Attention and Program Knowledge in Visualising Roles of Variables

Seppo Nevalainen

Licenciate Thesis

May 16, 2006

Department of Computer Science,
University of Joensuu, Finland

Abstract

One application area for visualisations in computer science is teaching of computer programming. So far, the effectiveness of these visualisations has been evaluated empirically mostly by measuring their long-term effects on learning programming, excluding possible short-term effects of visualisations and their relation to the long-term effects. In our research, we will study short-term effects when visualising the roles of variables with PlanAni program animator to novices learning programming. This thesis presents the research methodology and its evaluation, and describes the first of the experiments we will carry out during our research.

First, we conducted an experiment comparing the use of three eye tracking devices in the psychology of programming. On the basis of the results, we selected Tobii 1750 for measuring the eye movements in our research. Second, we analysed the replicability of Good's program summary analysis scheme. The investigation brought forth some problems, that are reported. Third, we studied how a person targets her visual attention, and what kind of a mental model she constructs concerning a computer program, when the program and especially its variables are presented using either a textual or a graphical program visualisation tool.

Preface

I would like to thank ...

My supervisors professor Jorma Sajaniemi and senior lecturer Marja Kuittinen for their valuable guidance through these years without which the writing of this thesis would not have been possible.

Pauli Byckling and Petri Gerdt for enjoyable and educational conversations about life, the universe and everything, and for the collaboration in cognitive science studies and various research questions.

People at the department office and at the department library for making all the practical matters much easier.

Faculty of science at the University of Joensuu for providing a grant to support my postgraduate studies, and the department of computer science at the University of Joensuu for providing several short-term positions to support my postgraduate studies.

This work was supported by the Academy of Finland under grant number 206574.

Contents

Chapter 1

Introduction

People have used visualisations for achieving various tasks throughout their history. For example generals have utilised maps in warfare and sailors have utilised star charts during their travels. During the computer era, numerous visualisation tools have been developed to be used in computer science education for tasks such as teaching programming and algorithms (e.g., ALADDIN (Helttula et al., 1989), ZEUS (Brown, 1991), Jeliot (Haajanen et al., 1997), Eliot (Lahtinen et al., 1998), DISCOVER (Ramadhan, 2000), ANIMAL (Rössling and Freisleben, 2002)), and in the software industry for tasks such as debugging and maintaining large programs (e.g., FIELD (Reiss, 1998), ZStep 95 (Lieberman and Fry, 1998), SeeSoft (Eick, 1998), PV (Kimelman et al., 1998)).

In computer science, the term visualisation can refer to situations, where people form mental representations concerning for example algorithms or computer programs on the basis of information provided by a visualisation tool, or to the representation of the information in the tool, depending on the context (Baecker and B.Price, 1998; Hundhausen et al., 2002; Price et al., 1993).

One application area for visualisations in computer science is teaching of computer programming, where the goal is to help students in evolving from programming novices to experts. This is challenging, because programs deal with abstract entities—formal looping constructs, pointers going through arrays etc.—that have little to do with everyday issues, and that make learning to program difficult for many students. Visualisation tools can be used for example to make programming language constructs and program constructs more comprehensible (Hundhausen et al., 2002; Mulholland,

1998), and to illustrate expert programmers' reasoning processes to the novices (Petre et al., 1998).

A designer of an effective visualisation tool has to carefully choose the target of a visualisation, and decide how it should be presented to the viewer. Variables are a good candidate for the target of visualisation, since they are central to the comprehension of computer programs. Programs consist of variables, operations on variables, and larger program constructs, such as functions, classes, and modules. In a study by von Mayrhauser and Vans (1995), information about variables was the most frequent information need type among professional maintenance programmers. Several taxonomies and frameworks (Myers, 1990; Stasko and Patterson, 1992; Price et al., 1993; Green and Petre, 1996; Ainsworth and Labeke, 2002) can be utilised in searching answers for the question how information should be presented in order for a visualisation to be effective.

Visualisation tools' effectiveness depends on their ability to guide people's thinking into a desired direction by transforming information into a meaningful and useful (usually pictorial) representation, that helps the viewer to gain more understanding of the target. Creating effective representations is not straightforward, since the human mind is a "black box", often even to the people themselves. The designer of the visualisations has to rely therefore on methods from for example psychology of programming and cognitive psychology when evaluating the visualisations' ability to produce the desired mental representations in the viewer.

So far, visualisations have been evaluated empirically mostly by measuring the performance of participants that have been shown the visualisation in relation to other participants, who have been shown similar material without visualising it. Performance has been measured either by comparing the scores of participants after the viewing, or by comparing the change in the scores of the participants between before and after the viewing. These post-tests and pre- versus post-tests of participants's performance have resulted in a body of evidence suggesting that properly designed visualisations can have beneficial effects on the learning in the students (Hundhausen et al., 2002; Byrne et al., 1999b; Hansen et al., 2000; Kann et al., 1997). However, these post-test or pre- versus post-test evaluations do not provide clear insight into the possible short-term effects of visualisations and their relation to the long-term effects. In other words, the results tell us if the visualisations of an individual visualisation tool are being helpful of unhelpful, but they do not tell us much about what effects take place during the use

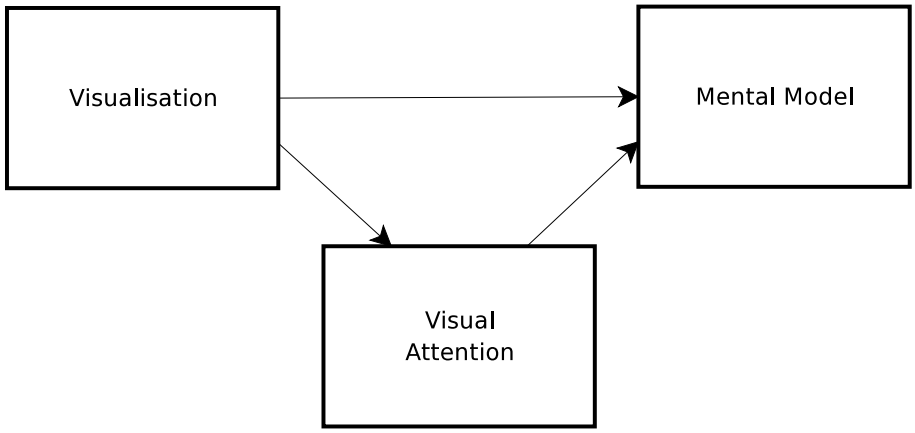

Figure 1.1: Interaction between visualisation, visual attention and mental model. The content of the mental model may be influenced both by the content of the visualisation, and by the distribution of the visual attention between different parts of the visualisation.

of the visualisation tool and how these effects build up into the overall helpfulness or unhelpfulness of the tool.

In our research, we will study these short-term effects when visualising the roles of variables. Roles of variables is a cognitive concept that has been developed by Sajaniemi (2002), and that can be utilized in teaching programming to novices. Roles of the variables can be visualised with PlanAni program animator (Sajaniemi and Kuittinen, 2004), in which the data contained in the variables and the code of a computer program are animated concurrently. The goal of the visualisation is to portray expert's tacit knowledge concerning the behavior of variables during program execution in a way that is accessible to novice programmers.

Earlier research has found support for positive long-term effects of the visualisation of roles for learning (Byckling and Sajaniemi, 2005; Sajaniemi and Kuittinen, 2005). Our aim is to find answers to the questions what short-term effects the visualisation of roles has on the viewer and how these relate to the overall benefit of the visualisation. This would help us understand better the detailed mechanisms through which the visualisation of roles gains its usefulness and would provide some possible answers to the general question of which visualisations are truly helpful for learning programming.

The context in which we conduct our research will be the visualisation of the roles of variables during introductory programming courses. The viewers of the visualisations

will be novice-level students, who are at the start of their programming studies. We will concentrate on investigating the interaction between the form of visualisations, the visual attention of the students, and the mental models students form concerning computer programs when they view programs with PlanAni program animator (Figure 1.1). As Figure 1.1 indicates, the content of the mental model may be influenced both by the content of the visualisation, such as images used to represent the roles, and by the distribution of the visual attention between different parts of the visualisation, such as animations and code.

Information concerning visual attention will be measured using eye tracking, and it can be used to provide insight into what the students found interesting, and possibly even to provide a clue as to how they perceived whatever scene they were viewing (Duchowski, 2003). The contents of the students' mental models will be investigated by using Good's program summary analysis scheme (Good, 1999) to analyse the program summaries students form concerning the visualised programs. The contents of the students' mental models can be used to characterize the quality of comprehension (Byckling et al., 2004).

In this thesis, I will give a description of the roles of variables that will be used as the main target of visualisation in our research. I will also present our research methodology, that is eye tracking and program summary analysis. During our research, I will conduct a series of experiments, in which the interaction between visual attention and program knowledge in visualising roles of variables will be investigated in detail. In this thesis, first of these experiments is described.

My research contributions are to:

- Provide an empirical analysis concerning the appropriateness of eye tracking as a psychology of programming research method.
- Evaluate the inter-rater reliability of Good's program summary analysis scheme.
- Conduct an empirical investigation of the short-term effects of visualising variables with a textual and a graphical visualisation tool.

The rest of the thesis is organised as follows. In chapter 2 I will present the roles of variables, PlanAni program animator that visualises them, and earlier results concerning the long-term effects of visualising the roles. Chapter 3 focuses on visual attention,

on utilisation of eye tracking in measuring visual attention, and on the question how appropriate eye tracking is as a psychology of programming research method. Chapter 4 speaks about mental models, about program summaries as a means for studying mental models, and about Good's program summary analysis scheme, that will be a part of our research methodology. In chapter 5 I will describe an empirical experiment that we carried out in order to study the short-term effects of visualising the roles of variables and will present the results. Chapter 6 contains the conclusion.

Chapter 2

Roles of Variables

The roles of variables are chosen as the main target of visualisations in our research on visual attention and program knowledge. The purpose of this chapter is to give reasons that support this choice and to familiarise the reader with the role concept. First, I will provide some background for the roles of variables and present its central ideas. Then I will discuss the visualisation of the roles and their use in teaching elementary programming. I will end the chapter with a presentation of the main results from the empirical evaluation of the roles of variables concept as a pedagogical tool. This chapter is based on the work of Sajaniemi (2002), Sajaniemi and Kuittinen (2003), Sajaniemi and Kuittinen (2004), Kuittinen and Sajaniemi (2004), Sajaniemi and Kuittinen (2005), Byckling and Sajaniemi (2005), and Sajaniemi et al. (Submitted).

2.1 Background on Roles of Variables

The knowledge about variables is central to the comprehension of computer programs. Programs consist of variables, operations on variables, and larger program constructs, such as functions, classes, and modules. The importance of variables in comprehension of programs is supported by for example a study by von Mayrhauser and Vans (1995), in which information about variables was found to be the most frequent information need type among professional maintenance programmers.

In the research literature, the knowledge about variables has been presented either as related to unique variables in unique programs or as related to variables on a general

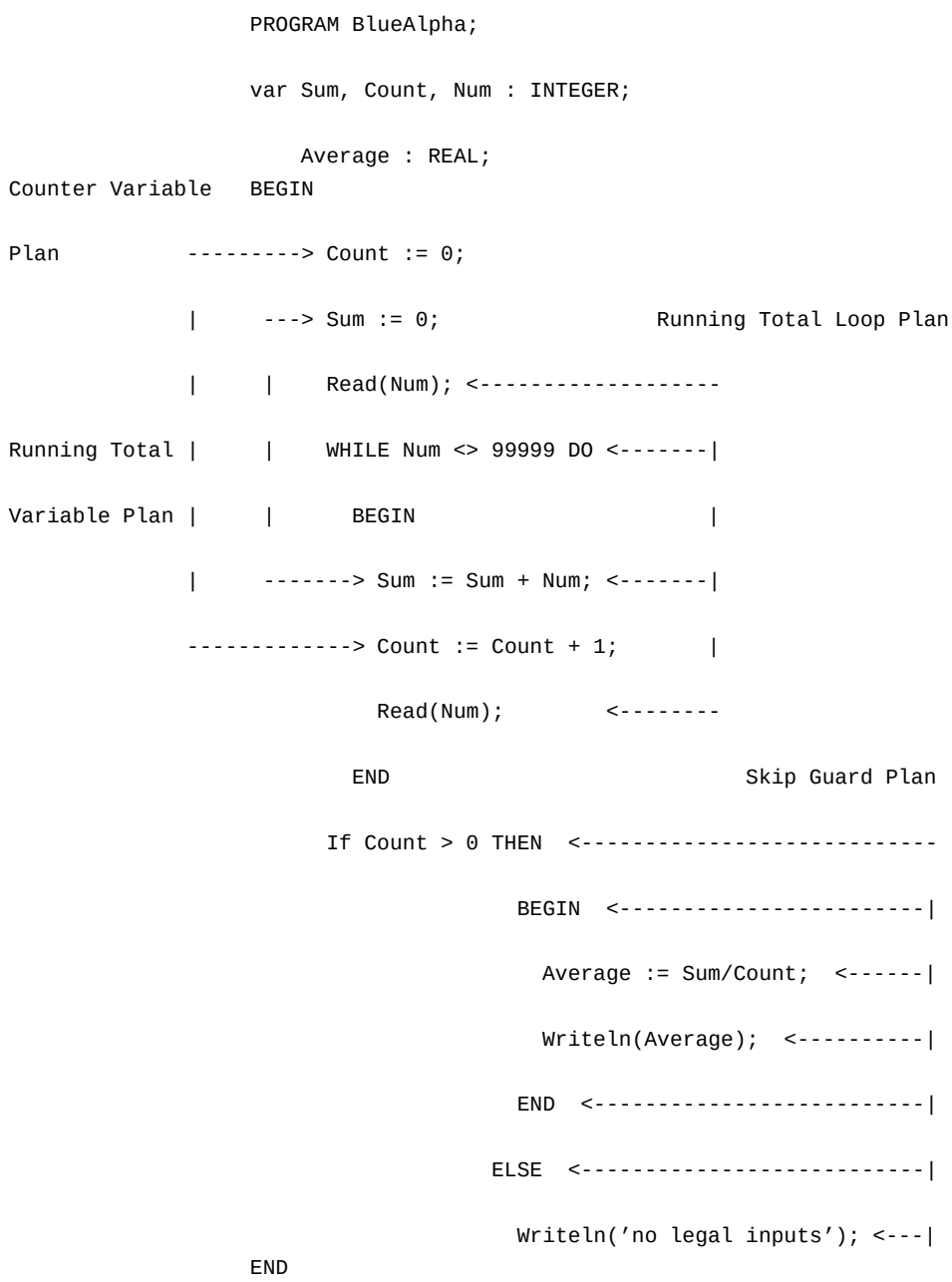

Figure 2.1: A sample program: The running total loop plan (Ehrlich and Soloway, 1984).

programming knowledge -level, in such a way that the knowledge applies to theoretically all variables in all programs. For example Brooks (1977, 1983), von Mayrhauser and Vans (1995), and Pennington (1987b) have used the first approach. General descriptions of the latter type are typically referred as programming plans, or schemas (von Mayrhauser and Vans, 1995; Ehrlich and Soloway, 1984; Pennington, 1987b).

Prior to the research on the roles of variables by Sajaniemi (2002), programming plans and schemas related to variables have been studied by for example Ehrlich and Soloway (1984), Rist (1991), and Green and Cornah (1985).

According to Ehrlich and Soloway (1984), variable plans consist of such aspects as the variable's role in the program (i.e. the function it serves), the way the variable is initialised and updated, and a guard that possibly protects the variable from invalid updates. A program example with variable plans is given in Figure 2.1.

In the example program, the variable *Count* serves as a counter variable, keeping track of the number of numbers read in, and *Sum* has a role of a running total variable, because it is accumulating the sum of the numbers read in. *Number* serves as a new value variable, holding the new number read in each time through the loop. Variables *Count* and *Sum* are initialised to 0. However, *Count* is updated by 1 through an assignment, and *Sum* is updated by the value of the new value variable through an assigment; *Number* is initialised and updated through a Read command. *Count* and *Sum* need to be protected from including the sentinel value (99999) in their respective totals. This guard is implemented by the test in the "while" loop of the program.

Table 2.1: Basic Pascal plan schemas (Rist, 1991).

	Initialization	Calculation	Output
Prompt	write ('Enter...');	read (number);	*value of number*
Label	write ('Output is...');	write (number);	*display*
Running total	count := 0;	count := count + 1;	*value of count*
Found	found := false;	if *test* then found := true;	*value of found*
Loop	for i := 1 to 30 do	read(rainfall);	*all rainfall values*

Rist (1991) has developed the idea further and defines a plan as a set of actions, that achieve a specific goal. Goal is defined by Rist as a state to be achieved, for example when calculating a value or a series of values. Rist defines five plans concerning the actions related to variables: *Prompt plan* to obtain input, *label plan* to produce output, *running total plan* to accumulate information, *found plan* to register an event, and *loop plan* to achieve iteration using loop counter variable. These basic plans are shown in Table 2.1. Neither Ehrlich and Soloway nor Rist claim their lists of examples to be exhaustive.

The approaches presented above have a strong cognitive basis and they are supported by empirical experimentation. A more practical approach to categorise variables has been given by Green and Cornah (1985), who have presented Programmer's Torch - tool with an effort to clarify the mental processes of maintenance programmers. One of the goals of the tool was to reveal different roles of variables. The variables were tentatively listed as: constant, counter, loop counter, most-recent holder, best-of holder, control variable, and subroutine variable.

The approaches discussed above consider both utilisation and behaviour of a variable in their descriptions. Later, Sajaniemi (2002) has defined the role of a variable to be dependent purely on the behaviour, not the use, of the variable, and found the following nine roles in novice-level procedural programs: constant, stepper, most-recent holder, most-wanted holder, gatherer, follower, one-way flag, temporary, and organizer. The role list has been supplemented later with a role called transformation (Ben-Ari and Sajaniemi, 2004).

2.2 The Role Concept

The roles of variables (Sajaniemi, 2002) is a new concept, by which the tacit knowledge experts have concerning the behaviour of variables can be presented explicitly. The roles of variables describe the use of variables on a general programming knowledge level, in such a way that it applies to theoretically all programs.

The purpose of the research on the roles of variables has been to develop a small but at the same time extensive list of roles, that covers in a concrete way all the variables found in novice-level programs, and which could be used for example in teaching elementary programming, or in analyzing large-scale programs.

The variables are used to handle the dataflow of a program. The purpose of an individual variable in a computer program is to work as a dynamic data element, that typically receives new values depending on other variables and external events. The role describes the nature and the behaviour of the variable during its lifetime and is not related to the use of the variable. Each role describes one stereotypical behaviour. In short, the role of a variable can be defined to be the dynamic character of a variable embodied by the succession of values the variable obtains, and how the new values

assigned to the variable relate to other variables (Sajaniemi, 2002).

To illustrate the roles of variables concept, consider following two lines of code, that output a one-dimensional array:

```
for i := 1 to 12 do
  write(Array[i]);
```

Here, variable i receives values from 1 to 12, known in advance, when the value of the variable is incremented in the for-loop. The role of a variable i, stepping through a systematic, predictable succession of values, is called stepper.

In another example, the two lines of code illustrate a typical way of collecting total sum:

```
TotalRainfall:=0
...
TotalRainfall:= TotalRainfall + Rainfall
```

Here, the TotalRainfall variable receives a new value when another value is added to its previous value. In other words, the variable gathers a total amount equal to the sum of several individual values. Therefore, the TotalRainfall variable can be said to have the role of a gatherer.

Ten roles presented in the Table 2.2 cover 99% of all variables found in novice-level imperative programs.

2.3 Visualising the Roles

There exists several different visualisation and animation tools that have been developed for facilitating teaching of programming (e.g., ZEUS (Brown, 1991), Jeliot (Haajanen et al., 1997), Eliot (Lahtinen et al., 1998), DISCOVER (Ramadhan, 2000), ANIMAL (Rössling and Freisleben, 2002)). The notion that these tools help the student in her struggle to learn programming is supported by for example Mayer (2001), who suggests that the power of animation in enhancing learning is based on its ability to

Table 2.2: The role set for novice-level programming (Sajaniemi et al., Submitted).

Role	Informal description
Fixed value	A variable that does not get a new proper value after its initialization.
Stepper	A variable stepping through a systematic, predictable succession of values.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input.
Most-wanted holder	A variable holding the best or otherwise most appropriate value encountered so far.
Gatherer	A variable accumulating the effect of individual values.
Follower	A variable that gets its new value always from the old value of some other variable.
One-way flag	A two-valued variable that cannot get its initial value once the value has been changed.
Temporary	A variable holding some value for a very short time only.
Transformation	A variable that gets its new value always with the same calculation from value(s) of other variable(s).
Organizer	An array used for rearranging its elements.

provide both verbal and pictorial information that are integrated in working memory to provide deeper understanding. Other suggested benefits that visualisations can provide for learning include for example the illustration of expert programmers' reasoning processes to the novice (Petre et al., 1998) and making the programming language constructs and program constructs more comprehensible (Hundhausen et al., 2002; Mulholland, 1998).

Current program visualisation tools can be divided into two categories: semi-automatic tools and hand-crafted tools (Sajaniemi and Kuittinen, 2004). This division is based on how much a program visualisation tool allows the user to influence the visual ap-

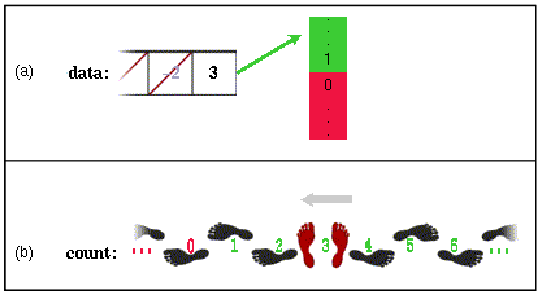

Figure 2.2: Two variable roles, most-recent holder (data) and stepper (count), and animations representing the comparison operation "$some\_variable > 0$" (Sajaniemi and Kuittinen, 2004).

pearance of variables. In semi-automatic tools, users select visualisations for variables from a set of ready-made visualisations. In hand-crafted tools, users make choices to reflect the value of a variable by choosing for example the appropriate size, color, and orientation for the visualisation. Program visualisation tools can use also predefined visualisations selected by the designer of the visualisation. The main effect of allowing users to participate in deciding the visual appearance of the visualisations is the increased interaction between the visualisation tool and the user; there is no guarantee of the appropriateness of the visualisation.

Many of the current visualisation tools represent variables and operations on variables in the program or programming language level. That is, they show what variables a program includes and the change of the values of these variables during program execution, treating each variable as an individual having at most programming language level abstractions (such as the type of the variable). According to Petre and Blackwell (1999), visualisations should not, however, work in the programming language level because within-paradigm visualisations, i.e., those dealing with programming language constructs, are uninformative.

The roles of variables represent knowledge that is not within the programming language paradigm but at an upper level, and they can be visualised with PlanAni program animator (Sajaniemi and Kuittinen, 2004). In PlanAni, visualisations of variables are predefined: each variable has a role image which is used also for the animation of operations on the variable. These role images represent the salient, stereotypical features of variables' behavior, and animations are focused on the role-like behavior of variables. For example, the role images of two roles, most-recent holder and stepper, and animations representing the comparison operation "$some\_variable > 0$", are shown

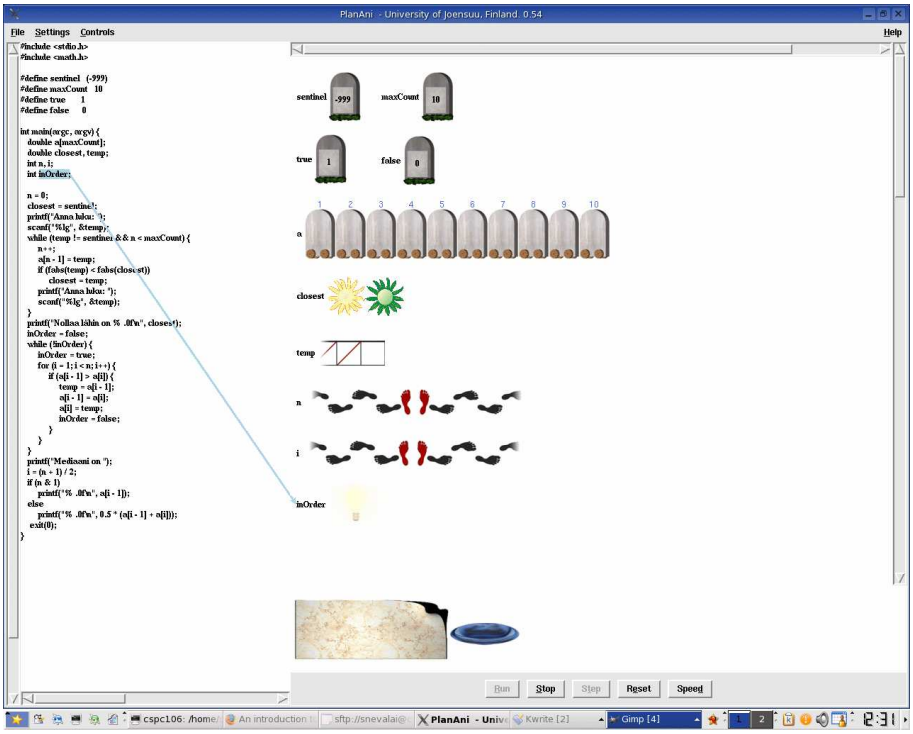

Figure 2.3: Graphical user interface of PlanAni.

in Figure 2.2.

PlanAni is designed to illustrate expert programmers' tacit knowledge related to the variables' behaviour during program execution in pictorial form, alongside the program code. These stereotypical features of behaviour are identified also by programming experts, who used them as one of their criteria, when grouping variables together (Sajaniemi and Navarro Prieto, 2005). The role images used in PlanAni have been evaluated empirically and found to enhance learning when compared with neutral control images (Stützle and Sajaniemi, 2005).

Figure 2.3 shows the user interface of PlanAni. The program code is located on the left side of the display. The role images and their animated operations are placed on the right side of the program code. The current phase of the program execution is higlighted with a blue rectangle, which is connected to the variable that is operated on with a blue arrow. The inputs and outputs of the program are displayed by a paper and a plate in the I/O area located on the bottom part of the display. The animation in PlanAni is controlled by buttons that are located below the I/O area. The buttons can be used to start, pause, stop or restart the animation, to execute the program lines one

at a time, and to change the speed of the animation.

2.4 Roles of Variables in Teaching Elementary Programming

Knowledge about computer programming covers the following three categories (Sajaniemi, 2004):

- **programming language knowledge:** the syntax and semantics of some certain language (e.g., how an assignment statement is written and what effect it has).
- **program knowledge:** knowledge about a specific program.
- **programming knowledge:** how to construct programs from abstract concepts within the programming paradigm in use (e.g., variables, iteration etc. in procedural programming). The most important type of knowledge for a programmer.

At the elementary stage of her studies, a student needs to increase her knowledge in several of the above levels. Usually, however, teaching of programming to novices concentrates around the syntax and semantics of a specific programming language, which can be categorised as low-level knowledge. The higher-level programming knowledge is usually not explicitly taught. The roles of variables represent higher-level programming knowledge and they can be applied to teaching elementary programming. Furthermore, the roles have been found to be easy to adopt in teaching (Kuittinen and Sajaniemi, 2004).

Figure 2.4 illustrates the relationships between the different roles of variables. Literal and constant are structures of a programming language, and other nodes in the figure represent different roles. When roles are used in teaching, a constructivist approach, suggested by the order present in the figure, can be applied. In this approach, new information is constructed upon previously learned information, as can be seen from Figure 2.4. For example, if variables's value does not change after it is set in initialisation, it can be described as a fixed value. However, if variable's values are set repeatedly in a loop, the variable can be described as a most-recent holder. Further, if the values set in the loop form a series, whose values are known in advance (for example in the case of

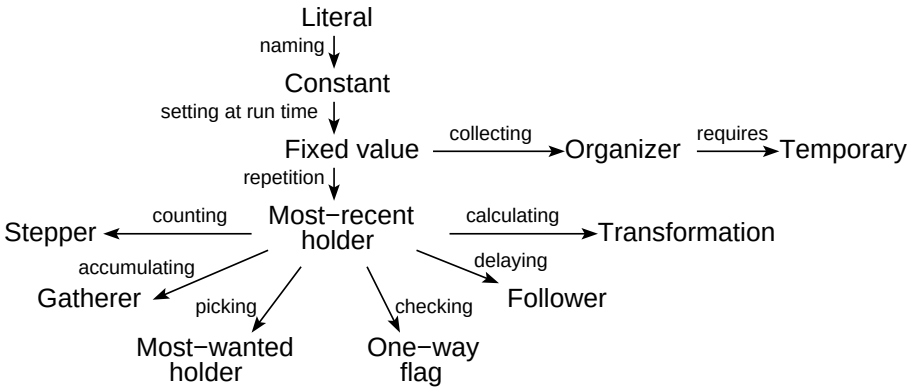

Figure 2.4: Relations between the roles (Kuittinen and Sajaniemi, 2004).

counting the number of some objects), the variable can be described as a stepper. The roles of variables have been used in teaching elementary programming as a part of a classroom experiment. The results supporting the usefulness of the roles of variables as a teaching aid are presented and discussed in Section 2.5. The use of the roles of variables as a teaching aid in elementary programming courses is discussed in more detail by Kuittinen and Sajaniemi (2004).

2.5 Empirical Evaluation of the Role Concept

The roles of variables concept has been evaluated empirically in several studies (Ben-Ari and Sajaniemi, 2004; Sajaniemi and Navarro Prieto, 2005; Sajaniemi and Kuittinen, 2005; Byckling and Sajaniemi, 2005). In this Section, I will shortly describe the main results of these studies.

Ben-Ari and Sajaniemi (2004) studied understandability and acceptability of roles of variables as seen by teachers of procedural programming, and found that teachers learn and internalise the role concept quickly and effortlessly. Similar results have been found by Byckling et al. (2005), who repeated the study with teachers of object-oriented programming, and by Kulikova (2005), who repeated the study with teachers of functional programming.

In another study, Sajaniemi and Navarro Prieto (2005) investigated the roles of variables as a part of expert programmer's tacit knowledge and found evidence suggesting that expert programmers did group variables according to their behaviour, and that the

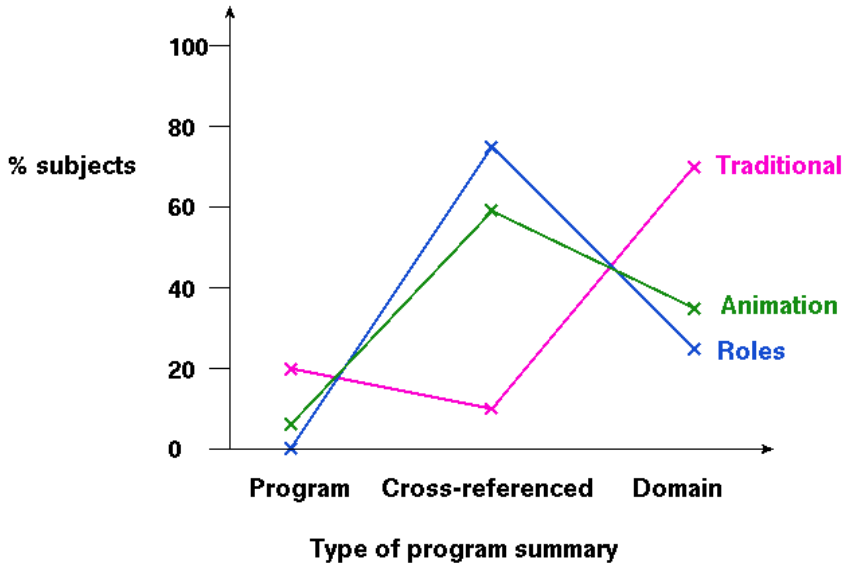

Figure 2.5: Mental representation of programs in three groups (Sajaniemi and Kuittinen, 2005).

roles used in the experimental materials existed in the groupings the participants made. The studies presented above suggest that the roles of variables can be regarded as a part of experts' tacit programming knowledge.

Sajaniemi and Kuittinen (2005) have carried out a classroom experiment in order to analyse the effects of teaching roles of variables on learning of programming. In the experiment, 91 students in an introductory Pascal programming course were divided evenly into three groups. The traditional group was given normal lectures without the introduction of the role concept and had exercises with animation that was not role-based. The roles group was given lectures that introduced the role concept, but had exercises with animation that was not role-based. Finally, the animation group was given lectures that introduced the role concept, and had exercises, in which role-based animation was used.

Based on the classroom experiment, both Sajaniemi and Kuittinen (2005), and Byckling and Sajaniemi (2005) have found results suggesting that visualisation of roles with PlanAni has positive long-term effects on learning programming when compared with textual Turbo Pascal environment. The results of these analyses are summarised in Table 2.3, and in Figures 2.5 and 2.6.

Table 2.3: Number of roles used by pairs of students in a program construction protocol task (Byckling and Sajaniemi, 2005).

Pair	MRH	TRN	MWH	ONE	STP	OTH	Total	Optimal
TRAD-E1	2	1	-	-	-	-	3	no
TRAD-E2	2	2	2	2	-	-	8	no
TRAD-E3	2	1	-	-	1	-	4	no
ROLE-E1	2	1	-	-	-	-	3	no
ROLE-E2	2	1	2	1	-	1	7	no
ROLE-E3	2	2	1	-	-	-	5	no
ANIM-E1	2	1	1	1	-	-	5	yes
ANIM-E2	2	1	1	1	-	-	5	yes
ANIM-E3	2	1	1	1	-	-	5	yes
ANIM-E4	2	1	1	1	-	-	5	yes
OPTIMAL	2	1	1	1	-	-	5	

In their analysis, Sajaniemi and Kuittinen (2005) sorted program summaries into three types depending on the amount of domain versus program statements in object descriptions. Summaries with at least 67% domain statements (indirect and unclear statements excluded) were called *domain-level summaries*, summaries with at least 67% program and program only statements were classified as *program-level summaries*, and all others were called *cross-referenced summaries* because they had a more even distribution of domain and program information. As can be seen from Figure 2.5, the roles and animation groups used significantly more cross-referenced program summaries than the traditional group. According to Pennington (1987a), these kind of program summaries are typical to high comprehension programmers.

Table 2.3 shows that the animation group was able to use the optimal amount of variables when constructing a new program more often than the other two groups, indicating the possession of more programming schemas. Animation group also used forward development in writing new programs more often than the traditional or roles group (Figure 2.6). This can be seen as indication of skilled programming (Rist, 1989).

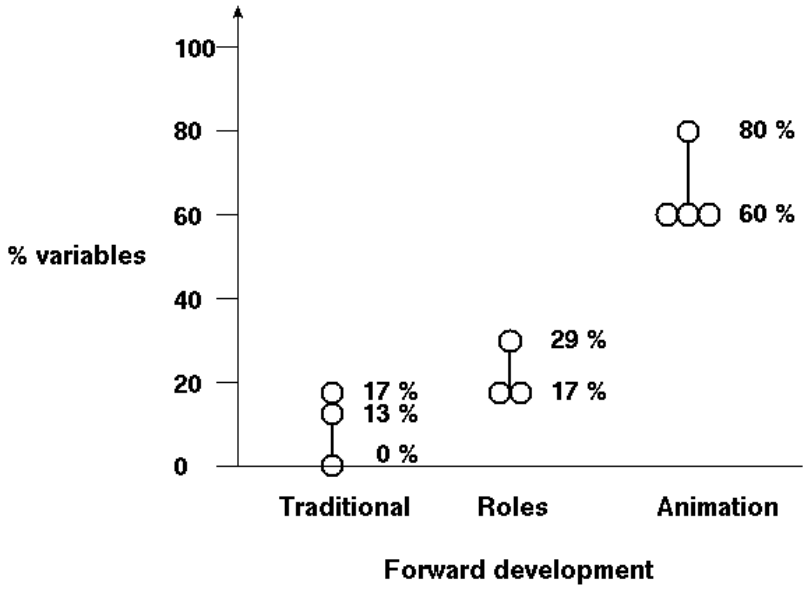

Figure 2.6: Use of forward development in a program construction protocol task in three groups (Byckling and Sajaniemi, 2005).

Chapter 3

Visual Attention and Eye Tracking

In our research, we will use eye tracking for collecting information about the visual attention of participants when they are viewing visualisations. This information can be used to investigate cognitive processes of participants, even though visual attention is not a direct implication of the focus of participants' cognitive activities. In this chapter, I will consider the role of visual attention in studying cognitive processes of programmers, utilisation of eye tracking in measuring visual attention, and the appropriateness of eye tracking as a psychology of programming research method. This chapter has been published in proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004) as a joint paper by Nevalainen and Sajaniemi under the title "Comparison of Three Eye Tracking Devices in Psychology of Programming Research" (Nevalainen and Sajaniemi, 2004).

3.1 Introduction

In the focus of psychology of programming research is the understanding of mental representations and cognitive processes of programmers when they are writing, reading and learning computer programs. The cognitive processes can't be observed directly. Instead, the researcher has to collect secondary data through which the processes can be inferred, for example by making observations of the participant's actions. These observations can consist of for example errors the participant makes, time the participant uses or location of the point of gaze (POG) of the participant.

In eye tracking, the collection of POG data can be performed without the need of the participant performing any action. This can be seen as a benefit when studying cognitive processes that can be easily disturbed. The collected POG data provides information of participant's attention, and it can be used as supportive evidence when studying cognitive processes.

An overview of eye tracking as a research method is given for example in Duchowski (2003) and in Hyönä et al. (2003). Eye tracking has been used in several usability studies (Goldberg and Kotval, 1999; Byrne et al., 1999a; Sibert and Jacob, 2000; Goldberg et al., 2003), studies related to eye-based interactive systems (Jacob, 1990; Majaranta et al., 2004; Ashmore et al., 2005), and cognitive psychology studies related to different search and reading strategies (Rayner, 1992, 1998; Findlay, 1992; Kennedy, 1992). In psychology of programming research, eye tracking has been used by Crosby and Stelovsky (1989), who studied the code viewing strategies of the participants. Bednarik and Tukiainen (2004) have compared eye tracking with blurred display.

Despite the potential usefulness of eye tracking in psychology of programming research, there exists only few instances where eye tracking has actually been used. Therefore experience concerning the benefits, disadvantages and problems of eye tracking in psychology of programming research is needed. In the experiment reported in this chapter, we used three eye tracking devices to record participants' POG when studying short computer programs using PlanAni program animator. We studied the easiness of use and accuracy of the three devices. We also observed and estimated the amount of disturbance the devices caused to the participants.

The rest of the chapter is organized as follows. Next Section gives an introduction to the eye tracking process and to the devices used in this experiment. Then the experiment is described and results are presented and discussed.

3.2 Eye Tracking Methodology

Eye tracking process can be divided roughly into the following steps: participant set-up, adjustments, participant calibration, and monitoring.

In the participant set-up phase, the participant is seated and her location in relation to the eye-tracking device is adjusted. If head mounted optics is used, the eye tracking

device is placed on participant's head and its position is adjusted.

The adjustments phase includes adjusting the settings of the eye tracking program, detecting and ensuring the recognition of the participant's eye(s), and opening the file used for the recording of the eye tracking data.

In the calibration phase, a calibration pattern consisting of a number of calibration points is shown to the participant. The participant is asked to direct her gaze to each of the calibration points and the location of the POG for each calibration point is recorded. The values from the calibration are used in calculating the locations of points of gaze from the values received from the eye tracking device. The calibration phase is repeated until satisfactory calibration values are recorded for each calibration point. One significant problem in eye tracking is the drift effect, which indicates a deterioration of the calibration over time (Tobii 2003). The drift effect can be reduced by ensuring the stability of the light conditions of the environment and the equal light intensity between calibration stimuli and the experiment stimuli.

The monitoring phase consists of viewing the status of the eye tracking and, if necessary, readjusting the settings during the tracking of the actual experiment tasks.

In the experiment we used the following three devices: Tobii 1750 from Tobii Technology, ASL 504 Pan/Tilt Optics from Applied Science Laboratories and ASL 501 Head Mounted Optics from Applied Science Laboratories. All three devices use video based combined pupil and corneal reflection eye tracking.

In Tobii 1750 (Tobii, 2004), the eye tracking device is embedded into the panels of the monitor that the participant is viewing (Figure 3.1). The device uses a wide-angle camera to capture images of the participant and near infrared light emitting diodes for eye illumination. The device uses both eyes of the participant for tracking. Tobii 1750 records data at the rate of 30 Hz (30 gaze data points/second). When the device does not detect the participant's eye(s), the recording rate is slowed down until proper detection is regained. The theoretical accuracy of POG coordinates provided by the device is 1 degree visual angle (approximately 1 cm error when the participant is seated at 50 cm distance from the display).

In ASL 504 Pan/Tilt Optics (ASL, 2003b), the eye tracking device is placed below the monitor the participant is viewing (Figure 3.2). The device has an adjustable wide angle camera that repositions itself according to the movements of the participant. The

Figure 3.1: Tobii 1750.

device uses the wide angle camera to capture an image of the participant's eye and near infra-red light emitting diodes for eye illumination. The device uses one eye for tracking. ASL 504 Pan/Tilt Optics records data at the rate of 50 or 60 Hz. The theoretical accuracy of POG coordinates provided by the device is 0.5 degree visual angle (approximately 0.5 cm error when the participant is seated at 50 cm distance from the display).

In ASL 501 Head Mounted Optics (ASL, 2003a), the optics device is placed on participant's head (Figure 3.3). The device uses one wide angle camera to capture image of the participant's eye and another wide angle camera to capture the participant's field of view (the scene camera). The device uses near infra-red light emitting diodes for eye illumination. The device uses one eye of the participant for tracking. ASL 501 Head

Figure 3.2: ASL 504 Pan/Tilt Optics.

Mounted Optics records data at the rate of 50 Hz. The theoretical accuracy of POG coordinates provided by the device is 0.5 degree visual angle.

3.3 Experiment

In the experiment, we studied the easiness of use of eye tracking devices by measuring the total amount of time needed for the preparations of the participant. The preparations consist of participant set-up, adjustments and calibration. We also observed and estimated the effort these activities required from the participant. The accuracy of the devices was measured by calculating mean distances between recorded points of gaze (in the data files) and requested points of gaze (measured with the eye tracking software). The experimenters were using eye-tracking devices for the first time.

Figure 3.3: ASL 501 Head Mounted Optics.

3.3.1 Method

Design: A within-subject design was used with one independent variable (the eye tracking device used for collecting the data) and two dependent variables (the time needed for the preparation of the participant, and the accuracy of the device).

All participants were measured using all three eye tracking devices (Tobii 1750, ASL 504 Pan/Tilt Optics, and ASL 501 Head Mounted Optics) and the order of the devices was counterbalanced. Each device occurred in each of the chronological position (1st, 2nd or 3rd measuring device) equal number of times. In the experiment we used two

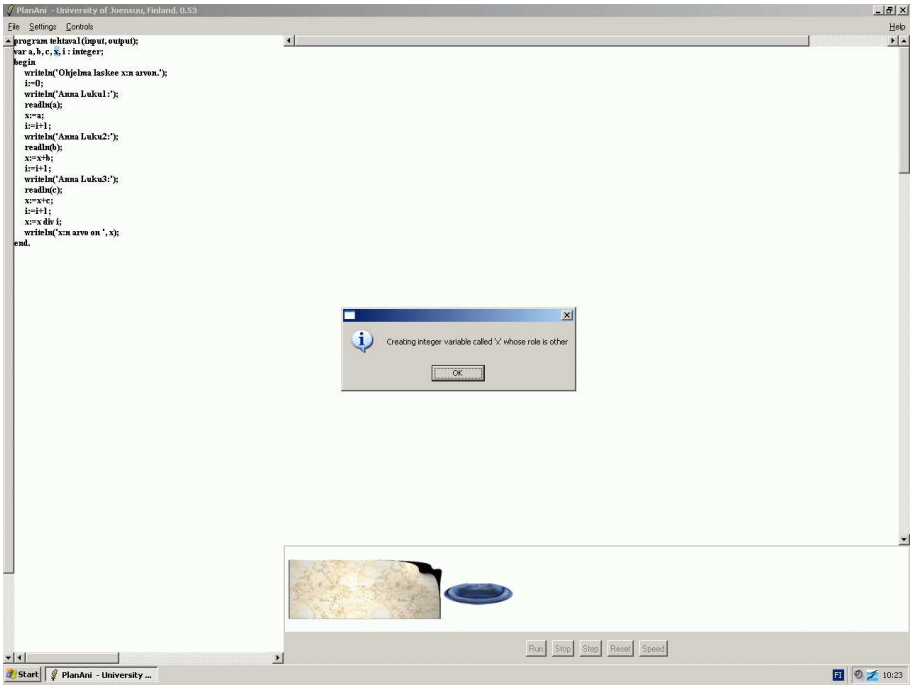

Figure 3.4: PlanAni with code view only.

different versions of PlanAni. The order of the versions was varied so that with each tracking device and each of the viewed programs two of the four participants used the animator with code view first and the other two used the animation view first.

Participants: Twelve participants, eight male and four female, participated in the experiment. The participants were required to have at least basic programming skills and some experience in programming. They were recruited from third year courses in computer science and were given a coffee ticket for their participation.

Materials: For the purpose of the experiment, PlanAni was modified so that it showed either only the code-view that is located on the top left corner of the animator (Figure 3.4) or only variable animation-view that is located on the top center of the animator (Figure 3.5). All variables were depicted by the same neutral image. Both versions showed notifications for the participant and the input/output area. For the task of focusing at specific targets on the screen, screenshots of PlanAni were used. The PlanAni version was v0.53.

Procedure: The participants used PlanAni to comprehend six short computer programs—two programs with each eye tracking device. They were allowed to view each program one time step by step. The POG of participants during these tasks was

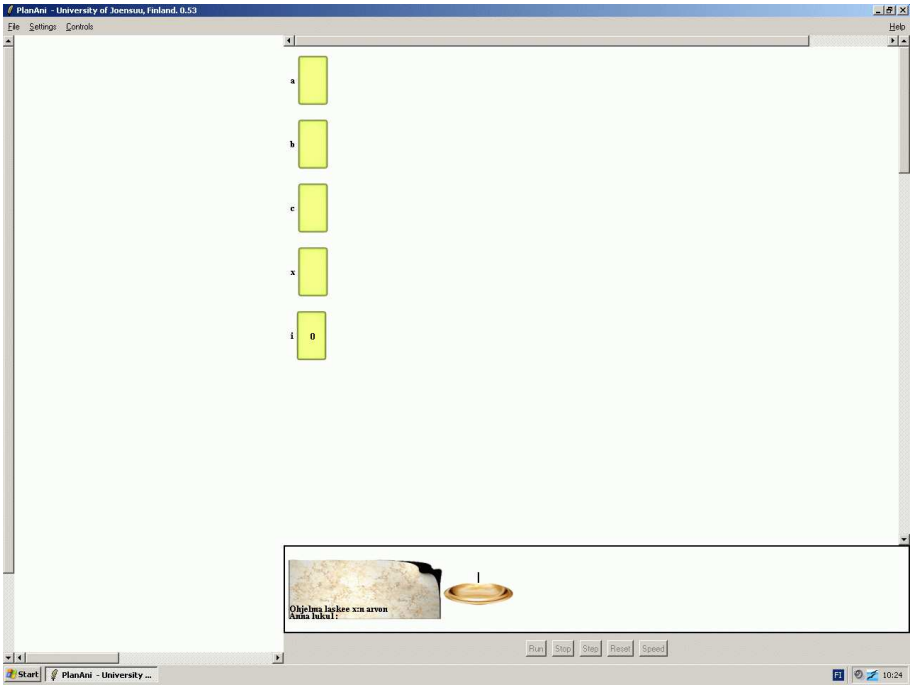

Figure 3.5: PlanAni with animation view only.

measured. With each device, the participant was first seated and the eye tracking device's location in relation to the participant was adjusted (participant set-up phase). The movement of the participant was minimized by using a chair without wheels, by setting the chair close enough to the desk to minimize the horizontal rotation and advising the participant to avoid quick and wide movements of her head. The participant was not explicitly demanded to stay perfectly immobile during the task. After set-up, the settings of the interface program were adjusted, detection of the participant's eyes was performed and the file used for storing the POG data was opened (adjustments phase). Then the calibration of the participant was performed (calibration phase). Time needed for these preparations was measured by the experimenter using a special program that required a single key-press to start and stop time measuring. With each device, the participant performed two program comprehension tasks so that she used both versions of PlanAni. After each viewing task, the participants were asked to give a short program summary. The program summaries were collected for the purpose of motivating the participants to study the program but they were not analyzed further in this experiment. After studying the programs, participants were asked to look at eight specific targets on the screen before proceeding to the next eye tracking device.

Table 3.1: Times (means in seconds and standard deviations) needed for the preparation and execution phases of the tasks.

Phase	Device					
	Tobii 1750		ASL 504		ASL 501	
	Mean	SD	Mean	SD	Mean	SD
Preparation	471.8	128.9	548.3	126.8	953.5	164.4
Execution	502.6	112.1	525.0	122.9	476.6	68.4

Table 3.2: Amount of valid, uncertain and invalid data from all collected gaze data and the percentage of invalid data.

Amount of	Device		
	Tobii 1750	ASL 504	ASL 501
Valid data	104101	182177	198812
Uncertain data	16486	-	-
Invalid data	10629	17722	6099
Percentage of invalid data	8.1	8.7	3.0

3.3.2 Results

Table 3.1 gives the mean times (in seconds) needed for the preparation phase and the execution phase of the program comprehension tasks. The preparation time is measured from the beginning of the set-up to the end of calibration. The difference in preparation times between Tobii 1750 and ASL 501 (paired t -test, $t = 8.187$, $df = 11$, $p < 0.0001$), and the difference between Tobii ASL 504 and ASL 501 (paired t -test, $t = 6.417$, $df = 11$, $p < 0.0001$) are both statistically significant.

Table 3.2 gives the amounts of valid, uncertain and invalid data as reported by the devices, and the percentage of invalid data. Tobii 1750 provides validity codes 0-4 (0 = valid, 1-3 = uncertain and 4 invalid) for the data. For the ASL devices, the validity of the data is determined by value in the pupil size field (0 = invalid, otherwise valid).

Table 3.3 gives the mean distances (in centimeters) of measured points of gaze from the requested points of gaze, and the corresponding visual angle when the participant is seated at 50 centimeters distance from the display. The distance was measured for points within a threshold of 2.5 cm from the center point of the target. The thresh-

Table 3.3: Distances (means in centimeters and standard deviations) of measured points of gaze from the requested points of gaze, and the corresponding degrees of visual angle.

	Device					
	Tobii 1750		ASL 504		ASL 501	
	Mean	SD	Mean	SD	Mean	SD
Distance	1.134	0.203	1.391	0.351	1.609	0.314
Visual Angle	1.3		1.6		1.8	

old was selected so that the theoretical accuracies of the devices and the microscopic movements of the eye fitted within the threshold. ASL 501 provides the POG coordinates on a plane that is in relation to the field of view of the participant, while the other two devices provide the coordinates on a fixed plane. With ASL 501, the location of the screen in the field of view shifts when the participant turns her head. This shift was visually detected and measured, and the corresponding corrections were calculated and applied to the coordinates before calculating the distances. The difference between Tobii 1750 and ASL 501 (paired t -test, $t = 3.707$, $df = 8$, $p = 0.006$) is statistically significant.

3.4 Discussion

The time needed for preparation when using 501 was approximately twice as long as the time needed for Tobii 1750 or ASL 504 (see Table 3.1). In our experience, there are two main reasons that explain this difference. Firstly, the participant set-up phase with ASL 501 consisted of more steps and required more effort than with Tobii 1750 or ASL 504. One time consuming step was locating the image of the participant's eye through the visor so that it was in correct angle and the visor was not in front of the participant's field of view. Secondly, the calibration with the ASL 501 was more troublesome and needed to be repeated more often than with Tobii 1750 or ASL 504, mainly because the participants were required to keep their heads perfectly still during the calibration phase. One possibility to make the calibration easier and faster with the ASL 501 is to use a bite bar or chin rest during calibration. This may, however, cause discomfort to the participant and its applicability in a psychology of programming experiment is

questionable.

Table 3.2 shows the amounts of collected valid, uncertain and invalid data. All the devices reported invalid data under 10%. The difference between ASL 501 compared to Tobii 1750 and ASL 504 occurred most probably because the two last mentioned devices lost the eye easily when the participant used the keyboard to provide input to the program and the eye moved out of the reach of the devices' cameras. ASL 504 also had difficulties in automatically relocating the eye and in some cases it needed to be aided by relocating the eye manually.

Table 3.3 shows the accuracies of the three devices. The values indicate that Tobii 1750 has the highest accuracy, ASL 504 provides second highest accuracy, and ASL 501 the lowest accuracy. Only the difference between Tobii 1750 and ASL 501 is, however, statistically significant. One factor in the low accuracy of the ASL 501 is probably an inaccuracy in the visually estimated correction due to head movements. The need for this correction can be removed by using magnetic head tracker with ASL 501. Tobii 1750's measured accuracy is quite near to the theoretical accuracy given in Section 3.2. The measured accuracy is 1 - 1.6 degrees for a participant sitting at 40 - 60 centimeters distance from the screen. ASL 504 and ASL 501 fall clearly behind the theoretical accuracy given in Section 3.2. For ASL 504 the measured accuracy is 1.3 - 2 degrees for a participant sitting at 40 - 60 centimeters distance from the screen. For ASL 501 the measured accuracy is 1.5 - 2.25 degrees for a participant sitting at 40 - 0 centimeters distance from the screen

3.5 Eye Tracking in Psychology of Programming Research

In psychology of programming research eye tracking can be used as an implication of the focus of participants' attention. The POG is not, however, the same as the focus of attention, because attention is not necessarily always associated with the visual scene, even though POG is. The participant can also voluntarily target his attention slightly off the POG (Posner et al., 1980).

The general unobtrusiveness of an eye tracking device can be seen as a factor when using this technology in psychology of programming research. Participants' cognitive

processes can be easily disturbed with objects in the field of view, sounds in the room, and extra activities required by the experimental settings. Some of the participants commented that the scene camera of the ASL 501, positioned according to the manual, was disturbingly in their field of view. The visor of the ASL 501 remained in the lower part of the participant's field of view during the measuring. This did not, however, invoke any comments from the participants. With ASL 504, the adjustable camera produced a buzzing sound when it repositioned itself, causing the participant to be aware of the device's existence. Tobii 1750 looks like a normal display device and makes no visible or audible interference.

When considering the required effort and caused disturbance, Tobii 1750 seemed to be the most unobtrusive for the participants. With ASL 504, the participant was required to keep her head perfectly still during the detection of the eye, since the auto-follow property of the camera could be turned on only after the pupil and corneal reflection were found. The positioning of the optics device of ASL 501 on the participant's head was time consuming and caused physical discomfort to the participant.

Tobii 1750 enabled a participant to easily observe the tracking status before the calibration phase, and to take part in the detection of the eye. The calibration was not dictated by the operator but the tracking program performed the calibration by showing the participant calibration points in random locations at a slow pace.

In eye tracking, the quality and amount of recorded data is influenced by the amount of participants' motions. The more immobile the participant is, the better data eye tracking devices usually record (Tobii 2003, ASL 2003a). When eye tracking is used in psychology of programming research, however, the immobilising of the participant can disturb the cognitive processes that are being studied. It seems that there is a trade off between the accuracy and the ecological validity of data. With the participant seating used in our experiment, we reached an accuracy that was quite near to the theoretical accuracy of Tobii 1750. With ASL devices, however, the measured accuracy was considerably behind the theoretical values.

Tobii 1750 and ASL 504 require the participant to be seated and tolerate limited movements of the head, only. ASL 501 allows the participant to move around — an activity needed in some experimental settings in psychology of programming.

Chapter 4

Mental Models and Program Summary Analysis

Mental models of novice programmers will be evaluated in our research with the help of program summaries, which will be analysed using Good's program summary analysis scheme (Good, 1999). Since the replicability of the Good's scheme has not been analysed earlier, we carried out an inter-rater reliability analysis on the scheme. In this chapter, I will first consider shortly mental models of programmers, and program summaries as a means for studying them. Then I will describe the investigation concerning the inter-rater reliability of Good's scheme. This chapter has been published in proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004) as a joint paper by Byckling, Kuittinen, Nevalainen, and Sajaniemi under the title "An Inter-Rater Reliability Analysis of Good's Program Summary Analysis Scheme" (Byckling et al., 2004).

4.1 Introduction

In research into the psychology of programming and computer science education one central question is the problem of measuring the quality of learning and comprehension of programming concepts. There is no universally agreed-upon measure of comprehension (Dillon and Gabbard, 1998) but the following methods have been applied: written or verbal tests on recall, recognition (e.g., multiple-choice questions), and relationships

between concepts; problem solving exercises (i.e., programming tasks); essays evaluated by their correctness or by qualitatively analyzing students' mental models; and interviews. In the context of programming, essays may also be program summaries, i.e., free-form explanations of programs.

Program summary analysis has been used in the psychology of programming and computer science education to characterize mental models of novice (Corritore and Wiedenbeck, 1991) and expert (Pennington, 1987a) programmers attaining high levels of comprehension; to characterize mental models of students capable of reusing program code (Hoadley et al., 1996); to describe how mental models depend on underlying programming paradigm (Good, 1999) or task type of the programmer (O'Shea and Exton, 2004); and to evaluate learning outcome in novice programmers (Hughes and Buckley, 2004; Sajaniemi and Kuittinen, 2005). These studies suggest that program summary analysis can be utilized in revealing novice and expert programmers' mental models and that the contents of the mental models can be used to characterize the quality of comprehension.

The basic idea of program summary analysis is to ask participants to provide a free-form explanation, or summary, of a program just studied. By omitting detailed instructions about the form of the summary, participants' own preferences guide the selection of information in the summary and a wide variation in the responses is usually achieved. The program summary methodology avoids the problems of false positive results often associated with binary choice questions, and the difficulties in designing sensitive and reliable multiple choice questions (Good and Brna, 2004). In program summary analysis, the interest is not in the correctness of the summary; the abstraction level and the types of information are more important characterizations of the mental model than a memorization of the program code details.

In order to analyze program summaries, some analysis scheme must be used. Earlier studies used Pennington's scheme (Pennington, 1987a) which is based on information types and levels of detail. Later, Good (1999) devised another scheme based also on information types, but more finely-grained and fully specified than in Pennington's scheme; and object descriptions, which is a restricted version of Pennington's level of detail. Hughes and Buckley (2004) have extended Good's scheme to cover information types and detail levels needed in summaries of concurrent programs; Burkhardt et al. (1997) have extended Pennington's scheme to cover new information types in object-oriented programming but they are using the scheme for question classification

instead of program summary analysis. Hoadley et al. (1996) used a simple classification based on abstraction level similar to Good's object descriptions but consisting of fewer categories. von Mayrhauser and Lang (1999) and O'Brien et al. (2001) have developed schemes for coding program comprehension protocols but these schemes cannot be used to analyze program summaries; Hughes and Buckley (2004) have extended Good's scheme to cover program comprehension protocols.

Good's scheme has been used in some studies (Good, 1999; Good and Brna, 2004; O'Shea and Exton, 2004; Sajaniemi and Kuittinen, 2005) but its replicability has not been analyzed (Good and Brna, 2004), e.g., there is no knowledge of its inter-rater reliability nor of reasons for possible differences among raters. In this chapter we report an investigation where three raters used Good's scheme to analyze real program summary data. The problems in using the scheme as well as differences between the raters were recorded and analyzed.

The rest of the chapter is structured as follows. Section 2 gives an introduction to Good's scheme. Section 3 describes the investigation and its results are discussed in Section 4.

4.2 Good's Scheme

Good's program summary analysis scheme has been described in two documents. Good's PhD dissertation (Good, 1999) gives detailed instructions including Coding Manuals and coding examples whereas an article by Good and Brna (Good and Brna, 2004) describes various categories of the scheme with few examples. We will now give a short summary of Good's scheme; see the above references for more exact definitions.

Good's program summary analysis scheme consists of two independent classifications of program summary segments. The first is based on *information types* (IT), i.e., what kind of information about the program a statement reveals. The other classification is based on *object description categories* (ODC) that look at the way individual objects are described in summaries. The interesting items are different in these two classifications and consequently program summaries are segmented differently for these two purposes.

For the purposes of IT coding, program summaries are segmented to short passages consisting of a subject and a predicate (either of which may be implied). The classification comprises eleven categories:

- **Function (FUN):** The overall aim of the program, described succinctly.
- **Actions (ACT):** Events occurring in the program described at a lower level than *Function*.
- **Operations (OPE):** Small-scale events which occur in the program, such as tests, assignments etc.
- **State-high (SHI):** Describes the current state of a program when a condition has been met (and upon which an action is dependent).
- **State-low (SLO):** A lower-level version of *State-high*. *State-high* describes an event at a more abstract level than *State-low* which usually describes the direct result of a test on a single data object.
- **Data (DAT):** Inputs and outputs to programs, data flow through programs, and descriptions of objects and data states.
- **Control (CON):** Information having to do with control structures and with sequencing.
- **Elaborate (ELA):** Further information about a process, event or data object which has already been described. This also includes examples.
- **Meta (MET):** Statements about the participant's own reasoning process.
- **Unclear (UNC):** Statements which cannot be coded because their meaning is ambiguous or uninterpretable.
- **Incomplete (INC):** Statements which cannot be coded because they are incomplete.

IT categories are related in terms of level of granularity. *Function* describes the highest level of abstraction, i.e., the purpose of the program which can be described with no reference to how it is achieved; *Actions* and *State-high* describe in an abstract manner

This program checks a basketball players height from the list given.	Actions
If the height of the player is over 180	State-high
then he is selected for the team.	Function
Once there are five players	State-high
the program is terminated.	Control

Figure 4.1: An example of IT coding (Good, 1999).

the way that the program works; *Operations* and *State-low* correspond to single lines of code. Figure 4.1 gives an example of IT coding.

For the purposes of the ODC coding, the interesting objects must first be selected and only then the program summaries can be segmented. ODC coding is applied to data objects only, and program summaries are segmented so that there is exactly one data object per segment. The classification comprises seven categories:

- **Program only (PON):** References to items which can occur only in the program domain.
- **Program (PRO):** References to objects, which could be described at various levels, described in program terms.
- **Program—real-world (PRR):** Object descriptions using terminology which is valid in both real-world and program domains, and is abstract and shared across various problem domains.
- **Program—domain (PRD):** Object descriptions containing a mixture of program and problem domain references, or a reference which is equally valid in the program and problem domains.
- **Domain (DOM):** References to objects described in problem domain terms.
- **Indirect reference (IND):** An anaphoric reference to an object.
- **Unclear (UNO):** Object references that cannot be coded because they are ambiguous or unclear, or because the object which is being referred to cannot be identified.

Figure 4.2 gives an example of ODC coding.

The program wants **all marks over 65** listed	Domain
and **all marks over 66** will pass	Domain
the exam.	Domain
The output will state	Program
the mark	Domain
and whether **the person** has passed.	Domain

Figure 4.2: An example of ODC coding (Good, 1999). The coded object descriptions are marked in **boldface**.

4.3 Investigation

This Section describes an investigation into inter-rater reliability of Good's scheme and the main reasons for differences among raters.

4.3.1 Method

Three raters learned to use Good's program summary analysis scheme and coded real program summary data. Problems in learning and using the scheme as well as differences between the raters were recorded and analyzed.

Raters: There were three raters, two male and one female, out of whom two were postgraduate students and one was a postdoc researcher. All raters had majored in computer science.

Materials: Forty-four program summaries were gathered as part of another study (Sajaniemi and Kuittinen, 2005). They consisted of students' answers to a program comprehension question in an exam at an university level introductory Pascal programming course. Students' task was to "describe what is the purpose of the given program and how it works", i.e., to write a program summary. Figure 4.3 gives an English translation of the program; in the original program both output strings and variable names were Finnish words or abbreviations. All other variable names were single-letter meaningful abbreviations except the variable `weight` which was a problem domain word (`paino` in the Finnish version). In the English version, the variable `x` should be `d` for "day" but this would clash with `d` for "dose"; in the Finnish version this problem did

```
program task4 (input, output);
var weight,x,d,t: integer;
begin
    t := 0;
    write('Enter patient''s weight (kg): '); readln(weight);
    for x := 1 to 3 do
        begin
        d := weight * 3;
        writeln('Day ', x, 'morning and evening ', d, 'ml.');
        t := t + 2*d
        end;
    for x := 4 to 7 do
        begin
        d := weight * 4;
        writeln('Day ', x, 'morning ', d, 'ml.');
        t := t + d
        end;
    writeln(t)
end.
```

Figure 4.3: English translation of the program summarized by students.

not arise.

Eight program summaries were selected for training the raters and the remaining 36 summaries were used for the inter-rater validation. Table 4.1 gives statistics of the program summary sizes measured in words. (Due to the absence of articles and prepositions, and to the use of compound words[1] in Finnish, the same information contents is usually achieved with less words in Finnish than in English.) The training material was selected to include a diverse set of summary statements; therefore, they tended to

[1]In Finnish, words can be combined in the same style as "textbook" or "database" in English. However, in Finnish this is much more common than in English, e.g., "annosteluohje" for "dosage instructions". Because raters are not willing to split a single word into two separate segments, we will use the notation "dosage-instructions" when compound words appear in our examples. As a compound word refers usually to a single object, this will probably not introduce language-dependent differences.

Table 4.1: Descriptive statistics of the program summaries used in the training and validation phases. Lengths are measured as word counts.

	Training	Validation
Number of program summaries	8	36
Minimum length	114	42
Maximum length	349	285
Mean length	199.0	137.6
Std dev of length	74.6	57.5

be longer than summaries on average. Moreover, the longest summary was included in the training set because of its anomalous nature: it consisted mainly of program code fragments. Excluding this summary, the longest program summary in the training set was 276 words long.

Procedure: There was a training period for segmenting and coding; first for IT and then for ODC. The training started by a meeting where the classification instructions were explained to the raters. The raters then segmented the training material individually on their own time. The resulted segments were compared in a meeting, differences were discussed and segmenting rules for the validation phase were agreed. The training phase continued with the raters coding individually the training material using the agreed segments. The codings were compared and discussed in a meeting and coding rules for the validation phase were agreed. The training phase was followed by the validation phase where raters segmented and coded the validation material independently on their own time.

The training and validation phases for ODC were similar to the IT case described above. In addition, in the training phase the set of objects to be used for segmenting was agreed, also. During the whole investigation, all meetings were lead by a fourth person.

Problems were gathered by the raters, who made notes during training and validation, and by the meeting leader during meetings. Sources of problems and differences in the numeric results were discussed in several meetings after the validation phase.

4.3.2 Results

In addition to the IT categories in Good's scheme, we used the category **Continuation (CUT)** to cover cases where one segment is embedded in another segment. For example, in Figure 4.4 the information (lines 1 and 3) that the variable `t` is increased by a certain amount is interrupted by another information describing the variable (line 2). Hence, only the first line is given a proper IT category, and the third line is coded as a continuation to the first line.

The variable `t`,	Operations
which gives the total drug amount,	Data
is increased by the morning and evening doses.	Continuation

Figure 4.4: An information segment embedded in another segment. The latter part of the surrounding segment is coded as *Continuation*.

Table 4.2 gives the distribution of IT categories by each rater. The first part of the table gives mean percentages of the 36 program summaries for each IT category. The second part is obtained by grouping categories describing high-level program information, low-level program information, and other IT categories. Finally, the last row gives the proportion of high-level information when other, program-unrelated information is discarded.

For each category, the probability of linear correlation of this magnitude to occur by change was calculated for each pair of raters. Categories for which the probability is at least 0.001 for any two raters are marked with † and probabilities being at least 0.05 with ††. For the grouped categories all probabilities are smaller than 0.0001.

The number of IT segments for each rater varied between 883 and 892. There were 811 segments that were common to all three raters, i.e., 8.6% disagreement in segmenting. Out of these 811 common segments the three raters coded 65.5% similarly and 94.0% of the segments were given the same category by at least two raters. Looking at the coding of each of the raters separately, each rater agreed with at least one of the other two raters in from 84.1 to 85.2% of the common segments.

Table 4.3 gives the distribution of ODC categories by each rater calculated as means of the percentages of the 36 program summaries. Probabilities of linear correlation are

Table 4.2: Percentages of IT categories by rater. HIG refers to high-level program information, LOW to low-level program information, and OTH to program-unrelated information. HIP is the proportion of high-level segments of all program-related segments. †: probability of linear correlation of this magnitude between any two raters ≥ 0.001; ††: probability ≥ 0.05.

		Rater					
		A		B		C	
Code	Information Type	Mean	S.D.	Mean	S.D.	Mean	S.D.
FUN ††	Function	9.3	7.6	8.1	5.5	10.5	8.5
ACT ††	Actions	20.7	12.0	18.2	12.4	14.6	12.1
OPE †	Operations	10.6	9.9	7.3	8.4	7.7	6.6
SHI	State-high	1.5	3.4	1.5	3.1	1.2	2.7
SLO	State-low	0.1	0.7	0.1	0.8	0.1	0.4
DAT	Data	24.5	11.1	27.4	12.2	29.9	15.4
CON	Control	4.9	5.3	3.8	4.8	4.8	5.3
ELA	Elaborate	21.6	14.9	26.3	17.1	20.4	16.6
MET †	Meta	1.5	2.7	1.4	2.8	2.7	5.7
UNC ††	Unclear	0.0	0.0	0.8	3.4	0.8	3.0
INC ††	Incomplete	1.0	4.8	1.1	1.9	2.3	4.9
CUT	Continuation	4.2	4.1	3.9	4.5	5.0	4.9
HIG	FUN+ACT+SHI+DAT	56.0	18.2	55.2	20.6	56.2	20.2
LOW	OPE+SLO+CON	15.6	12.2	11.3	11.5	12.6	10.5
OTH	100-HIG-LOW	28.4	16.4	33.4	18.0	31.2	18.7
HIP	HIG / (HIG+LOW) * 100	78.1	15.4	82.4	16.9	80.9	13.8

marked as in Table 4.2. Probabilities for unmarked categories are smaller than 0.0001.

The number of ODC segments for each rater varied between 712 and 734. There were 570 object descriptions that were common to all three raters, i.e., 21.0% disagreement in segmenting. Out of these 570 common segments the three raters coded 73.2% similarly and 99.3% of the objects were given the same category by at least two raters. Looking at the coding of each of the raters separately, each rater agreed with at least one of the other two raters in from 81.1 to 98.8 % of the common segments.

Table 4.3: Percentages of ODC categories by rater. †: probability of linear correlation of this magnitude between any two raters ≥ 0.001; ††: probability ≥ 0.05.

		Rater					
		A		B		C	
Code	Object Description Category	Mean	S.D.	Mean	S.D.	Mean	S.D.
PON	Program only	3.0	6.1	5.8	8.2	4.4	7.9
PRO	Program	15.8	15.1	22.5	16.3	20.5	16.6
PRR †	Program—real-world	1.6	3.2	3.1	4.7	2.4	4.0
PRD ††	Program—domain	7.1	8.1	16.9	9.9	6.1	7.6
DOM	Domain	63.1	16.8	49.6	19.6	59.5	17.8
IND †	Indirect reference	6.6	6.0	2.1	3.7	7.1	7.7
UNO ††	Unclear	2.9	6.5	0.1	0.8	0.0	0.0

4.3.3 Differences in Information Types

This Section first lists reasons for inter-rater IT differences obtained by recording the problems experienced by the raters during the training, and by analyzing the differences observed in the validation. We start with problems in segmenting, i.e., what constitutes a segment, and continue with problems in selecting IT categories for segments. We conclude this Section by looking at the frequencies of the problems in inter-rater differences.

Problems in Segmenting

The basic rule of segmenting program summaries to "short phrases consisting of a subject and a predicate (either of which may be implied)" (Good, 1999, p. 313) appears to be clear and unambiguous. However, the examples in (Good, 1999) and (Good and Brna, 2004) (which will be together called as *scheme defining documents* or SDD) contain exceptions to this rule, e.g., "by adding the adjustacent ones together" suggesting that non-finite clauses can be segments, and '[and then joins it to the other value it would have created if it had done what I just said] (complicated)" suggesting that even a single adjective, i.e., "complicated", referring to a different context than the rest of the sentence can be a segment. We found many occasions where the basic rule would

lead to segments containing several information types; these will be described in detail below.

One might be tempted to suggest that several segments should be used only if the passage contains several IT categories. However, segmenting is supposed to be done first—independently of IT coding— which means that segmenting cannot depend on the number of IT categories present in the passage. Hence, the question of segmenting must be determined by the presence of *information items* independently of their types.

The "and" problem: The SDD contains contradicting examples of the effect of the word "and" on segmenting. The passage

> the program is selecting ... and allowing them to ...

is coded as a single segment (Good and Brna, 2004) whereas

> It takes the numbers in the list and adds up numbers next to each other

is coded as two segments (Good, 1999, p. 320). However, in both examples, there is a predicate on both sides, and the subject is implied on the right-hand side. In our summaries, we found cases where both the subject and the predicate are implied:

> program tells how much and how many times a day

Contrasted with a slightly longer version:

> program tells how much and it also tells how many times a day

it becomes unclear whether one should use one or two segments.

The non-finite clause problem: Non-finite clauses and other similar linguistic constructs give rise to passages that contain several information items even though there is no new subject or predicate:

> once adjusted the numbers are added
> which tells **using the value of x** the number of the day

Special cases of this problem are specifications of time or other condition where the wording may contain a subject and a predicate or they may be absent:

> When we are at the beginning of the for-loop
> *vs.*
> At the beginning of the for-loop
>
> If the input is 0 [...] *vs.* With input 0 [...]

Time specifications can be even shorter and still carry an information item, e.g., “This time” or “Then”, making it problematic to decide when a time specification should have its own segment.

Another special case of this problem is formed by *Meta* passages that may consist of a single word:

> presumably
> probably
> clearly

The information–dense passage problem: Some passages of text contain several information items and, indeed, several IT categories even though there is a single subject and predicate. For example, the following passage describes both data (the meaning of the variable `t`) and control (the assignment):

> The total drug amount is stored into the variable `t` at every round.

The example problem: Examples (that should be coded as *Elaborate*) may be very short or consist of tens of lines. The SDD notes that “(65 in this case)” is a single segment (Good and Brna, 2004) but it is unclear how many segments should be made out of a long example. Furthermore, examples rarely have a subject or predicate, e.g.,

> (kg)
> i.e., single-dose,

Problems in Selecting IT Category

Coding problems considered either discrimination between two (or even three) IT categories, or were more general and not connected to any specific categories. Moreover, we found a need for two new IT categories. We start with the latter problem types.

The atmosphere problem: The SDD suggests (Good, 1999, p. 314) that coding can be carried out by category, i.e., by several passes through the summary in order to identify all segments of a particular type. In this style of coding, each segment is coded almost independently of the surrounding segments. However, a certain passage, e.g.,

> Finally the amount of agent needed for doses is output.

may have a very different interpretation depending of its context. If this segment appears at the beginning of a summary where the function of the program is described, it represents *Function* information. If it appears at the end of a summary as an explanation of the final output statement, it represents *Data*. Thus overlooking the context, or atmosphere, of the passage leads to incorrect coding.

The not-done problem: Program summaries contain passages that describe what the program does *not* do. The SDD does not explain how to code these. Examples are:

> input cannot contain letters
> and accepts even a negative value [i.e., there is no input check]

The missing "continuation" category problem: We found the need for a new IT category *Continuation* described at the beginning of this Section.

The missing "irrelevant" category problem: We found the need for a new IT category *Irrelevant* to code information that is not related to the activity or results of the program, e.g.,

> else the program works.
> The name of the program is `task4`.
> The program starts with the word "`begin`".
> The program contains two `for`-loops.

Irrelevant information is usually valid—it just is not related to activities of the program or to its results.

The Operations vs. Actions problem: Activity within a loop that is described based on a single round (i.e., it should be coded as *Operations*) may appear to the reader as describing the effect of all rounds (i.e., an activity at the *Actions* level). The reader may even change the way the segment appears to him or her. Examples of this type are:

> the variable `t` is increased again
> The total drug amount is stored into the variable `t` at every round

The Data vs. Operations/Actions problem: There are many occasions where an information item may be interpreted to describe a variable (i.e., *Data*) or an activity (i.e., *Operations* or *Actions*). First, the specific words used to refer to an object may make a difference:

Finally the variable `t` is increased again	Operations/Actions ?
Finally the dose-total `t` is increased again	Data ?
Finally the gatherer `t` is increased again	Data ?

Even though these examples are very similar, the use of a natural language concept ("dose-total") or role name ("gatherer" [2]) seems to change the information from operation to data flow.

Second, individual verbs seem to be attached to different levels of abstraction even though the information content is basically the same, e.g.,

the size of the dose is assigned to the variable `d`	Operations
the size of the dose is computed into the variable `d`	Operations/Data ?
the variable `d` holds the size of the dose	Data

Third, it is not clear how to code an abstract activity (which should be coded as *Actions*) that refers at the same time to the total life-cycle of a variable (which should

[2]Gatherer is one of the roles taught to the students in the original study. It can be compared to the concept of counter. While a counter counts something, a gatherer gathers the net effect of something, e.g., the sum of individual values.

be coded as *Data*). A reference to the total life-cycle of a variable may even be indirect ("This time"), e.g.,

For the fourth day the dose changes	Actions/Data ?
The dose is determined by	Operations
This time the dose is determined by	Operations/Data ?

The Data vs. Control problem: Passages concerning control variables contain information about both control and the life-cycle of a variable:

where `x` steps from the value 1 to the value 3	Data/Control ?

The State-low vs. State-high problem: According to the SDD *State-high* relates to state described at an abstract level and *State-low* to state described at a low level (Good, 1999, p. 315). However, in the SDD examples *State-high* seems to be connected to loop termination conditions and *State-low* to `if`-conditions, independently of the level of the description, e.g. (Good, 1999, p. 196, p. 321),

[program should terminate] when counter is greater than 4	State-high
if the head is greater than 180 ...	State-low

The Elaborate vs. some other category problem: Passages coded as *Elaborate* are examples or restatements of facts that have already been described. It is, however, unclear how far away the original fact is allowed to be, i.e., may the elaboration follow, say, 10 segments after the original fact, and how much can be deduced from earlier facts, i.e., may an elaboration contain a fact that is not explicitly said before but that can be easily deduced from earlier segments.

Furthermore, it is unclear whether descriptions of manner should always be coded as *Elaborate* as suggested by the examples in the SDD, e.g., (Good, 1999, p. 320):

It adds two successive numbers in the list	Function
putting a zero at the start	Elaborate

Rater Differences

The above list of problems in IT segmenting and coding was collected during the training and validation phases of the investigation. We devised *ad-hoc* solutions to problems that were found during training and the raters were advised to work accordingly. Nevertheless, there were differences between raters in the validation as described in Section 4.3.2.

Two problems accounted for more than half of the *differences in segmenting*: the *Meta* passages form of the Non-finite Clause Problem covered 27.7% and the "And" Problem covered 23.8% of the 231 non-common segments. Another 11.7% of the non-common segments were caused by segments coded as *Continuation* by some rater(s) but not considered to cause segments by other(s). However, these are side effects of other differences in segmenting and do not explain anything by themselves.

Most interesting *differences in coding* the 811 common segments regard the 49 segments coded differently by all raters. Two cases accounted for a majority of these differences. The first was the Missing *Irrelevant* Category Problem that covered 30.6% of the segment. The second was a combination of the *Operations* vs. *Actions* Problem and the *Data* vs. *Operations/Actions* Problem that covered 32.7% of the segments. These segments were of the form

> The total drug amount is stored into the variable `t` at every round

where a single assignment is *Operations*, the total effect of these assignments during all rounds of the loop is *Actions*, and the information that the variable `t` holds the total amount is *Data*.

Table 4.4 gives frequencies of information types in the 231 segments agreed by two raters but not by the third one. Taking the overall frequencies of the categories (Table 4.2) into account, the categories *Actions, Operations,* and *Incomplete* occur as codes for the problematic segments more often than expected, and the categories *Function*, *Data*, and *Continuation* occur less often than expected.

Table 4.4: Frequencies of information types in segments agreed by two raters only.

Information Type	Frequency
Function	7.6
Actions	25.3
Operations	12.4
State-high	0.0
State-low	0.0
Data	20.8
Control	4.6
Elaborate	19.5
Meta	1.5
Unclear	1.3
Incomplete	5.7
Continuation	1.2

The most common problems were the *Data* vs. *Operations/Actions* Problem (22.5%), the *Elaborate* vs. Some Other Category Problem (19.9%), and the *Operations* vs. *Actions* Problem (11.7%).

4.3.4 Differences in Object Description Categories

We now turn to ODC inter-rater differences. We will first look at the problems in identifying objects, and then continue with problems in segmenting and coding followed by an analysis of differences observed in the validation.

Problems in Recognizing Objects

The data object definition problem: According to the SDD (Good, 1999, p. 316) ODC coding is applied to data objects; any other objects (e.g., the program, actions/events within the program, such as recursive call, iteration) should not be used for segmenting. During the training phase we looked at the differences between the raters and realized that the raters did not, however, agree on what objects should be used for segmenting. As a consequence, we decided that coding should be based on the following objects:

weight, day, dose, total-amount, course-of-medication, and dosage-instructions. This decision was based on our idea of dividing objects into the following four categories:

- proper data objects, e.g., total-amount, `t`
- aggregate data objects, e.g., dosage-instructions, output
- control objects, e.g., beginning-of-the-week, first `for`-loop
- other/external, e.g., patient

The first example of each category above is a *Domain* description while the second is a *Program* description—for the fourth category no *Program* description is possible. The first two categories cover data objects and we selected them to be used as the basis for segmentation in the validation phase.

The synonym problem: Having decided the exact list of objects to look for, it was not always evident what words should be considered as synonyms for the selected objects. For example, under what conditions should the word "patient" be considered to be a synonym for the object "weight", or is the word "reading" a synonym for the object "dose" in the following:

> morning and evening **reading** in milliliters

The natural object problem: As with the Synonym Problem, it is not always clear whether natural language data references should be understood as a representative of some object included in the analysis, e.g.,

> plain **number** is multiplied by three
> right **amount** of medicine

The "value" problem: The word "value" is an object by itself in all the examples of the SDD. However, in several cases we were unsure if this really should be the case because we could not found any grounds for considering "value" as an object, e.g.,

accepts a negative value
its value
that value
the value of the variable

Problems in Segmenting

Problems in deciding what constitutes a segment are of two types: whether a sequence of words should be split into several segments, and whether single-word references should be skipped under some circumstances totally.

The qualification problem: It is common to have an object qualified by another object. In the SDD, these result in a single segment except when the qualifier is the word "value", e.g., (Good, 1999, p. 323):

The value of
the element of heights is preserved with that iteration.

In some cases references to two objects seem to represent those objects themselves rather than the qualified unity. The following example set starts with this kind of situation and proceeds gradually to references where a single segment might be more appropriate:

dose of the third day
dose of the day [in the first round of the second loop]
dose of the day [in any round of the second loop]
dose of a day [some day]
dose-of-the-day
daily dose

These examples demonstrate that the wording itself does not always explain the perceived presence of one or two objects, but the context of the reference counts also.

The multi-word reference problem: Objects are often referred to by several words representing different ODCs leading to a need to segment each passage as a separate segment, e.g.,

> total-amount (`m`)
> `m`, the total-amount, ...
> `m`, which is the total-amount, ...
> `m` — `m` is the total-amount — ...
> total-amount, which is kept in the variable `m`, ...
> the total-amount contained in `m`
> the total-amount is assigned to `m`

The variability is huge, and it is not at all clear how many segments should be used in each case. The category *Program—domain* may be used when there is a mixture of program and problem domain references (Good and Brna, 2004), but the examples in the SDD, e.g., "a list of marks", are cases where both parts of the reference are compulsory to make the passage understandable. Furthermore, there are no special categories for other combinations.

The pronoun problem: In the SDD, all examples of *Indirect references* are personal pronouns. It is unclear whether other pronouns, e.g., "which" and "that", are anaphoric references.

The verbatim problem: Examples and code segments may contain verbatim references to objects, e.g.,

> ```
> writeln('Day ', x, 'morning and evening ', d,
> 'ml.');
> ```
> Then the dosage-instructions is output: "Day 1. morning and evening 150 ml."

The expressions "`x`", "`d`", "Day", and "150 ml." do not actually refer to the objects but are verbatim copies of program or output text. It is unclear whether they should cause segmenting.

Problems in Selecting ODC Category

The context problem: This is similar to the Atmosphere Problem in IT coding but considers a single sentence rather than a larger context. In our material, sentences were much longer than those in the examples of the SDD, and the raters had different views of whether context should be taken into account.

The input coding problem: The SDD says that input is coded as *Program—real-world* but *Program* seemed to be more natural to the raters yielding differences between raters.

The Unclear-eagerness problem: The SDD does not state how eagerly the category *Unclear* should be applied: eager use will result in more reliable data but, of course, with fewer segments having a category that can be used in further analysis.

The natural language variable name problem: The name of one variable, `weight`, happened to be a natural language word carrying the meaning of the variable. This made it very hard to detect the correct category unless the context was obvious.

Rater Differences

Differences between raters were summarized in Section 4.3.2. Theoretically, there should have been *no differences in selecting the objects for segmenting* because we agreed on the set before the validation phase begun. However, *differences in segmenting* indicate that the recognition of even pre-defined objects is not easy: out of the 458 non-common segments, 63.6% were due to the Synonym and Natural Object Problems. The Pronoun Problem covered 17.9% of the non-common segments.

The vast majority of *differences in coding* were due to the Natural Language Variable Name Problem caused by the variable named `weight`. Out of the 570 common segments, only four were coded differently by all raters and they all referred to this variable. Out of the 149 segments coded similarly by two raters, 79.9% referred to this variable. Half of the rest were results of the Multi-word Reference Problem: the raters had used a single segment but based their coding on different aspects of the multi-word reference.

4.4 Discussion

Largest inter-rater differences in the IT category frequencies in Table 4.2 are over 5.0% (*Actions* 6.1%, *Elaborate* 5.9%, and *Data* 5.4%) but differences in the proportion of high-level information of program-related segments (HIP) were, however, smaller with the largest difference being 4.3%. The linear correlation coefficients of HIP values between two raters varied from 0.673 to 0.835, the probability of linear correlation of this magnitude to occur by change being smaller than 0.0001. In practice (Corritore and Wiedenbeck, 1991; Good, 1999; Hoadley et al., 1996; Sajaniemi and Kuittinen, 2005; Pennington, 1987a), individual IT category frequencies are usually grouped to high-level and low-level information making the smaller variability in HIP important for research purposes.

Inter-rater differences in the ODC category frequencies (Table 4.3) were even larger: 13.5% for *Domain*, 10.8% for *Program—domain*, and 6.7% for *Program*. The vast majority of the coding differences (80.4%) were, however, caused by the poor selection of a natural language word for one of the variables in the program to be summarized. Segmenting was more problematic for ODC than for IT.

In explaining differences among raters, the problems listed in Sections 4.3.3 and 4.3.4 are more important than their frequencies: many problems did not manifest themselves as differences in the validation phase because we devised solutions to them during the training phase. Our solutions were, however, *ad-hoc* and cannot be considered as general solutions if the scheme is to be applied in more general settings. Table 4.5 lists the problems together with our suggestions for solution types: whether a revision of the scheme is required, whether the problem can be solved by more detailed documentation of the scheme, whether the problem is due to problems in programs to be summarized, or whether the problem is still open.

In general, the documentation of the scheme should be improved by increasing consistency of examples and by including the summarized programs in order to make the relationship between coding examples and the programs explicit. For example, in the segment (Good, 1999, p. 316):

and whether **the person** has passed	Domain

Table 4.5: Suggested solution types for the problems. Scheme: revise the program summary scheme; Document.: revise the documentation of the scheme; Mater.: use appropriate experimental materials; Open: an open problem. The upper part contains problems related to information types (IT), and the lower part problems related to object description categories (ODC). S: segmenting problems; C: category selection problems; O: object recognition problems.

		Problem	Scheme	Document.	Mater.	Open
IT	S	"and"		•		
		non-finite clause	•	•		
		information–dense passage	•	•		
		example		•		
	C	atmosphere		•		
		not-done		•		
		missing "continuation" categ.	•			
		missing "irrelevant" categ.	•			
		Operations vs. Actions		•		
		Data vs. Operations/Actions				•
		Data vs. Control		•		
		State-low vs. State-high		•		
		Elaborate vs. some other categ.		•		
ODC	O	data object definition	•	•		
		synonym		•		
		natural object	•	•		
		"value"	•			
	S	qualification	•	•		
		multi-word reference	•	•		
		pronoun		•		
		verbatim		•		
	C	context		•		
		input coding	•			
		Unclear-eagerness		•		
		natural language variable name		•	•	

it is unclear whether persons are explicitly mentioned in the program (e.g., "Enter person's mark:") or only inferred (e.g., "Enter mark:"). This distinction is important in

explaining the correct solution to the Synonym Problem.

IT problems that require changes in the scheme are the addition of two new categories, and the abandonment of the subject–predicate requirement. As the SDD already contains examples with no subject or predicate, the latter solution can be considered not to be a change in the scheme but a documentation problem only. Another suggestion for improving the documentation is to use the new concept *information item* introduced in Section 4.3.3 to clarify the process of segmenting for IT analysis.

Other suggested IT documentation additions concern borderlines between individual categories, the atmosphere problem, and some special cases. The only open problem is the *Data* vs. *Operations/Actions* problem which covered 28.9% of the differences among segments common to all raters and depends on delicate interpretation of the true meaning of object descriptions.

ODC problems requiring changes in the scheme concern the selection of objects and the detection of object references. The documentation should be more precise on these problems, and furthermore suggest that the objects are listed before segmenting the summaries—particular, if there are more than a single rater. The scheme could be extended to include an option to use a limited set of objects only, e.g., objects that are explicitly mentioned in the program text both with a domain name and with a program name in order to avoid the *Unclear*-eagerness Problem, or to allow the use of a limited set of reference utterances for segmentation in order to avoid the Synonym and Natural Object Problems.

Other ODC documentation problems concern the context problem, and some special cases. The Natural Language Variable Name Problem can be avoided by careful design of programs to be summarized. The scheme cannot solve this problem although the documentation should mention it.

Chapter 5

Short-term Effects of Graphical versus Textual Visualisation of Variables

In this chapter, I describe an empirical experiment we carried out in order to study the short-term effects of visualising the roles of variables and present the results from the experiment. The experiment was first of a series of experiments in our research, and its purpose was to verify the existence of differences in the short-term effects of different visualisations and to work as a starting point for further studies. This chapter has been published in proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005) as a joint paper by Nevalainen and Sajaniemi under the title "Short-Term Effects of Graphical versus Textual Visualisation of Variables on Program Perception" (Nevalainen and Sajaniemi, 2005).

5.1 Introduction

To verify the effectiveness of a visualisation tool, it needs to be properly evaluated. Empirical evaluation of program visualisations has been based mostly on post-tests or pre- versus post-tests of participants' performance. These evaluations have resulted in a body of evidence suggesting that visualisations can have beneficial long-term effects on learning, when designed and used properly (Hundhausen et al., 2002; Byrne et al., 1999b; Hansen et al., 2000; Kann et al., 1997). Evaluation of post-test or pre- versus post-test performance of participants does not, however, provide clear insight into the

possible short-term effects of visualisations and their relation to the long-term effects.

In order to study these issues more rigorously, we conducted an experiment, in which we studied two visualisation tools for presenting program variables and their execution-time behavior. The first tool, PlanAni program animator (Sajaniemi and Kuittinen, 2004), presents pictorial metaphors for variables, operations on variables are animated, and information concerning the roles of the variables (Sajaniemi, 2002) is incorporated into the visualisation. The second tool, Turbo Pascal programming environment, provides textual representations of variables, operations on variables simply replace the value of the variable in the representation, and no role information is present. Previous studies (Sajaniemi and Kuittinen, 2004; Byckling and Sajaniemi, 2005) have reported the differences in long-term effects of the use of these two tools.

In order to determine the level of viewers' visual attention on program code and on visualisation of variables with the two visualisation tools, the locations of the participants' gaze on the screen were measured. Possible differences in the participants' mental models of the studied programs between the two visualisation tools were investigated by analysing participants' program summaries. In order to control possible differences between the participants, field-indepence of each participant was also measured.

In the following Sections, the experiment is described, and its results are presented and discussed.

5.2 Experiment

In order to study short-term effects of variable visualisations, we conducted an experiment where two visualisation tools for presenting information about variables and their values during program execution were used. The tools were used in the classroom experiment (Sajaniemi and Kuittinen, 2005), in which the long-term effects of the visualisations were investigated. The current study concentrates on possible differences in the locations of participants' visual attention and in participants' mental models of the studied programs between the two visualisation tools.

For control reasons, the level of field-independence of each participant was measured. Witkin (1971) defines field-dependence and field-independence as follows: "in a field-

dependent mode of perceiving, perception is strongly dominated by the overall organization of the surrounding field, and parts of the field are experienced as "fused". In a field-independent mode of perceiving, parts of the field are experienced as discrete from organized ground." Field-independence has been found to correlate positively with learning to program (Mancy and Reid, 2004), especially in computerised text-based and web-based environments (Parkinson and Redmond, 2002). Parkinson et al. (2004) have shown that the difference in performance between field-dependent and field-independent learners in computerised text-based and web-based environments can be diminished by accommodating field-dependence in the design of the environments.

The experiment consisted of 4 phases. In the first phase, the participants were asked to perform a test measuring participants' level of field-independence. In the second phase, the participants studied a recap material on roles of variables. The third phase consisted of viewing Pascal programs with the visualisation tools and of writing down program summaries. In the fourth phase, the participants filled a questionnaire about the visualisation tools.

5.2.1 Method

The experiment was a within-subject design with one independent variable (the visualisation tool) and two dependent variables (locations of the participant's gaze and the program summary provided by the participant). Locations of gaze were recorded using an eye-tracking camera (Tobii, 2004), and program summaries were analysed using Good's program summary analysis scheme (Good, 1999). The level of field-independence of the participants was measured using Group Embedded Figures Test (GEFT) (Witkin, 1971). The order of the visualisation tools and the order of the studied programs were counterbalanced.

Participants: Twelve participants, 7 male and 5 female, took part in the experiment. The participants were students who had taken an introductory programming course facilitating the roles of variables and continued their studies 1-2 years thereafter.

Materials: In the first phase, participants' level of field-independence was measured using GEFT test set (Witkin, 1971).

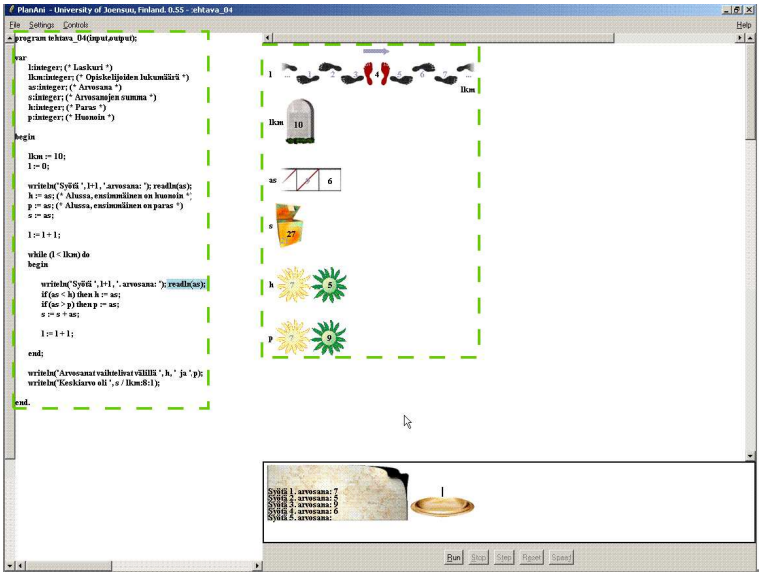

Figure 5.1: User interface of PlanAni program animator. (Dashed rectangles represent the code area and the variable visualisation area used in the analysis of gaze locations.)

In the second phase, written material from an earlier experiment (Sajaniemi and Stützle, Submitted) was used. The material consisted of descriptions of all roles and examples of their use. It included also a practice material consisting of three small Pascal programs with 14 variables, whose roles participants were asked to determine.

In the third phase, participants studied four simple Pascal programs. The programs were short (11-29 lines, empty lines omitted) and similar to the more difficult programs used in the introductory programming course of the earlier classroom experiment (Sajaniemi and Kuittinen, 2005). Participants entered predefined inputs for the programs. The use of fixed inputs enabled a participant to focus her attention to understanding the program, instead of wondering what inputs would be proper to the programs.

The visualisation tools used were PlanAni program animator (version 0.55) and the Turbo Pascal programming environment (version 5.5). In PlanAni (Figure 5.1), visualisations are graphical, operations on variables are animated, and information concerning the roles of variables is incorporated into the visualisations. Variable visualisations are located on the right side of the program code. PlanAni displays also notifications of each program action and has a separate area for input and output. In Turbo Pascal (Figure 5.2), visualisations are textual, operations on variables simply replace the old value of the variable with the new value, and no role information is presented. Variable visualisations are located below the program code. Turbo Pascal displays no notifica-

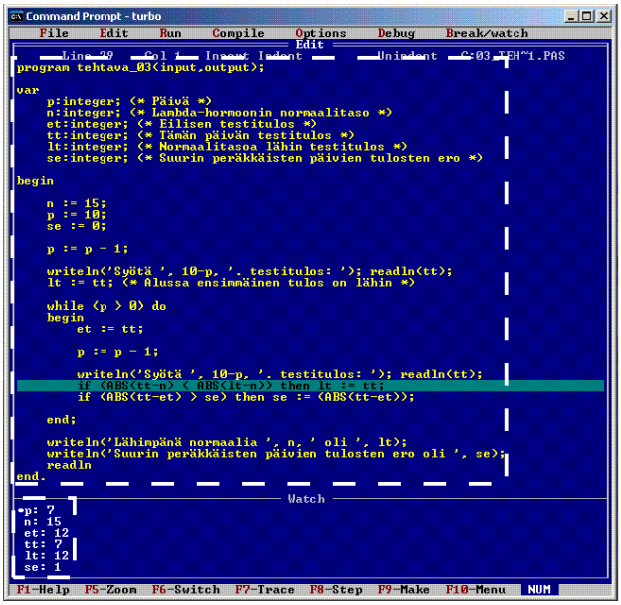

Figure 5.2: User interface of the Turbo Pascal programming environment. (Dashed rectangles represent the code area and the variable visualisation area used in the analysis of gaze locations.)

tions, and input and output are handled through command prompt. In the Turbo Pascal environment, watches displaying each variable and its values during execution were initialised in advance, and they served as textual visualisations of variables. Both visualisation tools display code and variables and were prepared so that participants were able to execute each program once, step by step. This limitation was used because the tools differed in many other aspects and we wanted to minimise differences having an influence on the participants.

In the fourth phase, participants were asked to evaluate the visualisation tools with an evaluation form including Likert scale questions and open questions about the tools and their use. In the Likert scale questions, participants were asked to use a scale of 1-5 (1 = totally disagree, 5 = totally agree) to statements concerning five characteristics of the visualisations: originality, pleasure, salience, understandability, and usefulness. For example, the understandability of the visualisations was evaluated by proposition “I found this representation easy to understand”. These characteristics were derived from experiments carried out by Hübscher-Younger and Narayanan (2003) who used them to characterise student visualisations of algorithms. In the open questions, participants were asked to report what issues the two visualisations did and did not highlight. The evaluation form included also a possibility for free commentary.

Procedure: Participants were run individually. Each participant's level of field-independence was measured with the GEFT test consisting of three phases that lasted 2 minutes, 5 minutes, and 5 minutes. After this, the participant was given 15 minutes to study the roles of variables recap material and perform the practice task. Then, after a short break, the participant was seated in front of a computer monitor that has an eye-tracking camera embedded in the panels. The procedure of measuring the movement of her eyes was explained to the participant, and she was advised about the locations of all available information on the screen for both visualisation tools. The participant had then an unlimited time to study each program. After the participant had finished studying a program, the program was dismissed from the screen, and she was instructed to give a written description of the program. Again, the time to do this was not limited, and the participant was not instructed in any way on what the program description should comprise of. The first two programs were shown with one visualisation tool and the next two with the other tool. The first program with each visualisation tool was used to familiarise the participant with the tool, and data from the second program only was analysed. When all four programs had been studied, the participant was asked to evaluate the visualisation tools.

5.2.2 Results

GEFT test results of participants' levels of field-independence are shown in Table 5.1. In the GEFT test, higher score means higher level of field-independence, and the theoretical maximum is 18.

Table 5.1: Results of GEFT test measuring participants' levels of field-independence.

	n	Min	Max	Mean	SD
GEFT score	12	7	18	14.75	3.22

The participants used on an average 26 minutes and 18 seconds to study a program with PlanAni. Standard deviation was 5 minutes and 58 seconds. With Turbo Pascal, mean time to study a program was 6.08 (SD 1.57). Due to the difference in the speed of the animation between the two tools, the minimum time it takes to view the shorter of the two analysed programs with the tools is 11.30 for PlanAni and 1.00 for Turbo Pascal.

Table 5.2: Mean proportions of viewing times on the three areas of the screen (*** = $p < 0.001$).

Code	Screen Area	Condition			
		PlanAni		Turbo Pascal	
		Mean	SD	Mean	SD
COD	Code ***	20.36	3.27	38.75	2.13
VAR	Variables ***	15.43	3.59	3.11	2.38
OTH	Other ***	64.21	2.89	58.14	2.52

For the purpose of the analysis, the screen was divided into three areas. The code area and the variable area were formed by taking the smallest bounding box that includes the symbols used in the code or the variable visualisations. These areas are illustrated by the dashed rectangles in Figures 5.1 and 5.2. Other parts of the screen formed the third area. A two-way within-subject Analysis of Variance was carried out. The ANOVA on absolute viewing times showed that there was a significant main effect of visualisation tool ($F(1,9) = 156.956$, $p < 0.001$), and of screen area ($F(2,9) = 78.125$, $p < 0.001$), and also a significant two-way interaction of visualisation tool and screen area ($F(2,9) = 55.984$, $p < 0.001$). The mean proportions of viewing times on these three areas are presented in Table 5.2. Paired t-test with Bonferroni correction was used for follow-up testing. The difference between PlanAni and Turbo Pascal is significant in the proportional viewing time on code ($t = -17.036$, $df = 11$, $p < 0.001$), variables ($t = 8.721$, $df = 11$, $p < 0.001$), and other parts of the screen ($t = 5.708$, $df = 11$, $p < 0.001$).

In order to study participants' mental models of the studied programs, we used Good's program summary analysis scheme (Good, 1999) that consists of two classifications: one based on information types (IT) and the other based on object descriptions (ODC). The information types classification is used to code summary statements on the basis of the information types they contain. Table 5.3 contains the distribution of information type statements in each condition. The object descriptions classification looks at the way in which objects are described. Table 5.4 contains the distribution of object description statements in each condition. No statistically significant differences between the conditions were found in information types or object description classifications.

We analysed the distribution of domain versus program information in participants'

Table 5.3: Mean proportions of IT categories used in program summaries.

Code	Information Type	Condition			
		PlanAni		Turbo Pascal	
		Mean	SD	Mean	SD
FUN	Function	14.73	32.96	8.33	21.62
ACT	Actions	17.98	18.12	16.08	16.98
OPE	Operations	13.16	15.42	10.64	12.83
SHI	State-high	4.42	5.15	2.38	5.79
SLO	State-low	3.84	5.92	3.68	6.18
DAT	Data	36.69	25.56	41.40	25.41
CON	Control	3.70	6.35	6.86	10.88
ELA	Elaborate	3.45	5.69	8.15	17.34
MET	Meta	0.52	1.79	0.38	1.30
IRR	Irrelevant	1.13	3.93	1.07	2.62
UNC	Unclear	0.00	0.00	0.00	0.00
INC	Incomplete	0.00	0.00	0.00	0.00
CUT	Continuation	0.00	0.00	0.38	1.30
HIG	FUN+ACT+SHI+DAT	73.80	25.54	68.18	24.55
LOW	OPE+SLO+CON	21.08	22.97	21.86	19.93
OTH	100-HIG-LOW	5.12	8.86	9.96	17.61
HIP	HIG / (HIG+LOW) * 100	77.35	23.78	76.08	22.11

program summaries further by using a similar strategy as Sajaniemi and Kuittinen (2005). We sorted program summaries into three types depending on the amount of domain versus program statements in object descriptions. Summaries with at least 67% domain statements (indirect and unclear statements excluded) were called *domain-level summaries*, summaries with at least 67% program and program only statements were classified as *program-level summaries*, and all others were called *cross-referenced summaries* because they had a more even distribution of domain and program information. The number of cross-referenced summaries was two in PlanAni condition and four in Turbo Pascal condition. This difference is not statistically significant (Fisher's exact test).

Participants' evaluation of the visualisation tools is presented in Table 5.5. The difference between PlanAni and Turbo Pascal is significant in both originality ($t = 7.374$,

Table 5.4: Mean proportions of ODC categories used in program summaries.

Code	Object Description Category	Condition			
		PlanAni		Turbo Pascal	
		Mean	SD	Mean	SD
PON	Program only	0.98	3.41	1.89	6.55
PRO	Program	1.96	6.78	2.25	6.56
PRR	Program—real-world	18.19	26.10	16.22	18.74
PRD	Program—domain	2.19	4.23	1.81	6.26
DOM	Domain	74.92	29.41	77.13	22.23
IND	Indirect reference	1.78	2.64	0.69	2.40
UNO	Unclear	0.00	0.00	0.00	0.00

Table 5.5: Participants' evaluation of different characteristics of the two visualisation tools (scale 1-5); the best is 5 (*** = $p < 0.001$, ** = $p < 0.01$).

Characteristic	Condition			
	PlanAni		Turbo Pascal	
	Mean	SD	Mean	SD
Originality ***	3.92	0.79	2.00	0.60
Pleasure	2.58	1.00	3.42	1.00
Salience **	4.00	0.43	3.00	0.74
Understandability	4.25	0.75	3.50	0.91
Usefulness	3.00	1.04	2.50	0.80

$df = 11, p < 0.001$) and salience ($t = 4.690, df = 11, p = 0.001$).

The correlations between participants' levels of field-independence and the dependent variables—the proportions of time used for viewing the program variables, and the proportions of different information types and object description categories in participants' program descriptions—were analysed using the Pearson correlation coefficient. Variables having statistically significant correlation with proportion of time used for viewing the visualisations of program variables (VAR) are shown in Table 5.6. There were no statistically significant correlations between proportion of time used for viewing the code and any of the variables.

In PlanAni condition, the Pearson correlation coefficient between proportion of vari-

Table 5.6: Variables having statistically significant correlation with time used for viewing the visualisations of program variables (** = $p < 0.01$, * = $p < 0.05$).

Correlation	Condition			
	PlanAni		Turbo Pascal	
VAR versus GEFT score	0.688	*	-0.071	
VAR versus HIP	0.601	*	-0.084	
VAR versus OPE	-0.655	*	0.042	
VAR versus SLO	-0.725	**	0.025	
VAR versus PRR	-0.445		-0.747	**

able viewing and the GEFT-score is $r = 0.688$, the two-tailed probability for a correlation of such magnitude to occur by chance being statistically significant ($t = 3.001$, $df = 10$, $p = 0.0133$). In PlanAni, correlation is statistically significant also between proportion of variable viewing and high-level IT-descriptions (HIP) ($r = 0.601$, $t = 2.377$, $df = 10$, $p = 0.0388$), proportion of variable viewing and operation level IT-descriptions (OPE) ($r = -0.655$, $t = -2.741$, $df = 10$, $p = 0.0208$), and proportion of variable viewing and state-low level IT-descriptions (SLO) ($r = -0.725$, $t = -3.331$, $df = 10$, $p = 0.0076$). In Turbo Pascal, statistically significant correlation occurs between proportion of variable viewing and program—real-world object descriptions (PRR) ($r = -0.747$, $t = -3.556$, $df = 10$, $p = 0.0052$).

5.3 Discussion

The purpose of this experiment was to investigate how a person targets her visual attention, and what kind of a mental model she constructs of a computer program, when the program is presented using a textual or a graphical program visualisation tool. The experiment is first in a series of experiments that will study in detail the effects of the visualisation of roles of variables in PlanAni. Two completely different visualisation tools were selected for the current experiment in order to bring forth clearly different effects that different visualisation tools might produce and to provide this way a starting point for a more detailed investigation in future. Furthermore, the long-term effects of these two tools have been studied earlier (Sajaniemi and Kuittinen, 2004; Byckling and Sajaniemi, 2005).

The results indicate that participants spent more time viewing both the code and the variables with PlanAni than they did with Turbo Pascal. Part of this can be explained by the difference in the speed of the animation. Another explaining factor may be the difference in the graphical richness and amount of details between the two tools.

The variable visualisations were viewed proportionately more with PlanAni than with Turbo Pascal ($p < 0.001$) which means that the animation tool has an effect on visual attention. One explaining factor is the location of animation: in PlanAni, most animations appear within the variable visualisations, whereas in Turbo Pascal they appear in the code area. The effect of other factors, e.g., the pleasantness of the images must be studied separately. The other area of the screen was viewed proportionately more with PlanAni than with Turbo Pascal ($p < 0.001$). This was probably because the area was substantially larger in PlanAni, and because it displayed input and output of the program to the viewer constantly, instead of displaying them only in command prompt.

Program summaries were used to study the mental models of the participants. No statistically significant differences were found between the two tools. However, in PlanAni the proportion of variable viewing correlated positively with high-level information ($r = 0.601$) and negatively with operations ($r = -0.655$) and state-low ($r = -0.725$) in program summaries. Thus the increase of visual attention in the variable visualisation area increased high-level data-related information; and the increase of visual attention to the code area increased low-level code-related information. In Turbo Pascal these effects could not be found. Thus either the smaller absolute time increase was not sufficient to cause changes in mental model or the Turbo Pascal interface did not provide the information required for the high-level mental model because it lacks role information.

In Turbo Pascal, proportion of variable viewing correlated negatively with program-real world object descriptions ($r = -0.747$; PlanAni $r = -0.445$). The program-real world object descriptions typically contained expressions such as “value” and “number”, which were used in a similar way as program object descriptions in describing the low-level operations of the programs. With both tools the increase of visual attention in variable visualisations decreased the participants use of low-level descriptions of the programs.

PlanAni has earlier been found to have positive long-term effects on programming

skills and content of mental models (Sajaniemi and Kuittinen, 2004; Byckling and Sajaniemi, 2005), but in this experiment such an overall effect could not be found. In addition to the location of visual attention, a person's mental model is influenced by other factors also. Hübscher-Younger and Narayanan (2003) have used six characteristics of visualisation tools and studied their effect on learning. They found pleasure and salience to be the two most important characteristics influencing learning. We asked our participants to evaluate both visualisation tools with five of these characteristics: originality, pleasure, salience, understandability and usefulness. The evaluation form included also open questions and a possibility for free commentary. In our experiment, participants judged PlanAni to be more salient ($p = 0.001$) than Turbo Pascal, but also more unpleasant ($p = 0.096$). One possible reason for the small effect on mental models in this experiment is that even though PlanAni was judged more salient, it was also found unpleasant to use.

On the basis of the open questions, it is obvious that the unpleasantness of PlanAni was mostly due to the slowness of the animation. Eight participants commented negatively about the slowness of PlanAni, and four commented positively about the fast use of Turbo Pascal. The salience of PlanAni was contributed mostly to the illustration of variables' roles and tasks in the program. These were commented positively by the participants five times (roles of the variables) and four times (tasks of the variables). In some of the free commentary by the participants, PlanAni was deemed to be appropriate for teaching elementary programming, not for intermediates. Because we had 2nd and 3rd year students in the experiment, the programs were easy for the participants and therefore properties highlighted by animation may have not shown up in the program summaries. This can also partly explain the unpleasantness the participants felt in using PlanAni, a tool designed for true novices.

The proportion of variable viewing correlated with the GEFT score in PlanAni ($r = 0.688$), but not in Turbo Pascal ($r = -0.071$). This can be explained by the difference in graphical richness between the two visualisation tools. Following the textual visualisation of Turbo Pascal does not require the viewer to be able to separate items from organised perceptual field in the same way as with PlanAni, which uses colorful graphical images and animations. This is consistent with previous experiments (Mancy and Reid, 2004; Parkinson and Redmond, 2002) that have studied the relationship between field-independence and learning. Thus, the level of field-independence influenced the targeting of visual attention, which influenced the mental model being

constructed. Therefore, the level of field-independence has direct implications on the usefulness of visualisations.

Chapter 6

Conclusion

In this thesis, I have reported a research in which we study the short-term effects of variable visualisation by investigating the interaction between the form of visualisations, the visual attention of the students, and the mental models students form concerning computer programs when they view programs with PlanAni program animator. This way, we try to find out how the students perceive the visualisations they are viewing, and how this relates to the quality of their comprehension of the visualised variables and programs.

We will use eye tracking to measure visual attention of the students. To provide verification for our methodology, we conducted an experiment comparing the use of three eye tracking devices in a psychology of programming experiment in which participants studied short computer programs using a program animator. The devices were ASL 501 Head Mounted Optics, ASL 504 Pan/Tilt Optics, and Tobii 1750.

The results show that there are significant differences in the accuracy and easiness of use between the three devices. For example, the ASL 501 Head Mounted Optics required approximately twice as much time for the preparation of a tracking process than the other two devices. The ASL 501 was also the least accurate of the devices when it was used for the task in which the participant viewed a computer screen. This can be partly explained by inaccuracies in the manual correction of the shifting effect, which can be removed by using magnetic head tracker with ASL 501.

When considering the amount of effort the devices required from the participant and the amount of disturbance they caused to the participant, Tobii 1750 seemed to be the

least obtrusive. The device allowed the participant to take part in the detection of the eyes and the process of calibration was performed without step-by-step dictation of the operator. The monitoring process didn't contain any clear differences between the devices. The ASL 504 needed to be aided by relocating the eye manually in some cases. On the basis of these results, we selected Tobii 1750 for measuring the eye movements of participants in our research.

Novice programmers' mental models can be studied using Good's program summary analysis scheme (Good, 1999), which can also be used to assess the quality of comprehension. To analyse the replicability of the scheme, we carried out an investigation of the inter-rater reliability of the scheme.

The investigation consisted of a training phase and a validation phase. The differences in ratings were analyzed, and all problems encountered during the whole investigation were recorded. Finally, possible solution types for the problems were discussed. In spite of the training and the mutual agreement on coding rules there were noticeable differences among the raters. Even though the correlation between the raters was high, the results suggest that absolute frequencies obtained in separate experiments should not be compared with each other. The findings do however indicate that by improving the scheme and its documentation, most of the observed inter-rater differences can be avoided. The only open problem concerns making the distinction between descriptions of data and activities in cases where the specific words that are used, or the abstractness of expression may affect the interpretation and hence the selection of the appropriate information type. These problems do not, however, prevent the use of the scheme in situations, where there is only one rater, as in the case of the experiments in our research.

Using the presented research methodology, we will conduct a series of empirical experiments. In the first experiment, we studied how a person targets her visual attention, and what kind of a mental model she constructs concerning a computer program, when the program and especially its variables are presented using either a textual or a graphical program visualisation tool. PlanAni program animator uses role images and animations on these images to highlight program variables, while Turbo Pascal displays variables and their values textually and without role information. These two tools were selected because their long-term effects on learning had been analysed previously in Sajaniemi and Kuittinen (2005), and in Byckling and Sajaniemi (2005).

The results indicate that visual attention of the participants was targeted on the variable visualisations clearly more with PlanAni than with Turbo Pascal. In PlanAni, the increase of visual attention to variables increased the proportion of high-level data-related information in program summaries and decreased low-level code-related information, thus effecting the mental models of the participants. In Turbo Pascal, these effects could not be found. Moreover, with PlanAni the proportion of variable viewing correlated positively with the level of field-independence. Thus field-independent students benefit more by the graphically rich PlanAni program animator than field-dependent students. PlanAni has earlier been found to have positive long-term effects on programming skills and content of mental model (Sajaniemi and Kuittinen, 2005; Byckling and Sajaniemi, 2005), but in this experiment such an overall effect could not be found.

The experiment established that there exists differences in short-term effects of different visualisations and works as a starting point for further studies. In forthcoming experiments, we will investigate the differences in the short-term effects in more detail, utilising classifications of factors of graphics and animation in visualisations. Possible classifications include for example eight visual variables introduced by Bertin (1983), that can be identified in an image, and algorithm animation taxonomy by Brown (1998). Our goal is to gather information on the contribution of the factors to the overall attractiveness of visualisations, and the effects of these factors on visual attention and mental models of the viewers.

Bibliography

Ainsworth, S. E., Labeke, N. V., 2002. Using a multi-representational design framework to develop and evaluate a dynamic simulation environment. In: Dynamic Information and Visualisation Workshop.

Ashmore, M., Duchowski, A. T., Shoemaker, G., 2005. Efficient eye pointing with a fisheye lens. In: Proceedings of the 2005 conference on Graphics interface. Canadian Human-Computer Communications Society, pp. 203–210.

ASL, 2003a. Eye Tracking System Instruction Manual - Model 501 Head Mounted Optics. Applied Science Laboratories.

ASL, 2003b. Eye Tracking System Instruction Manual - Model 504 Pan/Tilt Optics. Applied Science Laboratories.

Baecker, R., B.Price, 1998. The early history of software visualization. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 29–34.

Bednarik, R., Tukiainen, M., 2004. Visual attention and representation switching in java program debugging: a case study using eye movement tracking. In: Dunican, E., Green, T. (Eds.), The 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004). pp. 159–169.

Ben-Ari, M., Sajaniemi, J., 2004. Roles of variables as seen by cs educators. In: The 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004). Association for Computing Machinery, pp. 52–56.

Bertin, J., 1983. Semiology of Graphics. University of Wisconsin Press.

Brooks, R., 1977. Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies 9, 737–751.

Brooks, R., 1983. Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies 18, 543–554.

Brown, M., 1998. A taxonomy of algorithm animation displays. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 35–42.

Brown, M. H., 1991. ZEUS: A system for algorithm animation and multi-view editing. In: Proceedings of the 1991 IEEE Workshop on Visual Languages. IEEE Computer Society Press, pp. 4–9.

Burkhardt, J.-M., Détienne, F., Wiedenbeck, S., 1997. Mental representations constructed by experts and novices in object-oriented program comprehension. In: Howard, S., Hammond, J., Lindgaard, G. (Eds.), Human-Computer Interaction INTERACT '97. Chapman & Hall, Ltd, pp. 339–346.

Byckling, P., Gerdt, P., Sajaniemi, J., 2005. Roles of variables in object-oriented programming. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2005). Association for Computing Machinery, pp. 350–355.

Byckling, P., Kuittinen, M., Nevalainen, S., Sajaniemi, J., 2004. An inter-rater reliability analysis of good's program summary analysis scheme. In: Dunican, E., Green, T. (Eds.), Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004). pp. 170–184.

Byckling, P., Sajaniemi, J., 2005. Using roles of variables in teaching: Effects on program construction. In: Romero, P., Good, J., Bryant, S., Chaparro, E. A. (Eds.), Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005). pp. 278–292.

Byrne, M. D., Anderson, J. R., Douglass, S., Matessa, M., 1999a. Eye tracking the visual search of click-down menus. In: Human Factors in Computing Systems: CHI'99 Conference Proceedings. ACM Press, pp. 402–409.

Byrne, M. D., Catrambone, R., Stasko, J. T., 1999b. Evaluating animations as student aids in learning computer algorithms. Computers & Education 33, 253–278.

Corritore, C. L., Wiedenbeck, S., 1991. What do novices learn during program comprehension? International Journal of Human-Computer Interaction 3 (2), 199–222.

Crosby, M., Stelovsky, J., 1989. The influence of user experience and presentation medium on strategies of viewing algorithms. In: Vol. II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. pp. 438–446.

Dillon, A., Gabbard, R., 1998. Hypermedia as an educational technology: A review of the quantitative research literature on learner comprehension, control and style. Review of Educational Research 68 (3), 322–349.

Duchowski, A. T., 2003. Eye Tracking Methodology - Theory and Practice. Springer-Verlag.

Ehrlich, K., Soloway, E., 1984. An empirical investigation of the tacit plan knowledge in programming. In: Thomas, J. C., Schneider, M. L. (Eds.), Human Factors in Computer Systems. Norwood, NJ: Ablex Publishing Company, pp. 113–133.

Eick, S. G., 1998. Maintenance of large systems. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 315–328.

Findlay, J. M., 1992. Programming of stimulus-elicited saccadic eye movements. In: Rayner, K. (Ed.), Eye Movements and Visual Cognition: Scene Perception and Reading. Springer Verlag, pp. 8–30.

Goldberg, J. H., Kotval, X. P., 1999. Computer interface evaluation using eye movements. International Journal of Industrial Ergonomics 24, 631–645.

Goldberg, J. H., Stimson, M. J., Lewenstein, M., Scott, N., Wichansky, A. M., 2003. Eye tracking in web search tasks: design implications. In: Proceedings of the symposium on Eye tracking research & applications. ACM Press, pp. 51–58.

Good, J., 1999. Programming paradigms, information types and graphical representations: Empirical investigations of novice program comprehension. Ph.D. thesis, University of Edinburgh.

Good, J., Brna, P., 2004. Program comprehension and authentic measurement: A scheme for analysing descriptions of programs. International Journal of Human-Computer Studies 61, 169–185.

Green, T. R. G., Cornah, A. J., 1985. The programmer's torch. In: Human-Computer Interaction - INTERACT'84. IFIP, Elsevier Science Publishers (North-Holland), pp. 397–402.

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.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., Vanninen, P., 1997. Animation of user algorithms on the Web. In: VL'97, IEEE Symposium on Visual Languages. IEEE Computer Society Press, pp. 360–367.

Hansen, S. R., Narayanan, N. H., Schrimpsher, D., 2000. Helping learners visualize and comprehend algorithms. Interactive Multimedia Electronic Journal of Computer-Enhanced Learning 1.

Helttula, E., Hyrskykari, A., Räihä, K., 1989. Graphical specification of algorithm animations with ALADDIN. In: Proceedings of the Hawaii International Conference on Systems and Software. IEEE, pp. 892–901.

Hoadley, C. M., Linn, M. C., Mann, L. M., Clancy, M. J., 1996. When, why and how do novice programmers reuse code? In: Gray, W. D., Boehm-Davis, D. A. (Eds.), Empirical Studies of Programmers: Sixth Workshop. Ablex Publishing Company, pp. 109–129.

Hübscher-Younger, T., Narayanan, N. H., 2003. Dancing hamsters and marble statues: Characterizing student visualizations of algorithms. In: ACM 2003 Symposium on Software Visualization (SoftVis 2003). Association for Computing Machinery, pp. 95–104.

Hughes, C., Buckley, J., 2004. Evaluating algorithm animation for concurrent systems: A comprehension-based approach. In: Dunican, E., Green, T. (Eds.), Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group. pp. 193–205.

Hundhausen, C. D., Douglas, S. A., Stasko, J. T., 2002. A meta-study of algorithm visualization effectiveness. Journal of Visual Languages and Computing 13, 259–290.

Hyönä, J., Radach, R., Deubel, H., 2003. The Mind's Eye : Cognitive and Applied Aspects of Eye Movement Research. North-Holland.

Jacob, R. J., 1990. What you look at is what you get: Eye movement-based interaction techniques. In: Human Factors in Computing Systems: CHI '90 Conference Proceedings. ACM Press, pp. 11–18.

Kann, C., Lindeman, R. W., Heller, R., 1997. Integrating algorithm animation into a learning environment. Computers & Education 28, 223–228.

Kennedy, A., 1992. The spatial coding hypothesis. In: Rayner, K. (Ed.), Eye Movements and Visual Cognition: Scene Perception and Reading. Springer Verlag, pp. 379–396.

Kimelman, D., Rosenburg, B., Roth, T., 1998. Visualization of dynamics in realword software systems. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 293–314.

Kuittinen, M., Sajaniemi, J., 2004. Teaching roles of variables in elementary programming courses. In: The 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004). Association for Computing Machinery, pp. 57–61.

Kulikova, Y., 2005. Roles of variables in functional programming. Master's Thesis, Department of Computer Science, University of Joensuu, Finland.

Lahtinen, S.-P., Sutinen, E., Tarhio, J., 1998. Automated animation of algorithms with Eliot. Journal of Visual Languages and Computing 9, 337–349.

Lieberman, H., Fry, C., 1998. ZStep 95: A reversible, animated source code stepper. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 277–292.

Majaranta, P., Aula, A., Räihä, K., 2004. Effects of feedback on eye typing with a short dwell time. In: Proceedings of the Eye tracking research & applications symposium. ACM Press, pp. 139–146.

Mancy, R., Reid, N., 2004. Aspects of cognitive style and programming. In: Dunican, E., Green, T. (Eds.), Proceedings of the Sixteenth Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004). pp. 1–9.

Mayer, R. E., 2001. Multimedia learning. Cambridge University Press, U.K.

Mulholland, P., 1998. A principled approach to the evaluation of SV: A case study in Prolog. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 439–451.

Myers, B., 1990. Taxonomies of visual programming and program visualisation. Journal of Visual Languages and Computing 1, 97–123.

Nevalainen, S., Sajaniemi, J., 2004. Comparison of three eye tracking devices in psychology of programming research. In: Dunican, E., Green, T. (Eds.), Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004). pp. 151–158.

Nevalainen, S., Sajaniemi, J., 2005. Short-term effects of graphical versus textual visualization of variables on program perception. In: Romero, P., Good, J., Bryant, S., Chaparro, E. A. (Eds.), Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005). pp. 77–91.

O'Brien, M. P., Shaft, T. M., Buckley, J., 2001. An open-source analysis schema for identifying software comprehension processes. In: Kadoda, G. (Ed.), Thirteenth Workshop of the Psychology of Programming Interest Group. pp. 129–146.

O'Shea, P., Exton, C., 2004. Investigating patterns and task type correlations in open source mailing lists for programmer comprehension. In: Dunican, E., Green, T. (Eds.), Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group. pp. 185–192.

Parkinson, A., Redmond, J. A., 2002. Do cognitive styles affect learning performance in different computer media? In: The 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002). Association for Computing Machinery, pp. 39–43.

Parkinson, A., Redmond, J. A., Walsh, C., 2004. Accommodating field-dependence: A cross-over study. In: The 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004). Association for Computing Machinery, pp. 72–76.

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

Pennington, N., 1987b. Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology 19, 295–341.

Petre, M., Blackwell, A., Green, T. R. G., 1998. Coqnitive questions in software visualisation. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 453–480.

Petre, M., Blackwell, A. F., 1999. Mental imagery in program design and visual programming. International Journal of Human-Computer Studies 51 (1), 7–30.

Posner, M. I., Snyder, C. R. R., Davidson, B. J., 1980. Attention and the detection of signals. Experimental Psychology:General 109, 160–174.

Price, B., Baecker, R., Small, I., 1993. A principled taxonomy of software visualisation. Journal of Visual Languages and Computing 4, 211–266.

Ramadhan, H. A., 2000. Programming by discovery. Journal of Computer Assisted Learning 16, 83–93.

Rayner, K., 1992. Eye Movements and Visual Cognition: Scene Perception and Reading. Springer-Verlag.

Rayner, K., 1998. Eye movements in reading and information processing: 20 years of research. Psychological Bulletin 124, 372–422.

Reiss, S. P., 1998. Visualisations for software engineering — programming environments. In: Stasko, J. T., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization – Programming as a Multimedia Experience. The MIT Press, pp. 259–276.

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

Rist, R. S., 1991. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. Human-Computer Interaction 6, 1–46.

Rössling, G., Freisleben, B., 2002. ANIMAL: A system for supporting multiple roles in algorithm animation. Journal of Visual Languages and Computing 13, 341–354.

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 (HCC'02). IEEE Computer Society, pp. 37–39.

Sajaniemi, J., 2004. Roles of variables home page. http://www.cs.joensuu.fi/~saja/var_roles/, (Accessed Dec. 14th, 2005).

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., Kulikova, Y., Submitted. Roles of variables in three programming paradigms.

Sajaniemi, J., Kuittinen, M., 2003. Program animation based on the roles of variables. In: ACM 2003 Symposium on Software Visualization (SoftVis 2003). Association for Computing Machinery, pp. 7–16.

Sajaniemi, J., Kuittinen, M., 2004. Visualizing roles of variables in program animation. Information Visualization 3, 137–153.

Sajaniemi, J., Kuittinen, M., 2005. An experiment on using roles of variables in teaching introductory programming. Computer Science Education 15 (1), 59–82.

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

Sajaniemi, J., Stützle, T., Submitted. Evaluation techniques for animated software visualization metaphors.

Sibert, L. E., Jacob, R. J., 2000. Evaluation of eye gaze interaction. In: Human Factors in Computing Systems: CHI 2000 Conference Proceedings. ACM Press, pp. 281 – 288.

Stasko, J., Patterson, C., 1992. Understanding and characterizing software visualization systems. In: Proceedings of the 1992 IEEE Workshop on Visual Languages. IEEE Computer Society Press, pp. 3–10.

Stützle, T., Sajaniemi, J., 2005. An empirical evaluation of visual metaphors in the animation of roles of variables. Informing Science Journal 8, 87–100.

Tobii, 2004. User Manual - Tobii Eye Tracker, Clearview Analysis Software. Tobii Technology AB.

von Mayrhauser, A., Lang, S., 1999. A coding scheme to support systematic analysis of software comprehension. IEEE Transactions on Software Engineering 25 (4), 526–540.

von Mayrhauser, A., Vans, A. M., 1995. Industrial experience with an integrated code comprehension model. Software Engineering Journal 10, 171–182.

Witkin, M. A., 1971. A Manual for the Embedded Figures Test. Consulting Psychologists Press.

Appendix

A List of Roles of Variables (Sajaniemi and Kuittinen, 2003)

Following ten roles cover 99% of all variables found in novice-level imperative programs:

Fixed value (aka constant): A variable whose value does not change after initialization.

Technical definition: A variable whose value does not change after initialization (e.g., an input value stored in a variable that is not changed later) possibly done in several alternative assignment statements (e.g., a variable that is set to true if the program is executed during a leap year, and false otherwise) and possibly corrected immediately after initialization (e.g., an input value that is replaced by its absolute value if it is negative).

Full program example:

The variable fuel is a fixed value in the following program:

```
program fuelRate;
var start, finish: integer;
    fuel: real;
begin
    write('Enter fuel amount: '); readln(fuel);
    write('Enter odometer at start: '); readln(start);
    write('Enter odometer at finish: '); readln(finish);
    while finish <= start do begin
        write('Odometer cannot be smaller at finish. Re-enter: ');
        readln(finish)
    end;
    writeln('Average rate of fuel consumed was ',
             fuel / (finish - start) )
end.
```

Stepper: A variable stepping through a succession of values that can be predicted as soon as the succession starts.

Technical definition: A variable going through a succession of values depending on its own previous value and possibly on other steppers, stepper followers, and fixed values (e.g., a counter of input values, a variable that doubles its value every time it is updated, a variable that alternates between two values, or an index to an array that sweeps through the array using varying densities) even though the selection of possibly alternative update assignments may depend on other variables (e.g., the search index in binary search).

Full program example:

The variable month is a stepper in the following program:

```
program monthlySales;
var month, largest: integer;
    sales: array [1..12] of integer;
begin
    for month := 1 to 12 do begin
        write('Enter sales of month ', month, ': ');
        readln(sales[month])
    end;
    largest := sales[1];
    for month := 2 to 12 do
        if largest < sales[month] then largest := sales[month];
    for month := 1 to 12 do begin
        writeln('Month ', month, ' gave ', largest-sales[month],
                ' less than the best month.')
    end
end.
```

Most-recent holder: A variable holding the latest value encountered in going through a succession of values.

Technical definition: A variable holding the latest value encountered in going through a succession of values (e.g., the latest input read, or a copy of an array element last referenced using a stepper) and possibly corrected immediately after obtaining a new value (e.g., to scale into internal data representation format).

Full program example:

The variable finish is a most-recent holder in the following program:

```
program fuelRate;
var start, finish: integer;
    fuel: real;
begin
```

```
    write('Enter fuel amount: '); readln(fuel);
    write('Enter odometer at start: '); readln(start);
    write('Enter odometer at finish: '); readln(finish);
    while finish <= start do begin
        write('Odometer cannot be smaller at finish. Re-enter: ');
        readln(finish)
    end;
    writeln('Average rate of fuel consumed was ',
             fuel / (finish - start) )
end.
```

Most-wanted holder: A variable holding the best value encountered so far in going through a succession of values.

Technical definition: A variable holding the best value encountered so far in going through a succession of values with no restriction on how to measure the goodness of a value (e.g., largest input seen so far, or an index to the smallest array element processed so far).

Full program example:

The variable largest is a most-wanted holder in the following program:

```
program monthlySales;
var month, largest: integer;
    sales: array [1..12] of integer;
begin
    for month := 1 to 12 do begin
        write('Enter sales of month ', month, ': ');
        readln(sales[month])
    end;
    largest := sales[1];
    for month := 2 to 12 do
        if largest < sales[month] then largest := sales[month];
    for month := 1 to 12 do begin
        writeln('Month ', month, ' gave ', largest-sales[month],
                ' less than the best month.')
    end
end.
```

Gatherer: A variable accumulating the effect of individual values in going through a succession of values.

Technical definition: A variable accumulating the effect of individual values in going through a succession of values (e.g., a running total, or the total number of cards in hand when the player may draw several cards at a time).

Full program example:

The variable totalSales is a gatherer in the following program:

```
program sales;
var month, monthSales, totalSales: integer;
begin
    totalSales := 0;
    for month := 1 to 12 do begin
        write('Enter sales of month ', month, ': ');
        readln(monthSales);
        totalSales := totalSales + monthSales
    end;
    writeln('Total sales were ', totalSales)
end.
```

Transformer: A variable that gets its new value always with the same calculation from value(s) of other variable(s).

Technical definition: A variable going through a succession of values depending on, but being different from, one or more most-recent holders, steppers or their followers, and possibly on fixed values (e.g., the kind or type of an input item, or a time given in hours, minutes and seconds converted to milliseconds).

Full program example:

The variable rad is a transformation in the following program:

```
program sine;
const DegToRad = 2.0 * 3.14159 / 360.0;
var i: integer;
    rad: real;
    spaces: integer;
begin
  for i := 0 to 36 do begin
      rad := i * 10.0 * DegToRad;
      spaces := trunc( (sin(rad)+1.0) * 40 );
      writeln(' ' : spaces, '*')
    end
end.
```

One-way flag: A two-valued variable that cannot get its initial value once its value has been changed.

Technical definition: A two-valued variable that can be effectively changed only once (e.g., a variable stating whether the end of input has been encountered) even

though the new value may be re-assigned several times (e.g., a variable initialized to false and set to true each time an error occurs during a long succession of operations).

Full program example:

The variable error is a one-way flag in the following program:

```
program dateValidation;
    var day, month, year: integer;
        error: Boolean;
begin
    write('Enter day: '); readln(day);
    error := (day < 1) or (day > 31);
    write('Enter month: '); readln(month);
    error := error or (month < 1) or (month > 12);
    write('Enter year: '); readln(year);
    case month of
        1,3,5,7,8,10,12: (* ok *) ;
        4,6,9,11: error := error or (day > 30);
        2: if (year mod 400 = 0) or
              ((year mod 4 = 0) and (year mod 100 <> 0))
           then error := error or (day > 29)
           else error := error or (day > 28)
    end;
    if error then writeln('Date incorrect.')
             else writeln('Date correct.')
end.
```

Follower: A variable that gets its values by following another variable.

Technical definition: A variable which, apart from initialization, goes through a succession of values depending on the value of a single variable that is updated immediately after being used for updating the follower, and possibly on fixed values (e.g., the previous pointer when going through a linked list, or the low index in a binary search).

Full program example:

The variable previous is a follower in the following program:

```
program distance;
var month, current, previous, largestDifference: integer;
begin
    write('Enter 1. value: '); readln(previous);
    write('Enter 2. value: '); readln(current);
    largestDifference := current - previous;
```

```
    for month := 3 to 12 do begin
        previous := current;
        write('Enter ', month, '. value: '); readln(current);
        if current - previous > largestDifference
            then largestDifference := current - previous
    end;
    writeln('Largest difference was ', largestDifference)
end.
```

Temporary: A variable holding some value for a very short time only.

Technical definition: A variable holding the value of some other variable or input value for a very short time only (e.g., in a swap operation).

Full program example:

The variable temp is a temporary in the following program:

```
program bubbleSort;
var i, j, temp: integer;
    a: array [1..10] of integer;
begin
    for i := 1 to 10 do begin
        write('Enter number: '); readln(a[i]);
    end;
    for i := 1 to 9 do
        for j := i+1 to 10 do
            if a[i] > a[j] then begin
                temp := a[i];
                a[i] := a[j];
                a[j] := temp
            end;
    for i := 1 to 10 do writeln(a[i])
end.
```

Organizer: An array which is only used for rearranging its elements after initialization.

Informal definition: An array which is only used for rearranging its elements after initialization (e.g., an array used for sorting input values).

Full program example:

The variable a is an organizer in the following program:

```
program bubbleSort;
var i, j, temp: integer;
```

```
    a: array [1..10] of integer;
begin
    for i := 1 to 10 do begin
        write('Enter number: '); readln(a[i]);
    end;
    for i := 1 to 9 do
        for j := i+1 to 10 do
            if a[i] > a[j] then begin
                temp := a[i];
                a[i] := a[j];
                a[j] := temp
            end;
    for i := 1 to 10 do writeln(a[i])
end.
```