower.
7. A variable is a *one-way flag* if it only has two possible values and if a change to the variable's value is permanent. That is, once a one-way flag is changed from its initial value to the other possible value, it is never changed back. For example, a Boolean variable keeping track of whether or not errors have occurred during processing of input is a one-way flag.
8. A variable has the role *temporary* if the value of the variable is needed only for a short period. For example, an intermediate result of a calculation can be stored in a temporary in order to make it more convenient or efficient to use in later calculations.
9. An *organizer* is a variable that stores a collection of elements for the purpose of having that collection's contents rearranged. An example of an organizer is a variable that contains an array of numbers during sorting.
10. A variable is a *container* if it stores a collection of elements in which more elements can be added (and, typically, can be removed as well). For example, a variable that references a stack could be a container.
11. A *walker* is a variable whose values traverse a data structure, moving from one location in the structure to another. For instance, a variable that contains a reference to a node in a tree traversal algorithm and a variable that keeps track of the search index in a binary search algorithm can be considered to be walkers.

Variable-Oriented Programming

In traditional procedural and object-oriented programming, the behavior of a variable, that is, the logic that dictates how the variable is used, is often defined at multiple distinct locations in program code. Depending on the scope of the variable, the behavior may be described by

inconsecutive lines of code within a function or method, may be located in a number of functions, or even located in several program modules. Declaring a variable, if it is explicitly done in the language at all, is a matter separate from the variable's behavior.

There is an alternative way to organize variable behavior in programs. If a variable's behavior pattern is defined at the variable's declaration, the "usage plan" of the variable becomes localized in one place. This idea is central to the variable-oriented way of programming discussed in this paper. In a *variable-oriented program*, each variable declaration is accompanied by a definition of how the variable's value is initialized and later updated. A variable declaration could also include information of when the variable's value is read and dependencies on other variables. In a variable-oriented program, such rich variable declarations serve as the basis of, and indeed govern, the creation of algorithms.

Variable-oriented programming has made an appearance in literature before. It was introduced in connection with the program editor VOPE, which makes use of variable-orientation to provide multiple views of program code written in the Pascal language (Sajaniemi & Niemeläinen, 1989). In addition to a traditional control-flow oriented view of Pascal programs, VOPE shows a purely variable-oriented view, which groups code fragments so that all references to each variable are gathered together.

A ROLES-BASED APPROACH

A look at how an algorithm could be devised using roles of variables may be useful. The passage below presents a hypothetical thought pattern of how a student of programming, who has been taught to use the roles of variables, might go about the task of creating an algorithm for computing the *n*th Fibonacci number.

> Some way of keeping track of consecutive Fibonacci numbers is needed to compute to the *n*th one. Each new value is produced by computing a new value based on the current one. That's a job for a gatherer. And since, in this case, each new value is computed based on two older values, a follower is needed to store the older value of the gatherer. By starting from the first Fibonacci number (one), then after n-1 updates to the gatherer, the result should be reached.

While fictional and idealized, this example offers an idea of how roles-based reasoning might proceed and make use of the common patterns of variable use embodied by roles of variables. It is also an example of thinking ahead: The programmer uses existing schemas to plan in advance how he/she will use the two variables. Figure 1 shows a somewhat more formal and complete description of the algorithm, using a pseudocode notation that closely reflects the reasoning process described above.

In the pseudocode in Figure 1, two variables are declared, each with a different role. For each variable, its behavior has been declared as a part of the variable definition. The example illustrates how an algorithm can be built by attaching behavior to variable definitions. Further, it shows how roles of variables can serve as templates for common patterns in a variable-oriented program.

Each variable is declared as an instance of a role, which determines the kinds of operations that need to be defined for each instance of the role. For example, all gatherers require a definition of how their values change as a function of the same variable's old value,

```
define GATHERER curr:
        initial value is 1
        always updated by computing value of curr + prev

define FOLLOWER prev:
        initial value is 0
        follows curr (and always receives its old value)

make n-1 updates to curr (results in changes to both curr and prev)
print curr (which now holds the nth Fibonacci number)
```

Figure 1. Variable-oriented pseudocode.

whereas a follower is dependent on another variable whose old values it receives. For a fixed value (not shown in the example), only an initialization is needed, while a most-wanted holder would define an operation to test whether a given value is "more wanted" than the current value, and so on.

The next two subsections explore possible implementations for variable-oriented, roles-based algorithms such as that in Figure 1. The first one sketches out a variable-oriented programming language that uses roles of variables as language-level abstractions. The second then takes a look at how a similar framework could be implemented in an existing programming language.

A Roles-Based Language

Figure 2 provides an example of variable-oriented code based on roles of variables. It is written in a speculative language called ROTFL (Role-Oriented, Titillating but Fictional Language). The reader should note that ROTFL is at a draft stage and lacks a full syntactical and semantical specification. The notation is used here to provide "food for thought." In Figure 2 and in other Fibonacci examples in this paper, *n* is an integer-valued constant that determines which Fibonacci number is to be printed out.

```
Gatherer curr:
   inits to: 1
   updates with: curr + prev

Follower prev:
   inits to: 0
   follows: curr

update curr times n-1
print(curr)
```

Figure 2. The Fibonacci algorithm in the language ROTFL.

In ROTFL, there are no traditional variable definitions. Instead, all variables are defined in terms of roles and associated with behaviors appropriate for those roles. Roles of variables are language-level constructs, and there are reserved words related to defining or using variables with particular roles (e.g., follower, update). ROTFL does not feature assignment operators or statements in the traditional sense. Instead, variables' values are changed in role-specific ways. For instance, values are assigned to gatherers with the reserved word *update*, which uses the updates-with operation of the gatherer to compute a new value for the variable, and followers receive new values implicitly as the value of a followed variable changes.

Traditional loops are also conspicuous by their absence in Figure 2, despite the fact that the algorithm is an iterative one. In this example, repetition is achieved using the keyword *times* in association with updating the value of the gatherer curr. Another mechanism for achieving repetition is illustrated in Figure 3, where a *do each* command repeatedly updates a most-recent holder variable until a condition associated with the variable is reached. The same example also shows a most-wanted holder dependent on a most-recent holder that serves as its *source*.

```
MostRecentHolder input:
   updates with: readLine()
   until: input == 'stop'

MostWantedHolder longestInput:
   source: input
   wants value if: value.length() > longestInput.length()

do each input
print(longestInput)
```

Figure 3. A ROTFL code fragment to read in lines and print out the longest one.

Implementing Roles in an Existing Language

Variable-oriented programming can also be done within an existing programming language, provided a suitable framework is available. Figure 4 shows how the variable-oriented, roles-based program from Figures 1 and 2 can be written in the Python language. The program makes use of an anonymous function defined using Python's lambda mechanism.

The program in Figure 4 relies on a framework that defines roles of variables as Python classes, and role-related operations (such as updating the value of a gatherer) as methods of these classes. A partial framework for this purpose, defining the classes gatherer and follower, is given in the Appendix.

```
curr = Gatherer(1, lambda: curr + prev)
prev = Follower(0, curr)
curr.updateTimes(n-1)
print(curr)
```

Figure 4. A variable-oriented code fragment in Python.

DISCUSSION

Uses of Roles-Based Programming

As noted in the introduction to this paper, prior research suggests that the behavior of 99% of variables can be characterized with a small set of roles, at least within novice-level programs (Sajaniemi, 2002). It does not immediately follow, however, that 99% of even novice-level programs can be conveniently written as variable-oriented programs using roles as templates for variable behavior. Nevertheless, it seems roles form a solid foundation for creating variable-oriented programs, as the small role set provides a quite substantial number of variables with templates that capture some key aspects about how those variables are used. This matter calls for further study.

Variable-oriented programming localizes variable plans in program code. Prior work in cognitive psychology of programming suggests that it is likely that localizing variable plans facilitates the extraction and construction of variable-related schemas (Soloway et al., 1988) and therefore aids novices in acquiring some key programming skills. With this in mind, and in light of previous experiences of using roles of variables in teaching, one can speculate whether a variable-oriented, roles-based language could be useful for teaching introductory programming. Clearly, there could be merits to such an approach if variable-orientation helps students construct variable-related schemas, if roles can be used to encourage forward development (Byckling and Sajaniemi, 2007), and if there were roles-aware program development tools that could provide helpful feedback and error messages.

There also clearly are problems with such an approach. Not least of these is that while variable-oriented programs emphasize variable-related plans and the data flow of programs, the control flow of the program is not in focus. Understanding "what happens when" during the execution of a variable-oriented program may be quite difficult, especially for the beginner. There is a trade-off between emphasizing variable-related schemas and emphasizing control-flow-related schemas. Using tools similar to VOPE (Sajaniemi & Niemeläinen, 1989), which provides multiple views of programs, could be useful in combining these different aspects of programs. A notation based on roles of variables could be used to build variable-oriented views and to link them to procedural or object-oriented views.

Depending on the notation used, a variable-oriented program can be quite self-documenting of variable-related schemas (see, e.g., Figure 2). Roles of variables help in this, since role names succinctly describe patterns of variable use. However, it is not immediately obvious what the documentative value of variable-oriented notations is compared to non-variable-oriented notations that explicate the role of each variable (e.g., by simply tagging each variable declaration with a role name using code comments). Documenting delocalized variable behavior using role names may often do enough and using a variable-oriented language may be overkill for this purpose.

Even if beginners are not taught variable-oriented, roles-based programming directly, they might indirectly benefit from it. Bergin (2005) suggests that instructors of programming (and others) could benefit from "etudes" that take one particular programming technique to an extreme. While such etudes have no intrinsic value of their own, they can help hone one's skills in a particular technique and to ingrain that technique into one's thinking. For helping instructors (not novice programmers) make use of polymorphism, Bergin suggests the following etude:

> Find some old program that you have around and that you are proud of…. Strictly as an etude, rewrite that program with NO if/switch statements: no selection at all. Solve all of the problems your ifs solve with polymorphism. (Bergin, 2005, p. 1)

In a similar vein, roles-based programming could serve as an etude for using roles of variables in general. The intellectual exercise of rewriting programs in a variable-oriented way, using roles as templates for variables, with no traditional-style assignment and perhaps with no traditional-style loops, could deepen instructors' understanding of roles and help them think of algorithms in terms of variables and roles. At least, the exercise has expanded the mind of this author.

Variable-Oriented "Purity"

According to Sajaniemi and Niemeläinen (1989, p. 67, my emphases), "Variable-oriented programming is a new programming paradigm which collects *all* actions concerning any single variable together.... The plan of a variable is clearly visible and *totally* described in the variable definition."

A "pure" variable-oriented program, then, would gather all references (assignments and reads) to a variable into one complete variable definition, irrespective of the location of these references in the control flow of the program. The reader may note that the examples shown in this paper are not pure by this strict definition. For instance, in Figure 2, neither the command *update* nor reading the variable's value for printing purposes (i.e., the last two lines) is located within the variable definition. The example can be seen as a hybrid that is largely variable-oriented but partially control-flow-oriented. It can be contrasted with the pure variable-oriented views displayed by the VOPE tool (Sajaniemi & Niemeläinen, 1989).

Roles of variables are concerned with assignment, with change (or lack of change) in the values of variables, and with the way consecutive values of variables are related to each other. Roles are not concerned with *when* a variable's value is updated or read, or with what is done with the value after it has been read (whether it is printed, passed as a parameter, or something else). A variable-oriented program based solely on roles of variables will not be pure. A more complete discussion of the purity of variable-orientation is beyond the scope of this paper. The next subsection also touches on the issue of purity, however, as it briefly explores the relationship between object-oriented programming, variable-orientation, and roles-based programming.

Compatibility with Object-Orientation

The original set of roles of variables was discovered by analyzing procedural programs. Since then, roles of variables have been applied to object-oriented as well as functional programs (Sajaniemi, Ben-Ari, Byckling, Gerdt, & Kulikova, 2006). Roles seem to be a useful tool irrespective of the programming paradigm used.

What, then, is the relationship between variable-orientation and object-orientation? Quoting again from Sajaniemi and Niemeläinen (1989, p. 67), "In object-oriented programming all operations applicable to objects of a class are described in one place.... In variable-oriented programming programs center around the variables. A variable, and all the actions using that particular variable, are described in one place."

One of the two paradigms elevates classes as a key abstraction around which program code is structured; the other does the same to variables. These two abstractions are in competition, but not incompatible. It is quite possible to envision a hybrid of the object-oriented and variable-oriented paradigms, as illustrated by the example in Figure 5.

It is easy to see that Figure 5 is not pure in terms of variable-orientation. The generic plan for using the instance variable *balance*, a gatherer, is defined at the variable declaration. However, the precise ways in which the three methods make use of this generic plan are spread out in the code.

```
class Account:
   private Gatherer balance:
      inits to: 0
      updates with (FixedValue amount):
         if (balance + amount < 0) then:
            0
         else:
            balance + amount

      public method deposit(FixedValue depositSize):
         update(depositSize) balance

      public method withdraw(FixedValue withdrawalSize):
         update(-withdrawalSize) balance

      public method getBalance():
         balance
```

Figure 5. A ROTFL class representing simple bank accounts with non-negative balances.

Another issue needs to be considered when applying roles of variables to object-oriented programs. As was noted by Sorva et al. (2007, p. 419),

> Annotating a member variable and a local variable with the same role name indicates that we think of them as similar. However, our experience suggests that in many people's perception a most-recent holder member variable, for instance, is used rather differently than a most-recent holder local variable. A settable attribute of an object (the name of a person object, say) is experienced as being quite different from a local variable that stores the most recent element encountered in a collection during iteration.... This kind of dividedness of roles is potentially confusing….

It may be that, in order to apply roles-based programming to object-oriented programs, new roles are needed to represent different uses of instance variables. As an example, a role name *settable attribute* could better describe the purpose of most-recent holder instance variables. If needed, the roles-based language or framework could provide a somewhat different template for settable attributes than for other most-recent holders.

CONCLUSIONS AND FUTURE WORK

In this paper, I have revisited the previously discovered ideas of variable-oriented programming and roles of variables. This paper combines these two ideas by founding variable-orientated programs on roles, and sketches out how such a roles-based approach could be implemented using a roles-based programming language or a framework written in another language. The paper has described ongoing work on tools for roles-based programming, and discussed the possible applications, merits, and problems of the approach. It is my hope that this paper can serve as a basis for further discussions of variable-oriented, roles-based programming.

This paper has merely introduced the idea of using roles of variables in variable-oriented programming. There are many research paths that could be followed in the future. Roles-based languages or frameworks could be developed further from the drafts presented, investigating the suitability of the variable-oriented approach for more complex programs. Ways of defining dependencies between variables could be explored, as could the idea of actions that trigger when variables' values change. Here, inspiration could perhaps be drawn from earlier work, such as the language EDEN (Yung, Joy, & Ward, 1987), which, although not variable-oriented, allows the programmer to associate "action specifications" to variables.

The suitability of the current set of roles of variables for roles-based programming needs exploring, as does the idea of custom roles defined by the programmer. The possible usefulness of roles-based programming outside educational settings could be investigated.

The effects of a variable-oriented notation on understanding programs' control flow will need to be explored if this approach is to be taken further. Roles-based tools supporting both variable-oriented and other views of programs could be developed. If the approach looks promising, the potential of variable-oriented programming in instruction could be evaluated.

Using roles-based programming as an etude for instructors to deepen their understanding of roles of variables seems like a promising avenue to take in the future. This can be done even using a speculative language like ROTFL.

REFERENCES

Bergin, J. (2005, July). *Variations on a polymorphic theme: An etude for computer programming*. Paper presented at the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, Glasgow, UK. Retrieved April 15, 2007, from http://www.cs.umu.se/~jubo/Meetings/ECOOP05/Submissions/Bergin-full.pdf

Byckling, P., & Sajaniemi, J. (2006a). A role-based analysis model for the evaluation of novices' programming knowledge development. In *ICER '06: Proceedings of the 2006 International Workshop on Computing Education Research* (pp. 85–96). New York: ACM Press.

Byckling, P., & Sajaniemi, J. (2006b). Roles of variables and programming skills improvement. *SIGCSE Bulletin, 38*, 413–417.

Byckling, P., & Sajaniemi, J. (2007). A study on applying roles of variables in introductory programming. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC '07; pp. 61–68). Coeur d'Alène, ID, USA: IEEE Computer Society.

Détienne, F. (1990). Expert programming knowledge: A schema-based approach. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 205–222). London: Academic Press.

Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., & Sanders, K. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*, 119–150.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*, 125–180.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science, 13*, 389–414.

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (pp. 37–39). Arlington, VA, USA: IEEE Computer Society.

Sajaniemi, J. (2003). The roles of variables home page. Retrieved April 15, 2007, from http://cs.joensuu.fi/~saja/var_roles

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., & Kulikova, Y. (2006). Roles of variables in three programming paradigms. *Computer Science Education, 16*, 261–279.

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education, 15*, 59–82.

Sajaniemi, J., & Navarro Prieto, R. (2005). Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group* (PPIG; pp. 145–159). Brighton, UK: University of Sussex.

Sajaniemi, J., & Niemeläinen, A. (1989). Program editing based on variable plans: A cognitive approach to program manipulation. In *Proceedings of the Third International Conference on Human-computer Interaction on Designing and Using Human-computer Interfaces and Knowledge Based Systems* (2nd ed.; pp. 66–73). New York: Elsevier Science Inc.

Soloway, E., & Ehrlich, K. (1986). Empirical studies of programming knowledge. In C. Rich & R. C. Waters (Eds.), *Readings in artificial intelligence and software engineering* (pp. 507–521). San Francisco: Morgan Kaufmann Publishers Inc.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), *Directions in human-computer interactions* (pp. 27–54). Norwood, NJ, USA: Ablex Publishing.

Soloway, E., Lampert, R., Letovsky, S., Littman, D., & Pinto, J. (1988). Designing documentation to compensate for delocalized plans. *Communications of the ACM, 31*, 1259–1267.

Sorva J., Karavirta V., & Korhonen A. (2007). Roles of variables in teaching. *Journal of Information Technology Education, 6*, 407–423.

Yung, E., Joy, M., & Ward, A. (1987). *EDEN: The engine for definitive notations*. Retrieved April 15, 2007, from http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/eden/

Author's Note

All correspondence should be addressed to:
Juha Sorva
Helsinki University of Technology
Department of Computer Science and Engineering
Konemiehentie 2
02015 TKK, Finland
jsorva@cs.hut.fi

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

APPENDIX

A PARTIAL FRAMEWORK FOR VARIABLE-ORIENTED, ROLES-BASED PROGRAMMING IN PYTHON

The classes below form a partial (but working) framework for writing variable-oriented programs in terms of roles of variables in the Python language. The partial framework shown here has implementations for only some main features of three roles (fixed value, gatherer and follower). For an example of using the classes, see Figure 4.

Other variable roles can be implemented in Python along the same lines. Implementation-wise, most-recent holders are simple; they just need an update method that replaces the old value with the given new one. Steppers and most-wanted holders can be implemented similarly to gatherers and most-recent holders, respectively. Temporary variables are akin to fixed values and trivial to implement, one-way flags likewise. Containers need a more complex class, with methods for adding and removing values. Alternatively, containers could be left unimplemented as an explicit role, relying on Python's built-in data structures instead. Organizers are characterized by a variable-specific function that defines a means for ordering data, which can be passed as a constructor parameter (cf. the gatherer implementation below).

The few variables that do not have any of the roles in Sajaniemi's (2003) role set can be treated as most-recent holders, or with a functionally similar but differently named class (e.g., *special*), which effectively allows new values to be assigned freely to the variable by passing them as a parameter to update. Alternatively, programmers could define their own program-specific custom roles.

```
import types

class Role:
    def __init__(self, initsTo):
        self.followers = []
        if (type(initsTo) == types.FunctionType):
            self.value = initsTo()
        else:
            self.value = initsTo

    def __add__(self, x):
        return self.value + x
    __radd__ = __add__

    def __str__(self):
        return repr(self.value)

    def addFollower(self, follower):
        self.followers.append(follower)
```

```
class FixedValue(Role):
    def __init__(self, initsTo):
        Role.__init__(self, initsTo)


class Gatherer(Role):
    def __init__(self, initsTo, updatesWith):
        Role.__init__(self, initsTo)
        self.updatesWith = updatesWith

    def update(self):
        oldValue = self.value
        self.value = self.updatesWith()
        for f in self.followers:
            f.update(oldValue)

    def updateTimes(self, times):
        for time in range(times):
            self.update();


class Follower(Role):
    def __init__(self, initsTo, followedVariable):
        Role.__init__(self, initsTo)
        followedVariable.addFollower(self)

    def update(self, newValue):
        oldValue = self.value
        self.value = newValue
        for f in self.followers:
            f.update(oldValue)
```

HUMAN TECHNOLOGY

An Interdisciplinary Journal on Humans in ICT Environments ISSN: 1795-6889

www.humantechnology.jyu.fi Volume 4 (1), May 2008, 75–91

FROM PROCEDURES TO OBJECTS: A RESEARCH AGENDA FOR THE PSYCHOLOGY OF OBJECT-ORIENTED PROGRAMMING EDUCATION

Jorma Sajaniemi
Department of Computer Science and Statistics, University of Joensuu
Joensuu, Finland

Marja Kuittinen
Department of Computer Science and Statistics, University of Joensuu
Joensuu, Finland

Abstract: *Programming education has experienced a shift from imperative and procedural programming to object-orientation. This shift has been motivated by educators' desire to please the information technology industry and potential students; it is not motivated by research either in psychology of programming or in computer science education. There are practically no results that would indicate that such a shift is desirable, needed in the first place, or even effective for learning programming. Moreover, there has been an implicit assumption that classic results on imperative and procedural programming education and learning apply to object-oriented programming (OOP) as well. We argue that this is not the case and call for systematic research into the fundamental cognitive and educational issues in learning and teaching OOP. We also present a research agenda intended to improve the understanding of OOP and OOP education.*

Keywords: *programming education, procedural programming, object-oriented programming, psychology of programming.*

INTRODUCTION

During the last 10 years, programming education has experienced a shift from imperative and procedural programming to object-oriented programming (OOP). This shift has been motivated by educators' desire to please the information technology industry, on one hand, and potential students on the other. Object-orientation and Java have been spreading as the most important implementation platform for new, Web-based applications with widespread visibility among computer users, which has created the illusion that the word *programming* equals Java OOP. Thus, students want to learn Java from the very beginning of their programming studies. Teachers' selection of the first programming language is dominated by student demand and a willingness to provide students with marketable skills (de Raadt, Watson, & Toleman, 2002), that is, Java programming. With the current drop in enrollments to academic computing programs (Cassel, McGettrick, Guzdial, & Roberts, 2007) educators'

URN:NBNfi:jyu-200804151354

thirst for pleasing potential students will probably only increase. Moreover, many companies want to hire students who know how to program in Java and educators may think that if an institute is not teaching Java, its reputation among those companies is damaged.

It should be noted that the shift to object-orientation in education is not motivated by psychology of programming or computer science education research: There are practically no results that would indicate that such a shift is desirable, needed in the first place, or even effective for learning programming (Lister et al., 2006). Yet, learning programming should be the most important issue—not learning the peculiarities of a single paradigm or a certain language. Note that "learning programming" does not refer to imperative[1] or procedural—neither functional nor logic—programming, but learning programming in a way that can be applied in many programming paradigms and many programming languages.

Indeed, we are surprised to find out that the cognitive consequences of the shift to object-orientation had not been studied before the shift, and only superficially even after it. There are some studies on the misunderstanding of object-oriented (OO) concepts but the development of OOP skills and comprehension of OO concepts have not been studied. There has been an implicit assumption that classic results on imperative and procedural programming education and learning (see Robins, Rountree, & Rountree, 2003, and Winslow, 1996, for reviews) also apply to OOP, but we fear that this is not always the case. OOP is so much more complicated than imperative and procedural programming—both at the concrete notational level and at a more abstract conceptual level—that there are good grounds to question whether the classic results can be generalized to object-orientation.

What this means in practice is that educational institutions around the world are implementing curricula and teaching methods that are not based on research, but on intuition. There are practically no theories on the development of programming skills or comprehension of programming concepts in the OO case. It is no wonder that educators are fighting against high dropout rates from (e.g., Kinnunen & Malmi, 2006) and poor learning outcomes in (e.g., McCracken et al., 2001) programming courses. Research has offered educators various pedagogic tricks (e.g., Bennedsen & Caspersen, 2004; Bierre, Ventura, Phelps, & Egert 2006; S. Cooper, Dann, & Pausch, 2003; Holliday & Luginbuhl, 2004; Hsia, Simpson, Smith, & Cartwright, 2005; Kölling & Henriksen, 2005; Lopez-Herrejon & Schulman, 2004; Mahmoud, Dobosiewicz, & Swayne, 2004; Marrero & Settle, 2005; Shanmugasundaram, Juell, & Hill, 2006; Truong, Bancroft, & Roe, 2005; Utting, 2006), but the lack of solid psychological and educational theories makes a holistic approach impossible.

This paper presents a case for systematic research into the comprehension of programming and the development of skills in the OO paradigm. In order to understand the huge shift from imperative and procedural programming to object-orientation, we start by comparing these paradigms at three of the five domains that du Boulay (1989) presents as issues that a learner must master: *notations* of the particular language, the *notional machine* that describes how programs in the particular language are executed, and the *orientation*, describing what programs are for and what can be done with them. Differences between programming paradigms in du Boulay's two remaining domains, *structures* (abstract solutions to standard problems) and *pragmatics* (the skills of planning, developing, testing, debugging, etc.), are more complicated and will not be treated in this paper. It is clear that if differences in the basic constructs—notations, notional machine, and orientation—make the

applicability of classic results to object-orientation dubious, then differences in more complicated issues will make the situation even worse.

This paper is structured as follows. First, we will look at the differences between imperative and procedural programming versus OOP with respect to notations, notional machine, and orientation. Then, we will review research literature and see how it supports our claims. Finally, we will present a research agenda for OOP.

THE NOTATIONAL REVOLUTION

Notations needed in Java programs do differ remarkably from those of imperative and procedural programming[2]. This is partially due to the larger number of programming concepts needed, but also due to the structure of the Java language (Radenski, 2006).

For example, consider the algorithm for simple user interaction in Figure 1, given in a natural language, English. The pseudo code version of this algorithm is given in Figure 2, and a Pascal program for the same task in Figure 3 (from a popular textbook of its time, D. Cooper & Clancy, 1982, p. 15). Even though the notations differ in their level of formality, they look strikingly similar. When we compare the natural language version (that should be in a notation familiar to students) in Figure 1 to the Pascal version (that the students should learn to understand), the new notations and the related concepts are

- "program," name of the program: program
- interaction ports needed: input/output
- "integer" and the variable name: variables
- "write," "writeln," and "readln": input/output
- "var," "begin," "end," and punctuation: language syntax.

The first two of these are required by the language, but are simple to students (this is a program with input and output); the next two are just what the students are learning (the concepts of variable and input/output); the last one is something cryptic required by the language. Parts required by the language vary from one language to another. For example, in Python there would be no special punctuation or statement brackets and the program line would not be needed.

Now, let us turn to the Java version of the same program given in Figure 4, which must be stored in a file with a certain name, Interactive.java. (We assume the existence of another class for user input stored in the file Input.java). Compared with Figure 1, the new notations and the related concepts are:

- "public": visibility
- "class," name of the class: classes and objects
- "static": access rights
- "void": return values
- "main": program
- method name and its argument: methods and their arguments
- "String," "[]," "System," and "Input": predefined classes
- "int" and the variable name: variables

- "println," and "readInt": input/output
- punctuation: language syntax

This list is much longer than the corresponding list for Pascal. And, what is more important, it contains a large number of difficult concepts that are not required for the solution of the problem, but by the structure of the language: classes and objects, visibility, access rights, method definitions and calls, and return values.

```
Tell the user that this is an interactive program.
Ask the user to enter an integer value.
Get the number from the user.
Tell the user what the entered number was.
```

Figure 1: An example program in English.

```
write 'This program interacts with its user.'
write 'Please enter an integer value.'
read Number
write 'The number you entered was:'
write Number
```

Figure 2: The example program in pseudo code.

```
program Interactive (input, output);
      var Number: integer;
begin
      writeln ('This program interacts with its user.');
      writeln ('Please enter an integer value.');
      readln (Number);
      write ('The number you entered was:');
      writeln (Number)
end.
```

Figure 3: The example program in Pascal.

```
public class Interactive {
      public static void main(String[] args) {
            int Number;
            System.out.println("This program interacts with its user.");
            System.out.println("Please enter an integer value.");
            Number = Input.readInt();
            System.out.print("The number you entered was:");
            System.out.println(Number);
      }
}
```

Figure 4: The example program in Java.

One may argue that this example program favors imperative programming and that the first programs used in OOP courses do not contain this much input and output. Even if that were the case, the first Java program will contain almost all of the above concepts.

Thus, the shift to object-orientation and Java has made a revolution at the notational level, even though this might not be obvious at first sight: The lengths of the programs in Figures 3 and 4 are practically the same, yet the number of new notations and concepts is remarkably higher in the Java case. This rise is not due to the programming problems that are solved, but rather to the requirements of the language used.

THE NOTIONAL MACHINE REVOLUTION

In order to be able to understand what individual constructs of a programming language mean and how programs written in that language work, a student must understand how the notional machine (du Boulay, O'Shea, & Monk, 1981) underlying that language works. Programs cannot be understood as strings of characters only; students must understand, for example, what a variable is and how it is affected by assignments. A more thorough understanding of programming includes, for instance, knowledge of typical uses of variables and control structures (Détienne, 2002), which also relies on a proper understanding of the notional machine. The machine needed for understanding the first programs should be simple, or else learning programming becomes hard (du Boulay et al., 1981).

In the procedural approach, instruction typically starts with the imperative constructs: variables, input/output, conditionals, and looping constructs. The notional machine needed to explain these notions consists of

- variable: location or slot with a name and contents
- input/output: two devices connecting variables to external world
- program execution: a program counter referring to a certain point at the program.

A notional machine that consists of the above parts is clearly capable of executing the program in Figure 3 and can be used in teaching the first steps in imperative programming.

An extension to this notional machine is needed when pointers are included:

- pointer: contents of a variable may be the location of another variable.

Further extensions are needed when procedures are introduced:

- procedure call: a call stack
- parameter: room for parameters in the call stack and parameter-passing mechanisms
- return value: mechanism for return value, possibly with room for it in the call stack.

It should be noted that these extensions are fully compatible with the initial notional machine and they can be introduced gradually along with the introduction of new programming language constructs.

In contrast to the procedural approach, OOP requires a much larger and more complicated notional machine from the very beginning. A notional machine that is capable of executing the program in Figure 4 must contain all of the following parts (see the list of concepts of the program given in the previous section):

- object: a heap for objects
- method: a call stack
- parameter: room for parameters in the call stack and parameter-passing mechanisms
- return value: mechanism for return value, possibly with room for it in the call stack
- variable: location or slot with a name and contents (in the call stack)
- input/output: two devices connecting variables to external world
- object reference: contents of a variable or a parameter may be the location of an object in the heap
- program execution: a program counter referring to a certain point at the program.

Moreover, there are concepts that are needed even though they are not directly expressed in the notional machine: visibility and access rights concerning validity of the program, and the relationship between classes and objects concerning the relationship between the program text and the object heap.

Compared with the notional machine in the procedural case, the difference is huge. The OO notional machine described above and needed for the simple program in Figure 4 is not only larger than the corresponding notional machine needed for the equivalent program in Figure 3, but it is much larger than the total notional machine in the procedural case. Furthermore, the notional machine for OOP described above does not even contain parts needed to describe other OO constructs that are typically introduced in the first programming course: subclasses and inheritance, implicit calls of superclass constructors, and polymorphism.

One might argue that there is no need for students to understand notations and the notional machine completely—students can simply put aside unnecessary parts as boiler plates when first learning. The problem with this thinking is that novices have no means to decide which issues are unnecessary and which must be attended to when reading or writing programs. The use of boiler plate code mystifies programming and obscures concepts that should be learned. Programming should not be taught as a copy-and-paste art that only incidentally results in a correctly functioning program, but rather as a clearly defined activity that deals with unambiguous constructs. Otherwise, the central concepts remain blurred.

In summary, the shift to object-orientation and Java has made a revolution at the notional machine level. Not only is the size of the required notional machine much larger than in the procedural case, but the initial notional machine needed in order to understand the first programs is much more complicated, as well.

THE ORIENTATION REVOLUTION

Sajaniemi, Ben-Ari, Byckling, Gerdt, and Kulikova (2006) have studied example programs in elementary programming textbooks among three programming paradigms: procedural, object-oriented, and functional. They found major differences in the programming problem types used in these various programming paradigms. The most important issue in procedural programming textbooks is the functionality of programs: Example programs compute meaningful values based on input and print the results for users through simple output mechanisms. OOP textbooks deal with data modeling on one hand, and demonstrate specific

language features on the other. Even though message passing structures may be complex, their net effects are trivial from the user's perspective. Finally, functional programming textbooks stress data manipulation techniques. Thus, the orientation (i.e., what programs are for) is very different in these paradigms.

This finding also means that students' tasks are different depending on the programming paradigm used for learning. In procedural programming, students try to write programs that *do* meaningful actions and computations, whereas in OOP students concentrate on creating *conceptual models* for (usually concrete) data. Détienne (1997) notes that when novices design OO programs, the activity of finding classes consumes their attention; they think about functionality only late in the design activity. Ebrahimi and Schweikert (2006) found that students have problems in understanding object-orientation and incorporating OO concepts into problem solving. Students tend to spend more time trying to understand objects and less time on problem solving. Thus, the shift to object-orientation has made a revolution at the orientation level and regarding students' tasks in programming.

RESEARCH SUPPORT

In the previous sections, we have demonstrated that the shift from imperative and procedural programming education to OOP has denoted a revolution in the complexity of notations, concepts and the notional machine needed, and in the orientation and tasks carried out by students as programming exercises. In this section, we will look at research literature[3] and see what it says about this revolution.

Imperative and Procedural Programming

Classic works on programming education and the psychology of novice and expert programming (e.g., Brooks, 1983; Corritore & Wiedenbeck, 1991; Davies, 1993; Gilmore & Green, 1984; Letovsky, 1986; Pennington, 1987; Perkins & Martin, 1986; Rist, 1989; Soloway & Spohrer, 1989; see also Robins et al., 2003, and Winslow, 1996, for excellent reviews) are primarily based on imperative and, to some extent, also procedural programming—in many cases Pascal programming, which is why we used Pascal in Figure 3. It is evident from this literature that learning programming is challenging even in the imperative case. Novices often have problems understanding basic concepts, such as variables and basic imperative control structures (Ben-David Kolikant & Haberman, 2001; Samurçay, 1989; Spohrer, Soloway, & Pope, 1989)—that is, they have problems in understanding the basic notional machine required for imperative programming.

Novices' knowledge about the imperative parts of programming languages has been found to be at first fragile (Perkins & Martin, 1986), such as inert knowledge that students cannot readily master, or misplaced knowledge migrated to inappropriate contexts. As a consequence, students have problems in applying their knowledge even though the knowledge itself may be correct. From a cognitive perspective, the causes of fragile knowledge include a sparse network of associations in long-term memory, that is, weak connections between different concepts, and underdifferentiation of language commands. Yet, the hardest part of learning is not in grasping the syntax and semantics of some

language, but in adopting ways to construct larger program units that are needed to solve the problem at hand (see, e.g., Winslow, 1996).

A specific source of problems is the limited capacity of working memory (Anderson, 2000, p. 176). Even when writing simple imperative programs consisting of just a few lines, expert programmers—let alone novices—often cannot form a complete mental representation of the program in their working memory. Even with the help of external representations, the number of simultaneously needed details easily exceeds the limitations of human working memory (Green, Bellamy, & Parker, 1987). Highly economical chunking of knowledge is therefore crucial for good performance in programming. Because novices' programming knowledge is fragile, efficient chunking is difficult for them.

In summary, educational and psychological research into novice imperative and procedural programming indicates that even the simplest imperative notional machine is challenging for students to learn, students' knowledge is fragile, and students have serious problems in combining basic constructs of a programming language to form larger, meaningful structures.

Object-Oriented Programming

Very little psychological and educational research exists for novice OOP. Most papers (e.g., Bennedsen & Caspersen, 2004; Bierre et al., 2006; S. Cooper et al., 2003; Holliday & Luginbuhl, 2004; Hsia et al., 2005; Kölling & Henriksen, 2005; Lopez-Herrejon & Schulman, 2004; Mahmoud et al,. 2004; Marrero & Settle, 2005; Shanmugasundaram et al., 2006; Truong et al., 2005; Utting, 2006) introduce various pedagogic techniques and tips, such as visualization tools or curriculum changes, without consideration for educational or psychological theories. Some (e.g., Bednarik & Tukiainen, 2007; Romero, Lutz, Cox, & du Boulay, 2002) study the use of such tools in the context of an OOP language but not relating their findings to OO concepts or the OO paradigm. Only very few articles (see Tables 1 and 2) analyze object-orientation from a cognitive or educational perspective, that is, increase the field's understanding of OOP learning and how it differs from the imperative and procedural cases. We will next review these results.

Davies, Gilmore, and Green (1995) asked novices and experts to sort cards containing short fragments of a large OO program library and found that experts tended to focus on functional relations whereas novices were much more concerned with objects and inheritance relations. Thus, novices' mental representations of the structure of large OO programs concentrates on objects and inheritance, that is, on elements that do not exist in the procedural case. Corritore and Wiedenbeck (1999) and Wiedenbeck, Ramalingam, Sarasamma, and Corritore (1999) have studied novices and experts comprehending short procedural and OO programs and found that, in the OO case, the overall function of programs is understood better than details of, for example, control flow; yet with procedural programs, comprehenders' knowledge is more balanced. These results indicate that programmers' mental representations of procedural and OO programs do differ qualitatively. As the nature of mental representations is strongly related with learning programming, this finding proposes the existence of fundamental differences between learning procedural programming and learning OOP.

Eckerdal and Thuné (2005) have studied novices' understanding of class and object and found several categories of conception of these concepts. Détienne (1997), Holland, Griffiths,

and Woodman (1997), Ragonis and Ben-Ari (2005), and Teif and Hazzan (2006) have found that students have severe misconceptions about fundamental OO concepts, such as classes and inheritance. Fleury (2000) has found several misconceptions concerning the construction and use of objects in Java. In procedural programming, misconceptions about parameter passing (Fleury, 1991) and recursion (Levy, 2001) have been found; in imperative programming only fragile knowledge instead of misconceptions has been reported. In consequence, problems in learning seem to have different roots in OOP than in imperative programming.

Table 1. Psychological and Educational Research on OOP: Mental Representation

Topic of investigation	Expert performance	Novice performance	Cognitive development	Programming education
Notional machine/ structure				
Notional machine/ detailed contents				
Notional machine/ misconceptions				
OO programs/ structure	Davies et al. (1995)	Davies et al. (1995)		
OO programs/ detailed contents	Corritore and Wiedenbeck (1999)	Wiedenbeck et al. (1999)		
OO programs/ misconceptions				
OOP/ structure				
OOP/ detailed contents		Eckerdal and Thuné (2005)		Mead et al. (2006)
OOP/ misconceptions		Détienne (1997); Fleury (2000); Holland et al. (1997); Ragonis and Ben-Ari (2005); Teif and Hazzan (2006)		

Table 2. Psychological and Educational Research on OOP: Skills and Strategies.

Topic of investigation	Expert performance	Novice performance	Cognitive development	Programming education
Program comprehension				
Tracing and debugging		Lister et al. (2004); Vainio and Sajaniemi (2007)		Thomas et al. (2004)
Program design	Détienne (1997); Lee and Pennington (1994); Pennington et al. (1995); Rosson and Gold (1989)	Détienne (1997)		

Mead et al. (2006) have compared cognitive problems in learning procedural and OOP and developed a set of central concepts in the form of "anchor concept graphs" for both paradigms. The two graphs differ considerably, providing more evidence for the assumption that learning procedural programming and learning OOP are very different in nature.

Thomas, Ratcliffe, and Thomasson (2004) found that students did not perform better in tracing OO code fragments when they were provided with ready-made partial object diagrams, nor did they draw their own diagrams more often in a follow-up test. On the other hand, Lister et al. (2004) found that many students were able to track values of numeric variables on paper, and Vainio and Sajaniemi (2007) found that students were able to draw values of primitive types, but not object references. Taken together, these results imply that students have more problems in making external representations of OO parts than imperative parts of the notional machine, that is, the OO notional machine is even more poorly understood by students than the imperative notional machine.

In her state-of-art review of empirical research on object-oriented design, Détienne (1997) examined the processes involved in designing in the OO paradigm and in the procedural paradigm. Among other things, she reports on findings of Lee and Pennington (1994), Pennington, Lee, and Rehder (1995), and Rosson and Gold (1989) concerning the differences between OO designers and procedural designers. OO designers seem to base their solutions on the problem domain itself, whereas procedural designers use generic programming constructs for structuring their solutions. Thus, the overall approach in program design differs between procedural and OO programming, and teaching should acknowledge this difference.

Discussion

Even though studies into OOP are few, the above results make it clear that both OOP itself and learning OOP are very different from their imperative and procedural counterparts: Mental representation of programs is different, problems have different roots, conceptual contents of knowledge are different, the level of understanding the underlying notional machine is different, and the overall approach to program design is different. These differences are so fundamental to learning that we dare to claim that the classic educational and cognitive results of novice imperative and procedural programming should not be used in the OO context.

Furthermore, the number of educational and cognitive studies of learning OOP is small. Lister et al. (2006) studied several popular claims about learning OOP and found practically no evidence for them in scientific literature. Neither do we know of any results that would provide evidence for the desirability or efficiency of replacing imperative/procedural programming education by object-orientation. On the contrary, Chen, Monge, and Simon (2006) found no effects of the first programming paradigm and later design skills; Détienne (1997), Pennington et al. (1995), and Sharp and Griffyth (1999) found positive transfer effects of traditional structured and procedural approaches to OO design.

PROPOSAL FOR RESEARCH AGENDA

Tables 1 and 2 draw together OOP research described in the previous section. We have tabulated research articles according to two dimensions: the first describing the cognitive

content or skill targeted in an investigation, the second telling whether the investigation deals with experts' performance, novices' performance or problems, development of novices' mental representations and skills, or ways to improve this development with educational techniques. The tables make it clear that large areas are totally neglected: Even the most researched areas—novices' misconceptions in OOP knowledge and experts' program design processes—have been studied in only a few papers.

If novices are to be helped in their struggles when learning OOP, it is necessary to know their problems and misconceptions as well as what experts know and how they apply their knowledge. Only then can efficient teaching methods and contents that have a strong cognitive basis be devised. Many studies in traditional programming have compared expert and novice performance and mental representations, thus providing information on what distinguishes experts from novices. In the OO domain, such studies are rare; only two studies in Tables 1 and 2 (Davies et al., 1995; Détienne, 1997) cover both experts and novices. We therefore suggest that research into *expert and novice differences* should be carried out in all cognitive aspects listed in the tables.

A notable gap in Table 1 covers the OO notional machine. There are no studies on experts' or novices' understanding of the notional machine behind OOP; neither are there studies on teaching a viable notional machine to students. Some suggestions have been presented for visualizing OO program execution (e.g., Gries & Gries, 2002; Moreno, Myller, Sutinen, & Ben-Ari, 2004; Sajaniemi, Byckling, & Gerdt, 2006), but their correspondence to experts' or novices' mental representations or their efficiency in providing a mental model of a correct notional machine has not been studied in detail. In a recent study (Sajaniemi, Kuittinen, & Tikansalo, 2007), students were found to be poor in visualizing relationships between objects and method calls during program execution and students' understanding of these relationships (i.e., the structure of the notional OO machine) was found to contain many errors. We therefore suggest that *experts' mental representations of the notional OO machine* should be studied in detail. Moreover, effective *ways to convey this knowledge to novices* should also be investigated.

Another gap in Tables 1 and 2 is the lack of studies into the cognitive development of novices' mental representations and skills. In order to support learning by teaching, steps in cognitive development must first be known. Basic cognitive activities—such as chunking—do, of course, appear in the context of OOP as well. However, the building of the notional machine, construction of OOP knowledge, and detailed development of OOP skills and strategies presumably have components that are specific to OOP. We therefore suggest that *novices' cognitive development in OOP* should be studied.

Investigations of mental representations of OO programs (Corritore & Wiedenbeck, 1999; Wiedenbeck et al., 1999) have probed participants' knowledge with yes/no questions divided into categories determined by the researchers a priori. Such a method reveals whether participants possess knowledge in those categories but it does not reveal what other types of knowledge they might have. As a consequence, exact contents of experts' mental representations of OO programs are largely unknown and teachers have only vague ideas of how to best explain important program elements and their relationships to students. We therefore call for *exploratory research into experts' mental representations of OO programs.*

Studies in cognitive processes, such as skills and strategies, cover mainly experts' program design. In imperative programming, research into experts' and novices' program

comprehension has increased our understanding of the comprehension processes and, moreover, of the mental representations of imperative programs and imperative programming knowledge. The structure of OO programs differs so much from imperative and procedural programs that one may presume that their comprehension processes do also differ considerably. Again, some elements (e.g., hypothesis-driven comprehension) are the same, but issues related to program structure can be assumed to differ. We therefore suggest research into *experts' and novices' OO program comprehension processes.*

Finally, results of the research suggested above and summarized in Table 3 should be utilized in devising effective methods for teaching OOP. However, we do not include this work in the research agenda proposal for two reasons. Firstly, the right time for such educational-oriented research will come only after there is a large body of results obtained from the research agenda. Secondly, it may well be that effective ways to transfer experts' mental representations, skills and strategies are at least partially revealed during the earlier research covered by the agenda.

Table 3. Proposal for Research Agenda in OOP and OOP Education.

	Performance			Development	
Topic of investigation	**Expert**	**Expert vs. Novice**	**Novice**	**Cognition**	**Education**
Mental representation of notional machine	•	•		•	•
Mental representation of OO programs	•	•		•	
Mental representation of OOP		•		•	
Program comprehension	•	•	•	•	
Tracing and debugging		•		•	
Program design		•		•	

CONCLUSION

In programming education, there has been a major shift in the programming paradigm used in the first courses. To please industry and students, educators have moved from imperative and procedural programming to object-orientation without studying its necessity or consequences and without studying how OOP education should be carried out. Moreover, classic results from imperative and procedural programming have been used as such even though their applicability in the OO case can be questioned. The shift from imperative/procedural

programming to object-orientation is so revolutionary that the use of research results obtained in the imperative and procedural cases is doubtful in the OO case. The number of notations and concepts needed, the size of the notional machine required, and the whole orientation of programming are so different that the basic assumptions used in imperative and procedural programming research do not necessarily hold for object-orientation. Even though some results may apply in object-orientation, there is a need to find out on what occasions this happens to be the case.

There is a lack of systematic research into the fundamental cognitive and educational issues in learning and teaching OOP. Lister et al. (2006, p. 160) conclude their paper by noting that "our community needs to discuss—and debate—this issue," but we claim that the computer science education research community and the psychology of programming community need to rigorously study these issues first. For that purpose, we have presented a research agenda comprising

- *Constructing a model of the OOP expert:* experts' mental representations of the notional OO machine; exploratory research into experts' mental representations of OO programs
- *Understanding the differences between OOP experts and novices:* experts' and novices' differences in mental representations, program comprehension processes, skills and strategies within OOP
- *Fostering OOP novices' cognitive development:* novices' cognitive development in OOP; ways to convey the notional OO machine to novices

High dropout rates from OOP courses and poor learning outcomes pose problems to students, educators, and educational institutions. These problems can be attacked only with rigorous research into the psychological and educational issues involved.

ENDNOTES

1. Imperative and procedural programming are often considered synonyms, but in this paper *imperative* refers to programming with variables, assignment, and simple imperative control structures, such as sequence, iteration, and conditionals, whereas *procedural* covers procedures, parameters and recursion, also.
2. Here we are interested in differences that are inherent to object-orientation and the way object-related concepts are implemented in Java. We do not treat Java problems that occur within the imperative parts of Java, for example, that using "=" as the assignment operator makes some students to confuse assignment with mathematical equality.
3. In this literature review, we look at programming only. Thus, we do not include system design literature even though we do include program design literature.

REFERENCES

Anderson, J. R. (2000). *Cognitive psychology and its implicatio*ns (5th ed.). New York: Worth Publishers.

Bednarik, R., & Tukiainen, M. (2007). Analysing and interpreting quantitative eye-tracking data in the studies of programming: Phases of debugging with multiple representations. In J. Sajaniemi, M. Tukiainen, R. Bednarik, & S. Nevalainen (Eds.), *Proceedings of the 19th Annual Workshop of the Psychology of*

Programming Interest Group (pp. 158–172). Joensuu, Finland: University of Joensuu, Department of Computer Science and Statistics.

Ben-David Kolikant, Y., & Haberman, B. (2001). Activating "black boxes" instead of opening "zippers": A method of teaching novices. In *ITiCSE '01: Proceedings of the Sixth Annual Conference on Innovation and Technology in Computer Science Education* (pp. 41–44). New York: ACM Press.

Bennedsen, J., & Caspersen, M. E. (2004). Programming in context: A model-first approach to CS1. In *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (pp. 477–481). New York: ACM Press.

Bierre, K., Ventura, P., Phelps, A., & Egert, C. (2006). Motivating OOP by blowing things up: An exercise in cooperation and competition in an introductory Java programming course. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (pp. 354–358). New York: ACM Press.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*, 534–554.

Cassel, L. B., McGettrick, A., Guzdial, M., & Roberts, E. (2007). The current crisis in computing: What are the real issues? In *SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 329–330). New York: ACM Press.

Chen, T.-Y., Monge, A., & Simon, B. (2006). Relationship of early programming language to novice generated design. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (pp. 495–499). New York: ACM Press.

Cooper, D., & Clancy, M. (1982). *Oh! Pascal!* New York: W. W. Norton & Company.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (pp. 191–195). New York: ACM Press.

Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, *3*, 199–222.

Corritore, C. L., & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, *50*, 61–83.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*, 237–267.

Davies, S. P., Gilmore, D. J., & Green, T. R. G. (1995). Are objects that important? Effects of expertise and familiarity on classification of object-oriented code. *Human-Computer Interaction*, *10*, 227–248.

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

Détienne, F. (2002). *Software design: Cognitive aspects*. London: Springer-Verlag.

de Raadt, M., Watson, R., & Toleman, M. (2002). Language trends in introductory programming courses. In E. Cohen & E. Boyd (Eds.), *Proceedings of Informing Science and IT Education Conference* (pp. 329–337). Santa Rosa, CA, USA: Informing Science Institute.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 283–299). Hillsdale, NJ, USA: Lawrence Erlbaum Associates.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies*, *14*, 237–249.

Ebrahimi, A., & Schweikert, C. (2006). Empirical study of novice programming with plans and objects. *SIGCSE Bulletin*, *38*(4), 52–54.

Eckerdal, A., & Thuné, M. (2005). Novice Java programmers' conceptions of "object" and "class", and variation theory. In *ITiCSE '05*: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 89–93). New York: ACM Press.

Fleury, A. E. (1991). Parameter passing: The rules the students construct. *SIGCSE Bulletin*, *23*(1), 283–286.

Fleury, A. E. (2000). Programming in Java: Student-constructed rules. In *SIGCSE '00: Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education* (pp. 197–201). New York: ACM Press.

Gilmore, D. J., & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies, 21*, 31–48.

Green, T. R. G., Bellamy, R. K. E., & Parker, J. M. (1987). Parsing and gnisrap: A model of device use. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 132–146). Norwood, NJ, USA: Ablex Publishing Company.

Gries, P., & Gries, D. (2002). Frames and folders: A teachable memory model for Java. *The Journal of Computing Sciences in Colleges, 17*(6), 182–196.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *SIGCSE Bulletin, 29*(1), 131–134.

Holliday, M. A., & Luginbuhl, D. (2004). CS1 assessment using memory diagrams. In *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (pp. 200–204). New York: ACM Press.

Hsia, J. I., Simpson, E., Smith, D., & Cartwright, R. (2005). Taming Java for the classroom. In *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 327–331). New York: ACM Press.

Kinnunen, P., & Malmi, L. (2006). Why students drop out CS1 course? In *ICER '06: Proceedings of the 2006 International Workshop on Computing Education Research* (pp. 97–108). New York: ACM Press.

Kölling, M., & Henriksen, P. (2005). Game programming in introductory courses with direct state manipulation. In *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 59–63). New York: ACM Press.

Lee, A., & Pennington, N. (1994). The effects of programming on cognitive activities in design. *International Journal of Human-Computer Studies, 40*, 577–601.

Letovsky, S. (1986). Cognitive processes in program comprehension. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 58–79). Norwood, NJ: Ablex Publishing Company.

Levy, D. (2001). Insights and conflicts in discussing recursion: A case study. *Computer Science Education, 11*, 305–322.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119–150.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., & Whalley, J. L. (2006). Research perspectives on the objects-early debate. *SIGCSE Bulletin, 38*(4), 146–165.

Lopez-Herrejon, R. E., & Schulman, M. (2004). Using interactive technology in a short Java course: An experience report. In *ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 203–207). New York: ACM Press.

Mahmoud, Q. H., Dobosiewicz, W., & Swayne, D. (2004). Redesigning introductory computer programming with HTML, JavaScript, and Java. In *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (pp. 120–124). New York: ACM Press.

Marrero, W., & Settle, A. (2005). Testing first: Emphasizing testing in early programming courses. In *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 4–8). New York: ACM Press.

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David Kolikant, Y., Laxer, C., Thomas, L., & Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125–140.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *SIGCSE Bulletin, 38*(4), 182–194.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. In *AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces* (pp. 373–376). New York: ACM.

Pennington, N. (1987). Comprehension strategies in programming. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Company.

Pennington, N., Lee, A., & Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction*, *10*, 171–226.

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 213–229). Norwood, NJ, USA: Ablex Publishing Company.

Radenski, A. (2006). "Python first": A lab-based digital introduction to computer science. In *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 197–201). New York: ACM Press.

Ragonis, N., & Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, *15*, 203–221.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, *13*, 389–414.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*, 137–172.

Romero, P., Lutz, R., Cox, R., & du Boulay, B. (2002). Co-ordination of multiple external representations during Java program debugging. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (pp. 207–214). Los Alamitos, CA, USA: IEEE Computer Society Press.

Rosson, M. B., & Gold, E. (1989). *Problem-solution mapping in object-oriented design*. New York: IBM T. J. Watson Research Center.

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., & Kulikova, Y. (2006). Roles of variables in three programming paradigms. *Computer Science Education*, *16*, 261–279.

Sajaniemi, J., Byckling, P., & Gerdt, P. (2006). Metaphor-based animation of OO programs. In *SoftVis '06: Proceedings of the ACM Symposium on Software Visualization* (pp. 173–174). New York: ACM Press.

Sajaniemi, J., Kuittinen, M., & Tikansalo, T. (2007). A study of the development of students' visualizations of program state during an elementary object-oriented programming course. In *ICER '07: Proceedings of the Third International Workshop on Computing Education Research* (pp. 1–15). New York: ACM Press.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 161–178). Hillsdale, NJ, USA: Lawrence Erlbaum Associates.

Shanmugasundaram, V., Juell, P., & Hill, C. (2006). Knowledge building using visualizations. In *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 23–27). New York: ACM Press.

Sharp, H., & Griffyth, J. (1999). The effect of previous software development experience on understanding the object-oriented paradigm. *Journal of Computers in Mathematics and Science Teaching*, *18*, 245–265.

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the novice programmer*. Hillsdale, NJ, USA: Lawrence Erlbaum Associates.

Spohrer, J. C., Soloway, E., & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 355–399). Hillsdale, NJ, USA: Lawrence Erlbaum Associates.

Teif, M., & Hazzan, O. (2006). Partonomy and taxonomy in object-oriented thinking: Junior high school students' perceptions of object-oriented basic concepts. *SIGCSE Bulletin*, *38*(4), 55–60.

Thomas, L., Ratcliffe, M., & Thomasson, B. (2004). Scaffolding with object diagrams in first year programming classes: Some unexpected results. In *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (pp. 250–254). New York: ACM Press.

Truong, N., Bancroft, P., & Roe, P. (2005). Learning to program through the web. In *ITiCSE '05*: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 9–13). New York: ACM Press.

Utting, I. (2006). Problems in the initial teaching of programming using Java: The case for replacing J2SE with J2ME. In *ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 193–196). New York: ACM Press.

Vainio, V., & Sajaniemi, J. (2007). Factors in novice programmers' poor tracing skills. In *ITiCSE '07: Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 236–240). New York: ACM Press.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*, 255–282.

Winslow, L. E. (1996). Programming pedagogy: A psychological overview. *SIGCSE Bulletin*, *28*(3), 17–22.

Authors' Note

All correspondence should be addressed to:
Jorma Sajaniemi
University of Joensuu
P.O. Box 111
FI-80101 Joensuu
Finland
saja@cs.joensuu.fi

Human Technology: An Interdisciplinary Journal on Humans in ICT Environments
ISSN 1795-6889
www.humantechnology.jyu.fi

Human Technology:
An Interdisciplinary Journal on Humans in ICT Environments

www.humantechnology.jyu.fi