Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Teemu Sirkiä

Recognizing Programming Misconceptions

An analysis of the data collected from the UUhistle program simulation tool

Master's Thesis
Espoo, May 21, 2012

Supervisor: Professor Lauri Malmi
Instructor: Lic.sc. (Tech) Juha Sorva

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Teemu Sirkiä

Title:
Recognizing Programming Misconceptions – An analysis of the data collected from the UUhistle program simulation tool

Date: May 21, 2012 **Pages:** vi + 76

Professorship: Software Technology **Code:** T-106

Supervisor: Professor Lauri Malmi

Instructor: Lic.sc. (Tech) Juha Sorva

Learning to program has many challenges. If a student encounters problems of understanding how the programming language works or how a program is executed, it might be hard to fix these misconceptions later. These misconceptions may also make it hard to learn more complicated concepts if the student does not understand the basics correctly.

In program simulation exercises a student takes on the role of the computer as executor of a program using a graphical user interface. In these exercises, students should be able to understand the execution model to simulate the execution correctly and therefore simulation mistakes are interesting because they can reveal what kind of misconceptions students may have.

The main research question of this master's thesis is to analyze log files collected by UUhistle program simulation tool used in the basic programming course and try to find out what the most common simulation errors the students have made are and try to figure out the causes of the errors. Is the error a simple mistake, is it caused by the user interface, is the error related to the simulation exercises or can we find a similar programming misconception in the literature?

We selected common 26 errors which were likely caused by something else than a simple mistake. The errors are related to the basics of the execution, conditions, loops, functions and object-oriented programming. Many of these errors are similar to those misconceptions reported in the literature earlier. Therefore we can make an assumption that visual program simulation makes it possible to recognize possible misconceptions and be used to fix the misconceptions by giving feedback to students.

We also present a few ideas how to improve UUhistle based on the results we got. Some of the errors are clearly related to the user interface and by making some changes to the interface we can reduce the number of the errors caused by the tool itself. We also noticed that the students did not read much the feedback UUhistle gave to them and we should also improve the way UUhistle shows the feedback.

Keywords: computing education, introductory programming, programming misconceptions, UUhistle, visual program simulation, VPS

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Teemu Sirkiä		
Työn nimi:	Ohjelmoinnin virhekäsitysten tunnistaminen – Vislaamo-simulointityökalusta kerätyn tietoaineiston analysointi		
Päiväys:	21. toukokuuta 2012	**Sivumäärä:**	vi + 76
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Lauri Malmi		
Ohjaaja:	TkL Juha Sorva		

Ohjelmoinnin opiskeluun liittyy monia haasteita. Mikäli opiskelijalle syntyy heti alussa virheellisiä käsityksiä ohjelmointikielestä tai ohjelman ajonaikaisesta toiminnasta, näiden virheiden korjaaminen myöhemmin voi olla haastavaa, ja ne hankaloittavat myöhempien asioiden oppimista.

Visuaalisissa ohjelmasimulaatiotehtävissä opiskelija ottaa tietokoneen roolin ja suorittaa hänelle annettua ohjelmaa graafisessa käyttöliittymässä simuloiden suoritukseen liittyviä vaiheita. Koska opiskelijan täytyy itse simuloida ohjelman suoritusta, tämänkaltaiset tehtävät voivat paljastaa, millaisia virhekäsityksiä opiskelijoilla on.

Työn päätavoitteena on tutkia ohjelmoinnin peruskurssilla käytetyn Vislaamo-ohjelmasimulaatiotyökalun keräämistä lokitiedoista, minkälaisia yleisiä virheitä opiskelijat ovat simulaatiotehtävissä tehneet ja pyrkiä päättelemään, mistä virhe on voinut aiheutua. Onko virhe puhdas vahinko, onko Vislaamon käyttöliittymä voinut vaikuttaa sen syntymiseen, johtuuko virhe simulaatiotehtävien luonteesta vai voisiko virheeseen liittyä jokin kirjallisuudessa tunnettu ohjelmoinnin virhekäsitys?

Kerätyistä lokitiedoista valittiin 26 yleistä virhettä, joiden taustalla vaikutti olevan jotakin muuta kuin vahingossa tehty virheellinen askel. Virheet liittyvät ohjelman suorituksen perusteisiin, ehtolauseisiin, silmukoihin, funktioihin ja olio-ohjelmointiin. Näistä virheistä moni on hyvin samankaltainen kuin aiemmin kirjallisuudessa esitetyt virhekäsitykset, mikä tukee olettamusta, että ohjelmasimulaation avulla on mahdollista havaita mahdollisia virhekäsityksiä ja yrittää korjata niitä antamalla mahdollisimman hyvää ja täsmällistä palautetta opiskelijalle.

Työssä esitellään myös muutamia ideoita, joiden avulla Vislaamo-työkalua voidaan kehittää nykyistä paremmaksi tässä työssä tehtyjen havaintojen perusteella. Muutama virheistä johtuu selkeimmin käyttöliittymästä, jota kehittämällä näiden virheiden osuus saadaan todennäköisesti pienemmäksi. Toisaalta havaittiin, että opiskelijat lukevat hyvin vähän työkalun antamaa palautetta, joten sen tuomista paremmin esiin täytyy myös kehittää.

Asiasanat:	tietotekniikan opetus, ohjelmoinnin alkeet, ohjelmoinnin virhekäsitykset, Vislaamo, ohjelmasimulaatio, VPS
Kieli:	englanti

Preface

My work with the UUhistle project started quite exactly three years ago. I saw a work announcement where Juha Sorva was searching for a person to implement a prototype of a new program simulation tool based on Juha's idea. I went to the interview and meanwhile I was driving back home Juha had sent me an email that I will get the job.

At that time, I did not know exactly what I was going to implement and what we were going to achieve but I had promised to do my best. Together with Juha we brainstormed ideas, tried out many different things and gradually UUhistle started to become a simulation tool that could really work.

The work has been challenging but I have enjoyed the coding and working with Juha. It has also been a privilege to work with LeTech group and see the tool being used in the basic programming courses here in Aalto University.

The process to write this master's thesis began in the fall 2011. I want to thank both my supervisor professor Lauri Malmi and instructor Lic.Sc Juha Sorva for their invaluable guidance and feedback throughout the writing process.

I am grateful to my parents and grandparents who have always supported me with my studies.

I want also to thank my work colleagues Teemu Koskinen and Petri Ihantola sitting in the same room with me and helping me with the questions I have had. And of course, I want to thank all the students who have used UUhistle. Without you it would have been impossible to write this thesis.

Espoo, May 21, 2012

Teemu Sirkiä

Contents

Chapter 1

Introduction

Learning a new thing is always a long process before mastering the topic well. Making errors is a part of the process and we use to learn from our mistakes. But if you are learning on your own or you do not get enough guidance, how do you know that you might have understood something incorrectly?

1.1 Difficulties in programming

Learning programming is no exception. To become a good programmer, it requires a lot of learning, understanding and especially coding. The problem is that if you don't understand some basic concepts correctly, it might be very hard or impossible to continue to learn because the computer won't behave at all like you have thought. At this point, how to continue the studies if the behavior seems to be totally unexpected? That is why it is important to recognize different misconceptions as early as possible and try to fix them.

Usually most of the misconceptions are related to things that happen during the execution and there is no clear way to see what is going on under the hood. A novice programmer may know something about variables or the concept of parameter passing but how to learn these things correctly if you can not see them?

1.2 Program visualization and simulation

Program visualization tries to help in understanding the concepts and the different phases in the program execution by showing them visually. This kind of visu-

alization has been used when teaching algorithms and it seems to be an efficient way to improve the understanding and learn better.

Although program visualization is a good way to make abstract things visible, the problem is that only by looking at an animation, you may not notice that animation proceeds differently than you originally thought.

Visual program simulation is a concept that is a form of program visualization which encourages students to think and be active instead of sitting passively in front of the screen and watching an animation. Sorva gives the following definition: ”In visual program simulation, a learner takes on the role of the computer as executor of a program” [29]

When novice programmers are doing VPS exercises, they have to really think how the execution proceeds and what the next step is because they have to simulate it by themselves. Active thinking means that you have to understand what you are doing and why. If your idea is wrong, you will see that the idea is not going to work and you have to think what is wrong with that.

VPS tools can detect possible misconceptions automatically by some predefined rules and give immediate feedback to students. These tools also make it possible to log the simulation steps and collect the data making the logs to a great source to find common mistakes and other interesting data about how the tools are used. The main focus of this thesis is to analyze the logs collected with one VPS tool, UUhistle, and see what kind of information can be extracted from the log files.

1.3 Structure of this thesis

The thesis discusses first in the second chapter programming misconceptions in general based on the literature. Then in the third chapter UUhistle program simulation system is presented to understand better the collected data. UUhistle is also compared with a few similar existing systems. After that the objectives of this thesis are discussed in the fourth chapter in more detail. The fifth chapter is about the collected data and how it is analyzed. The results are presented in the sixth chapter. In the seventh chapter, we discuss the errors we found and how those errors are related to the misconceptions reported in the literature. We also discuss how to improve the learning experience based on the results we have. In the last chapter, we make a conclusion and a short summary of the results.

Chapter 2

Programming misconceptions

In this chapter we discuss the programming misconceptions in general: what kind of misconceptions exist, where do they come from and how to avoid them. This chapter mainly focuses on the literature of this topic.

2.1 Background

A misconception means that you believe you know how something works but in reality the model is wrong. The less you know about the topic, the more severe the misconception can be. If the subject is somehow familiar, it may be that only the irrelevant details are not correct, but when you do not have any previous knowledge or just a little, your way of thinking and understanding the concept can be totally incorrect.

Programming misconceptions are mostly related to the knowledge how the program code is executed and how the different concepts, for example, variables, control structures and objects are linked to the execution model.

Programming misconceptions have been researched since the 1970s but the most active era was in the 1980s. At that time the psychological background of programming and the understanding of procedural programs written in BASIC and Pascal, for example, were under active research. The research gained more momentum again in the late 1990s and at the beginning of the 2000s when object-oriented programming started to become popular and many basic programming courses switched to object-oriented programming languages such as Java and C++.

2.2 Types of the misconceptions

Novice programmers may have many misconceptions related to many different topics. The list of the reported misconceptions is long and in this thesis it is not possible to cover them all. However, in this section we give a brief overview of the main categories and what kind of misconceptions exist in those categories.

One of the main issues is to understand the program execution. The code lines are executed sequentially and the behavior of the program depends on the executed lines, not the upcoming ones. Pea [19] states that some students believe that all the lines in the program are alive at once. Pea also constructed the term *superbug* which means computers have some kind of intelligent powers to know what will happen in the lines that are not currently executed.

Du Boulay [4] discovered that many students have problems with variables and assignments. Students do not understand that variables can hold only one value at a time and it is important to write assignment statements in the right order. Du Boulay also noticed that some students believe that after assigning a variable to another, the variables are somehow linked together which means changing the value of one variable would also change the value of the second variable.

One type of the misconceptions is related to the program flow. Du Boulay made similar observations as Pea that novice programmers can not always understand that the next instruction is always executed if the program does not instruct otherwise. Do Boulay states that some novices believe instructions are saved to a buffer and then executed all at once when the program ends. Loops may also cause problems because some students may believe that the condition of `while` loop is constantly checked. The variable of `for` loop is also updated behind the scenes which can make it hard to understand. Conditionals were also problematic for students analyzing BASIC code in the research of Bayman and Mayer [1]. The students did not understand how `IF` statements control the program flow especially when `GOTO` instructions were used together with `IF` statements.

Functions and function calls have difficult concepts to understand. Parameter passing, meaning of the return value and variable scope require a lot of thinking. Misconceptions related especially to parameter passing have been researched by Madison and Gifford [14] and Fleury [5]. It is hard to understand what it means when a parameter is passed to a function. Parameters are fetched from the caller's scope and the value is passed to callee's scope. In this process the same value gets a different name and the callee can not access the caller's variables. Pascal language was used in previous articles and Pascal, as well as many other languages, can also handle parameters as references which is far more confusing.

Object-oriented programming can be seen as an extension to procedural programming. Therefore it suffers from the same misconceptions as the procedural programs but the object-oriented concepts bring a completely new set of misconceptions. Classes, objects, methods and references are important to understand but if the object-oriented world is not clear, it might be impossible to construct a clear vision how object-oriented programming works and how to use it efficiently. Holland et al. [8] lists a few problems: objects are just wrappers for variables or simple records, methods are mainly assignments and it is difficult to make difference between classes and objects. Sanders and Thomas [24] point out similar results. They also discuss linking and interaction problems and problems with understanding inheritance.

2.3 Sources of the misconceptions

All novice programmers have their own misconceptions and the sources of the specific misconceptions vary but in the literature there are many suggested ways how misconceptions arise.

The syntax of the programming language might be thought to be a clear source. If the syntax is difficult to understand, it may lead to a situation where it is hard to understand how the code is executed. However, this claim may not hold. Sheil [25] has already in 1981 concluded that the notation is not a major factor in the difficulty of programming. Spohrer and Soloway [32] and Kaczmarczyk et al. [10] got similar results in their research.

If the programming language is not the problem, what could be? Du Boulay [4] suggests that the analogy with English can be a problem. The keywords and commands in the programming language are chosen to be similar to the spoken language although the semantics can be totally different. For example, in the spoken language it is clear that `while` means something is being done until something happens and when it happens the execution will be stopped at once. However, in the programming languages this does not hold. The condition is checked only when one iteration is done and the next iteration is about to begin. If a novice programmer does not know this beforehand, the misconception can arise and stay until it is somehow proven to the student that the loop does not work in that way.

Natural language is also a problem according to research made by Bonar and Soloway [2]. They have stated that the preprogramming knowledge is a major source of bugs. This means that the similar structures in natural and programming

language cause difficulties because the structures do not work in the same way as the novice programmer may expect. Bonar and Soloway also bring the concept of fragments in the programming knowledge. If a novice does not have enough knowledge to create the program he is working with, the novice will fill the gap with a guess which is likely to be incorrect.

Fleury [6] researched programming in Java and noticed that the students create their own rules based on the current knowledge and they combine it with new experiences. The students told the programs they have seen are the most important source of their knowledge. Lectures were a close second and textual materials a distant third. Fleury cites that because students construct their own meanings, it is not surprising that students will not create a complete model although complete and accurate information is given to them.

2.4 Preventing and correcting misconceptions

A key issue is to get a student to realize that the concept he has built is somehow wrong. The best way to do this is to show a concrete model which breaks the mental model of the student. Mayer [15] has discovered that a concrete model can have a strong effect on the use of new technical information. Mayer also suggests that students should be able to explain the statements in the program code. This helps students to use the previous knowledge and use that to comprehend the new material.

As Fleury [6] discovered, the given program examples are important because novice programmers get their knowledge from them. Therefore it is important to select examples so that they break the usual misconceptions. Holland et al. [8] have listed a number of misconceptions related to object-oriented programming and what kind of examples should be used to avoid those misconceptions. Sanders and Thomas [24] have created a similar checklist what kind of pieces should be found in the code and what kind of understanding that piece represents. For example, if return values and references are used, this indicates the understanding of linking and message passing. They have also constructed a checklist which helps to notice if a student may have a misconception. For example, if a class contains only getters and setters, the student may think objects are just simple data records.

Chapter 3

The UUhistle program simulation tool

In this chapter we present UUhistle program simulation tool. The chapter covers aspects like how the tool is used in general and what kind of exercises it supports. The purpose is not to give a complete view of the features but enough background that is needed to understand the chapters related to the data analysis and the results. This chapter also contains a brief review of the other similar tools.

3.1 General overview

UUhistle (pronounced as *whistle* and *Vislaamo* in Finnish) is a program visualization and also a program simulation tool [29], [33]. UUhistle is based on the original idea of Juha Sorva and the tool is being developed since May 2009. The author of this thesis is responsible for the coding of the tool.

UUhistle is designed to be used in introductory programming courses where Python language is used. It provides an easy way to create program examples and show the execution as step-by-step animations. In addition to that, UUhistle also features a new form of exercises: visual program simulation exercises where a student takes the role of the computer and simulates the execution by dragging, dropping and clicking the elements with the mouse manually.

UUhistle has been used in Aalto University in CS1 and Data Algorithms and Structures courses since spring 2010. Currently over 2500 students have used the tool. UUhistle is integrated into Goblin [7] and TRAKLA2 [12] online assessment systems where the students also find the other exercises that are part of the course they are participating. UUhistle is an applet that opens a new window on

the top of the browser but it can also be used as a standalone application in the lectures, for example.

3.2 User interface

The main idea of UUhistle's user interface is to show the important parts of the computer memory as abstract graphics in a novice friendly way. These parts include the heap, the stack and the stack frames. When using a normal debugger, only the call stack and local variables are normally visible but they are shown in a way that is not very easy to understand if you are not familiar with debuggers. UUhistle also visualizes all the steps inside a code line compared with the debuggers which usually handle one line of code as an atomic operation.

3.2.1 Original user interface

The original user interface is shown in the Figure 3.1. It was used in the spring 2010 when the tool was used for the first time. In the picture there is a small object-oriented program running. The executed code is on the left where the blue arrow points at the current line, the heap is on the top showing the literals and objects in the memory, class definitions, functions and operators are on the right and the stack is in the middle. In the stack there are two stack frames containing their local variables and the evaluation areas which are used to evaluate the statements.

Orange boxes are operators, purple functions or methods, green variables, lighter blue values and darker blue boxes are classes and class instances. The dark orange box in the bottom frame indicates a method call which is currently active in the frame above.

Students can control the animation with the control buttons located on the left below the code area. When a student is doing a program visualization exercise by watching the animation, it is possible to move backward and forward during the animation and see some interesting steps again if needed. Also when a student is doing a program simulation exercise and makes a mistake, after the error dialog the student can undo the previous step or steps and think what went wrong and how the simulation should proceed.

The status bar in the bottom shows after each animation step what has just happened and how to continue. In the simulation mode UUhistle just asks the user to do the next step.

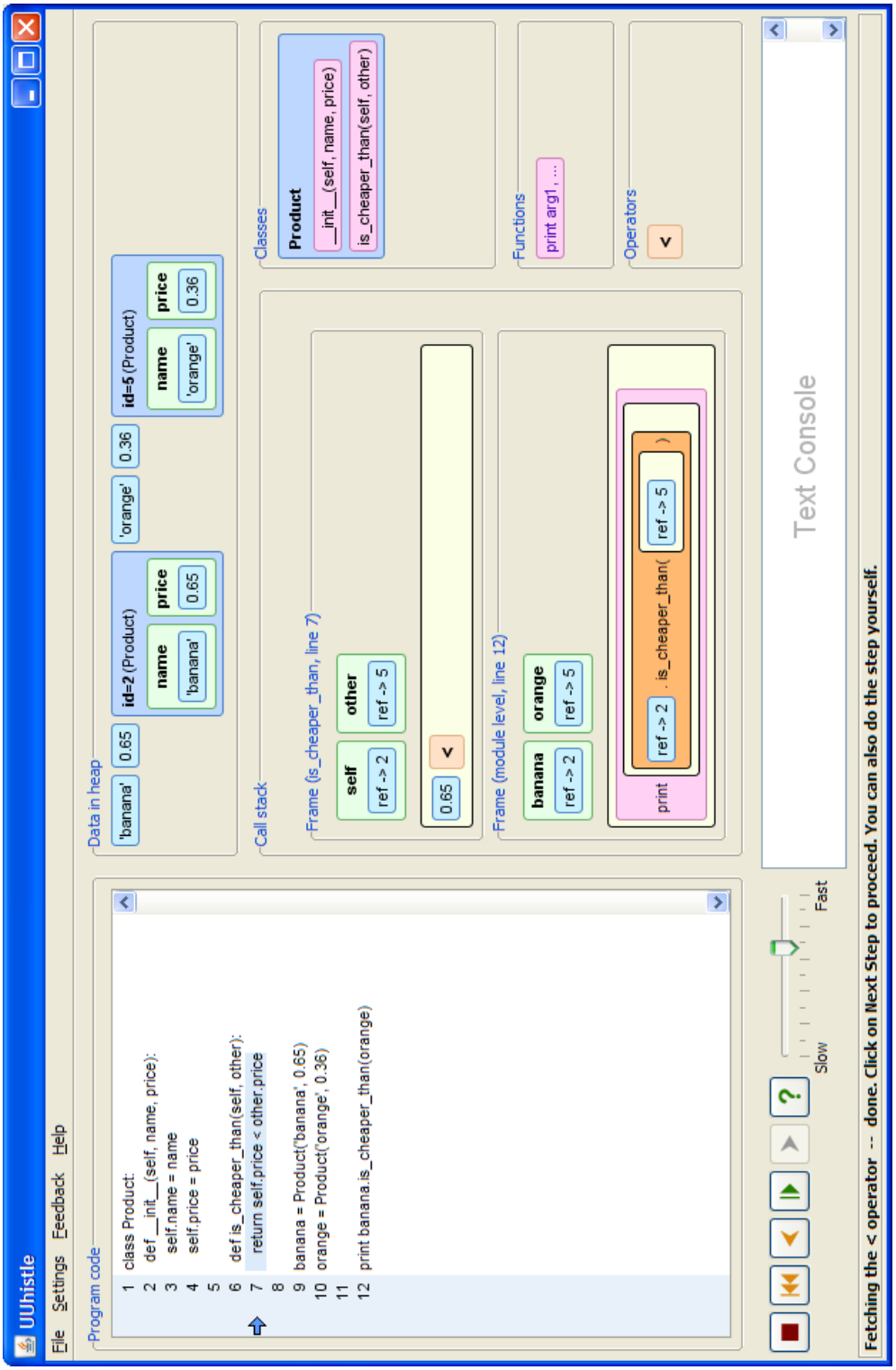

Figure 3.1: The original UUhistle user interface. UUhistle is used in the program visualization mode.

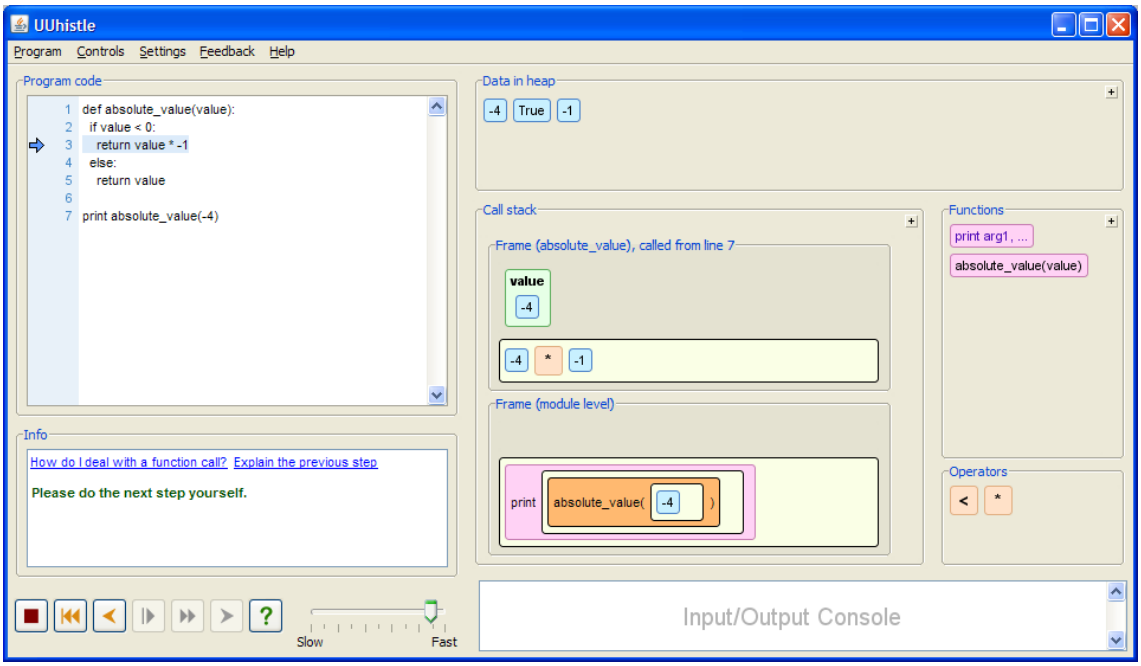

Figure 3.2: The improved UUhistle user interface. The new status area is on the left below the code area. UUhistle is used in the visual program simulation mode in this picture.

3.2.2 Improved user interface

The problem with the original user interface was mainly the status line. It showed important information but the place and the format was not perfect. Therefore in the next version a new area was added to show the current status and other relevant information. The new area is shown in the Figure 3.2. This new version was released after the course ended in the spring 2010.

In addition, to give a better overview what has happened and how to proceed, the status area also gives a possibility to offer more help. After each step (made by computer or student) UUhistle shows a link *Explain the previous step* and if a student clicks the link, UUhistle shows a textual description of the previous step. An example is shown in the Figure 3.3.

In some situations, UUhistle also shows links to help dialogs which explains how to simulate a function call in UUhistle, for example. A more detailed description of the user interface is in [30].

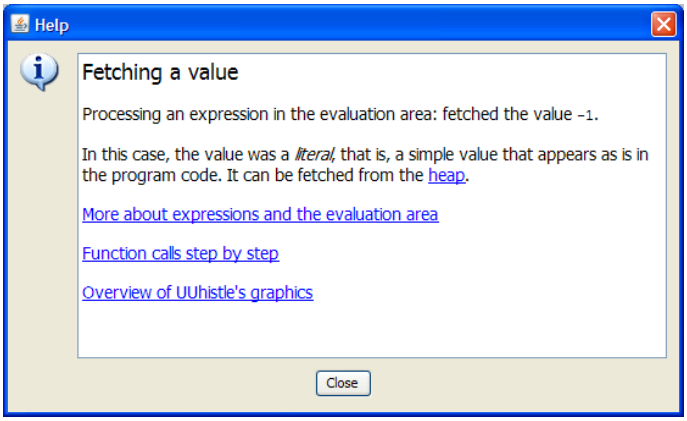

Figure 3.3: UUhistle shows an explanation of the previous step. There can also be links to another explanations that are related to the topic.

3.3 UUhistle exercises

As already mentioned, there are two main categories of the exercises. The first category contain traditional program visualization exercises where students only watch the animated program execution. In this mode students are not required to tell how the execution proceeds and the main goal is to understand the concept being trained and see how UUhistle visualizes the steps. Teachers may add questions or pop-up dialogs to emphasize important things but these do not affect the execution.

The second category contains visual program simulation exercises where students have to do the steps by themselves in the same manner as the computer did in the previous examples. The exercises are normally arranged so that there is first a program visualization exercise and after that a similar program simulation exercise. In program simulation exercises students are encouraged to think what happens next in order to be able to continue with the exercise. This kind of exercises let students think the details and simulation may also reveal misconceptions if the execution does not proceed as the student thought.

The newer version of UUhistle also tries to recognize misconceptions by some predefined rules. In this case it can show a dialog explaining what was wrong with that step and why. A more detailed description of this functionality is in [31].

The program simulation exercises are done by dragging and dropping the values. Normally in VPS exercises UUhistle controls the program flow (except in branches where students have to select the next line manually) and students sim-

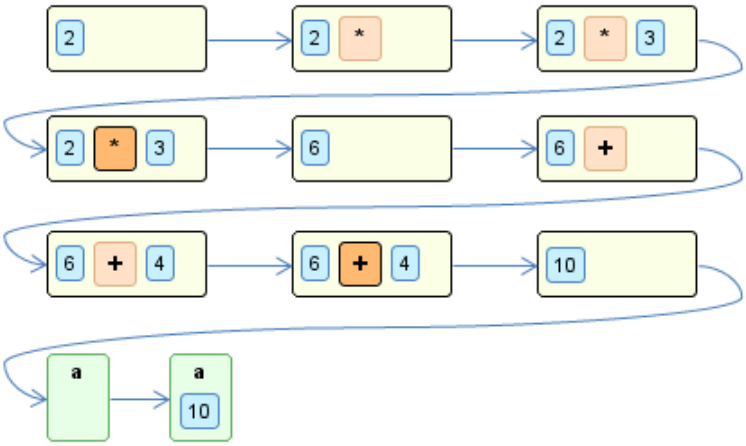

Figure 3.4: The simulation steps of the code line `a = 2 * 3 + 4`. The dark orange operators mean that the student clicks the operator and after that the expression is replaced with the result. The result is shown in the next step which is not an actual simulation step.

ulate the other steps. It is also possible to create so called hybrid exercises where students do some of the steps and UUhistle does the rest automatically. In this kind of exercises it is easy to train only some particular things avoiding the exercise to become too long and complex.

A concrete example of the simulation is presented next. The code line in this example is a simple assignment statement: `a = 2 * 3 + 4`. The simulation of this line in UUhistle used in 2010 consists of nine simulation steps. The steps are presented visually in the Figure 3.4.

The student should evaluate the right-hand side of the assignment first. This is done by dragging the needed values from the heap and the operators from the operator panel so that the expression is constructed to the evaluation area as it is in the code. However, the important part is that UUhistle requires the multiplication is executed before dragging the addition operator, as the execution proceeds in reality. In this way students learn in which order the more complex expressions are evaluated after they have first understood the basic idea. After the right-hand side is evaluated and the value `10` is in the evaluation area, the user creates the variable inside the stack frame and moves the result from the evaluation area to the variable.

More examples of different simulation steps are presented in Chapter 6 which discusses the errors the students have made. More information about the tool and also the tool can be downloaded from the web site *http://uuhistle.org*.

3.4 Related systems

There are many program visualization tools but only few where students can somehow interact with or control the execution. Here is a short presentation of some existing tools having the same kind of features as in UUhistle.

3.4.1 ViLLE: Clouds and boxes

ViLLE is originally a Java applet based language-independent program visualization system [22]. The current version discussed here uses modern web technologies and runs in the browser without any plugins [34]. This version also features a new kind of exercises, called *Clouds and boxes*, where the student must simulate the execution of the given small programs written in Java. The user interface of is presented in Figure 3.5. ViLLE has also other exercise types but in this section we discuss only the Clouds and boxes exercises.

The main idea of these exercises is to see a line of code as one atomic step. For example, an assignment such as `int b = a + 2` is done by creating a new integer variable and assigning the correct value to the variable at once. There are no intermediate steps to calculate the new value, for instance. If there is a need to change the value assigned to a variable, the new value is just changed to the text field presenting the current value. When on line of code is completed, the student clicks Next Step button and the execution jumps to the next line. There are also modes where the current line is not visible or the student must always explicitly select the next line instead of an automatic jump.

ViLLE supports function calls (or actually static method calls in this context because the exercises are in Java) defined in the simulated code but it does not support classes. If the code line contains, for example, string manipulations with `substring` method, the user needs to know what the return value of that method located in Java's standard library is. The method call itself is totally invisible to the student.

There is no contextual help or immediate feedback while doing the exercise but student gets the grade after submitting the exercise. The exercise can be submitted at any time but after getting the grade the student can not anymore proceed the execution without resetting the exercise and starting again from the beginning. If the student does not get the full points, the tool will not give any feedback about what went wrong.

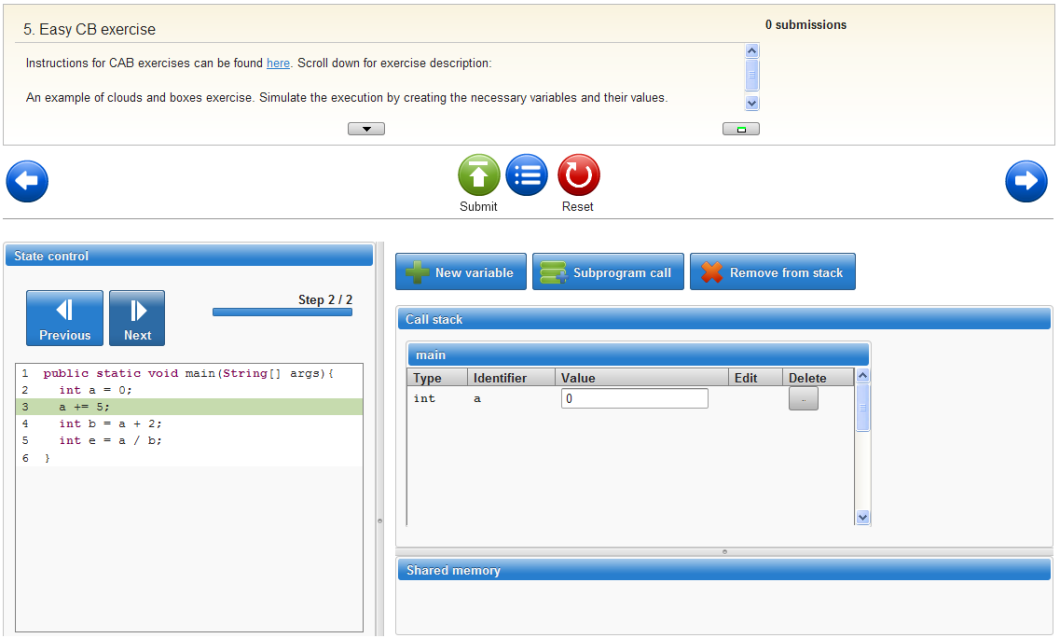

Figure 3.5: A screenshot of ViLLE Clouds and boxes exercise running inside a browser.

3.4.2 Online Tutoring System

Online Tutoring System is a Java applet that lets students to simulate the execution of the given code [11]. The code examples are written in Visual Basic but the tool itself is language independent.

The tool shows the code and the student must click always the next line. If the line contains a simple assignment, the tool asks which value will be assigned to this variable. More complex statements are evaluated in a separate view, where the student always clicks the next part to be evaluated and evaluates the statement in the correct order. Two views of the tool are presented in Figure 3.6.

The exercises contain basic operators, branches, loops, type conversion functions and some self defined functions but are no classes or object instances. Function calls are simulated in a separated but similar view as the other code but the stack is not visualized at all.

The tool gives immediate feedback if the step is not correct. If the student was asked to write a result and it was incorrect, the tool asks to repeat the step and gives the correct answer. If the student should select the part to be evaluated next or the next line and the selection was incorrect, the tool states the choice was

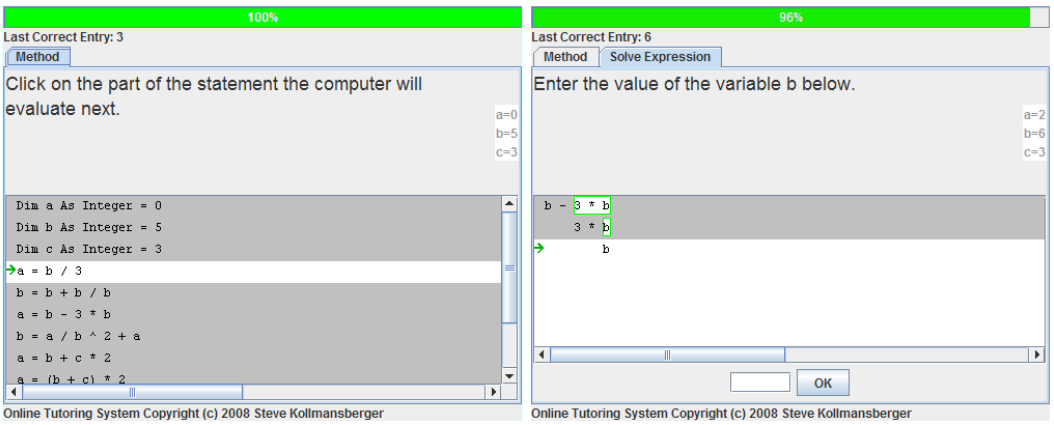

Figure 3.6: Two views from Online Tutoring System [18]. On the left there is an example program running and the student should select the next statement to be evaluated. On the right there is a view how the given statement is evaluated stepwise.

incorrect but does not reveal the correct answer. There are no explanation texts or a possibility to undo and redo the actions. Of course there is no need to undo because the simulation will not go further before the step is correct but if the student would like to do some steps again to train that, the only way is to reset the exercise and start again.

3.4.3 The tool by Donmez and Inceoglu

This unnamed tool is made with Flash and it is used to let students simulate small C# programs [3]. It supports a small subset of the language: integers, floats, booleans, strings, basic operators, console and type conversion functions and `if` and `while` statements.

The features allow simple exercises covering basic arithmetics, variables, type conversions and simple I/O. The statements are evaluated in so called *active area* which is seen in Figure 3.7. The statements are evaluated stepwise and the functions can be called by selecting the right function from the drop-down list. If the result is stored to a variable, a new variable can be created to the variable area and then assign the value to the variable by selecting the variable name from the drop-down list.

The tool does not show the stack or support self-defined function calls. There is an area called *help topics area* but the description of the tool does not discuss in more detail about what kind of help the tool shows.

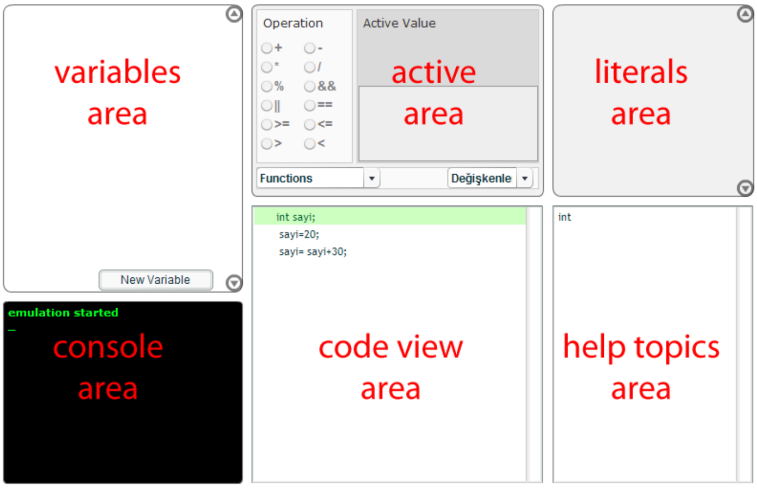

Figure 3.7: The interface of the tool made by Donmez and Inceoglu as presented in [3]. The view consists of six areas.

3.4.4 Other systems

There are two systems that do not fall in the category of visual program simulation but they are in other ways related to UUhistle.

One of the existing systems that has influenced on UUhistle is Jeliot 3 program visualization tool and its previous versions [16]. Jeliot 3 shows the execution of Java programs as abstract animations. Jeliot 3 supports almost all Java language features and is able to show them visually just by typing the code. Jeliot 3 can automatically create questions related to the code being executed [17] but it does not have any simulation features.

TRAKLA2 is an online environment containing exercises related to data structures and algorithms [12]. In the exercises data structures are presented visually and the algorithm itself as pseudo-code. Students manipulate the data structure by dragging and dropping the elements in the order the algorithm proceeds. After the exercise is done the student gets the points and can compare the submission to the model solution. If compared with UUhistle, the abstraction level in TRAKLA2 is much higher.

3.5 UUhistle compared with the other systems

UUhistle can be seen as a combination of the systems which were previously presented. The implementation is different but the ideas behind UUhistle contain many common elements. However, UUhistle is the only tool that supports both the program simulation and visualization exercises. The comparison below is mainly between the three presented tools and TRAKLA2 and Jeliot 3 are not included in this comparison if not mentioned otherwise.

One of the main differences between the systems is the support of the language features. All the three presented systems support only a very limited set of features, mainly variables, basic arithmetics, branching and looping. In the two systems, function calls can be somehow simulated but because the call stack is not visible, the parameter passing and return value are not clearly shown although these are important steps. In Jeliot 3 these steps are shown but the call history is in a separate tab. UUhistle always shows the call stack if there are function or method calls in the program code and therefore these concepts should become familiar. This makes it also possible to demonstrate, for example, how recursion works in addition to the normal function calls.

The tools support the basic arithmetics and control flow structures which cover the basics but nowadays object-oriented programming is very popular and used in many CS1 courses. Object-oriented world contains many abstract things such as classes, objects and references. None of the three tools can handle these. ViLLE can show strings as references but these tools do not have exercises where classes and objects would be used. Because these concepts are vital to understand, UUhistle supports defining own classes and methods. New objects can be created and then seen as references in the local variables. This makes it easier to understand what classes and objects are and what a reference to an object means. Jeliot 3 also supports classes, objects and references but shows them in a slightly different way.

While doing the exercises, sometimes it might be a good idea to go one step back and see or do something again. Unfortunately, ViLLE is the only tool which allows moving backwards without resetting the whole exercise. In UUhistle students can always *undo* and *redo* the steps if needed. Especially the undo feature is important because UUhistle allows students to make mistakes and even continue after the wrong step although usually students undo the incorrect step straight after the error message.

It is also important to give explanations what a specific step in the animation means or what was the purpose of the step the student just made. Therefore

UUhistle automatically produces a textual description of the each step as explained earlier. Only the tool by Donmez and Inceoglu seems to have some kind of explanations but the feature is not described in the article. Although Jeliot 3 shows detailed animations, it relies also only on students' skills to understand the steps without any textual explanations.

Summing up, UUhistle is a more sophisticated tool for VPS exercises than the few other tools supporting the idea of letting students to act as computer. UUhistle can provide richer exercises by supporting more language features, allowing to move backward and forward during the execution, giving textual feedback and generating explanation texts of the visualized or simulated steps.

Chapter 4

Objectives

Computer-based exercises can collect data containing information such as how a student has solved the exercise, how much time it took, what kind of mistakes the student made etc. If there are many students using the tools, the number of the log files can be very high. The more the data is available the more likely it is to find something interesting in those files.

Because UUhistle collects this kind of logs and the tool has been used three years, the logs might reveal interesting information about the usage of the tool. Our research questions are related to this information.

4.1 Common errors

Can we recognize the common errors the students have made?

The log files contain a sequence how a student solved the simulation exercise. This sequence also contains all the incorrect steps and makes it therefore possible to track what kind of mistakes the students have made.

One of the objectives is to create a tool which can analyze the log files and show at each step what kind of steps the students have made at that point. By looking at that data we can see if there are steps where many students have made the same mistake. This analysis shows also if the data is useful for this kind of purposes.

4.2 Reasons for errors

What have caused the errors? Is the user interface unclear or could there be a misconception? Can we find any new misconceptions?

After recognizing the common errors, one of the objectives is to figure out if the errors are caused by a simple mistake, the user interface, the nature of the VPS exercises or can we show that we have discovered similar errors that have been reported as misconceptions in the literature. We would also like to see if the visual program simulation arises new kind of possible misconceptions that are not yet reported.

4.3 Creating better VPS exercises

How can we make better VPS exercises and how to improve UUhistle?

The collected data may also help to create better VPS exercises in the future. If we notice that some errors are caused by UUhistle's user interface, we can improve it and in that way help students.

In case we find some common errors that can be easily recognized and there can be a possible misconception, we can give more precise feedback in those situations.

Chapter 5

Data analysis

In this chapter we discuss what kind of data UUhistle has collected, how it has been analyzed and what kind of challenges we had related to the analysis.

5.1 General description of the collected data

While solving the exercises, UUhistle collects all the visualization or simulation steps to a log. This means that the steps made by students as well as the computer are logged. In addition to the simulation steps, UUhistle also logs stepping backward or forward, asking help and other kind of activities.

The log of each submission will be saved to the server when the student submits the exercise. If the student decides not to submit the exercise, in this case the log will be lost.

Normally, students solve the exercises in a way that they undo or start over the exercises and continue in this way until the exercise is solved. Although a student restarts the exercise, the log data is log data is still available if the tool itself is not closed. Therefore the logs should cover the usage of the tool well.

5.2 The log files

The log is a simple textual representation of the steps where one line corresponds to one simulation step. A log entry begins with a timestamp which tells how many

seconds after the beginning of the exercise the step was made. After the timestamp there is the name of the action or `ERROR` if the previous step was incorrect. If the action was a real simulation step, then there are also value and position information and the executor of the step which can be `user` or `computer`. The actual meaning of value and position information depends on the action but normally they contain the element and a position where the value or the operator was placed in the evaluation area, for example.

The following example shows a piece of a log where the code line should be evaluated in the similar way as in Figure 3.4. The code line in this example is:
`fahrenheit = 1.8 * 100 + 32`

```
191: jump  @ 2 (computer)
211: add_value 1.8 @ 0 (user)
224: add_operator * @ 1 (user)
233: add_value 100 @ 2 (user)
235: add_operator + @ 3 (user)
235: ERROR: You didn't perform the right type of step.
238: UNDO
253: operator * @ 1 (user)
261: add_operator + @ 1 (user)
264: add_value 32 @ 2 (user)
270: operator + @ 1 (user)
276: create_var fahrenheit (user)
281: assign fahrenheit (user)
```

The example starts with a jump to the line 2 which was made automatically 191 seconds after the beginning of the exercise. After that the student has dragged the value `1.8` to the evaluation area. At that point, the evaluation area was empty, because the position is zero which means that the value is the first element in the evaluation area and it is not inside any other element, for example as a parameter in a function call. Then the student has dragged the multiplication operator and the value `100` after each other to the evaluation area.

In the next step, the student should click the multiplication operator but the student has dragged the addition operator to the evaluation area which has caused an error. After the error, the student moved one step backward and evaluated the multiplication operator as expected. The rest of the log contains dragging the other elements, evaluating the operator, creating a new variable `fahrenheit` and assigning the result to that.

The original purpose of the logs was to create more precise bug reports. When a bug is reported, the bug report automatically contains the log to help reproducing

the bug. In many cases to reproduce a reported bug, the sequence of steps must be done exactly in the same order as the student did or otherwise the bug can not be reproduced. The quality of bug reports increased a lot when the log was attached to the bug report with the student's own explanation.

Because UUhistle was already capable of collecting logs, we got an idea to include the log also in the submission. Before this master's thesis, the log files were analyzed just to see some statistics such as the average time to complete an exercise or what is an average number of the errors made in a specific exercise etc. but no further analysis had been done.

5.3 The analysis

The main idea of the analysis is to go through the log files and find the most problematic steps. Because the log files contain only the steps the student have done, the log file must be compared with the model solution in order to find out the expected step when an error has occurred.

The analysis is based on an algorithm which can be seen as some kind of state machine which gets two inputs: the model solution of the exercise being analyzed and the log files one at a time. The state machine reads the log file line by line and after reading each line, the state machine compares if the step is correct. If the step is correct, the pointer that points to the current correct step is increased. If the step is `undo` or `redo`, the pointer must be decreased or increased accordingly in order to track the execution correctly.

If the state machine recognizes an incorrect step, the incorrect step will be saved. The model solution is transformed into a data structure where each step contain a list where all the incorrect steps done at that step will be saved. Therefore after all the log files have been analyzed, it is possible to count how many wrong steps have been made at each step and what kind of steps have been tried. If the there are steps where many students have done the same wrong step, this might indicate a misconception or another kind of a problem.

Normally, students undo immediately after an error, but if the student continued the execution after the mistake, only the first incorrect step would be saved and the following steps are ignored. The tracking continues again after the student moves backward until he reaches the last point where the execution was correct and does the next step correctly.

The analysis tool was written in Python and it produced its output as plain text

that could be then opened in Excel for further analysis.

When an exercise is analyzed, the results are printed out as plain text. One simulation step is printed out as one line as in the following example:

```
add_function | 366 85 add_function 0 float | 56 13 add_function 0 raw_input
```

This shows that the correct step was to drag a function call to the evaluation area. The details of the correct step are not included because it is easy to trace the execution by looking at the exercise and usually the wrong steps are not partially correct as in this example. In this example, at this point there have been two common simulation steps. 366 students (85 %) have dragged `float` function to the evaluation area where `0` means that the function call is currently the only element in the evaluation area because the position is zero. Another common step has been to drag `raw_input` function the evaluation area with a share of 13 %. At this point, the correct step was to drag `float` function and therefore 85 % of the students made the correct simulation step. The steps with a very low percentage are not included because the list would otherwise be very long and contain some steps that only a few students have tried. We are interested only the mistakes that many students have made.

5.4 Challenges with the analysis

Although there is only one way to solve a simulation exercise, the problem is that in some situations UUhistle allows so called *shortcuts*. For example, if the line to be simulated is a simple assignment as `a = 3`, UUhistle will not require to drag the value to the evaluation area but allows to create the variable and then drag the value to the variable. If the shortcut is used, `add_value` event is not logged although it exists in the model solution. The longer way can also be used so the algorithm must recognize these shortcuts or otherwise the model solution and the log file do not match each other anymore.

Therefore, if there is a sequence of steps in the model solution where a shortcut is possible, the state machine recognizes that and checks if the student has used the shortcut. If the shortcut is used, then the pointer to the correct step in the model solution is increased twice in order to skip the shortcut step which does not exist in the log.

The exercises also contain steps where the computer makes one or more steps automatically after the student has made a step. This kind of situation occurs, for example, at the end of the line when the execution jumps automatically to the next

line. There are also so called hybrid exercises where UUhistle adds literal values automatically to the evaluation area, for example. If a student clicks Undo button after this kind of situation, the simulation jumps back to the state where it was before the last step made by the student. This means the simulation can move several steps backwards and the pointer pointing to the correct step in the model solution must be changed accordingly.

Another problem is that in the older versions there were bugs that cause problems in the analysis. After undoing a step the simulation might have moved two steps backwards. Some steps were also not logged at all if the step caused internal errors in UUhistle. The state machine can recognize some of these situations automatically and insert the missing steps or move two steps backwards but this is not always possible which leads to a situation where the analysis fails and the file must be fixed manually.

If the analysis failed because of an incorrect log file, the log file is automatically opened to locate manually the reason. The tool shows at which step the synchronization with the model solution failed and the incorrect log entry is usually near that step. In the most cases, the file can be corrected by adding or removing one `UNDO` line or removing a duplicated step. These modifications are necessary to a small number of the log files especially collected with the older versions of UUhistle.

All these aspects together do the analysis not that straightforward as one could expect. The analysis tool must be able to emulate UUhistle's behavior quite exactly or otherwise the analysis will fail. If there had been a sequence number in the log files which refers to the model solution, it would have helped the analysis a lot but however the analysis tool worked well and we could get the results we were looking for. The results are presented in the next chapter.

Chapter 6

Results

In this chapter we present the results from the analysis. We have collected the errors to four different categories and for each error we explain what the students have done, what they should have done and what may have caused the error.

6.1 Background for the data

The results presented in this chapter are processed from the data that has been collected in 2010, 2011 and 2012. UUhistle has been used in Aalto University's course T-106.1208 *Basics of Programming Y (Python)* where about 45 percent of the students did not have any kind of previous knowledge of programming.

This course is arranged twice in a year. First in the spring semester and then as a summer course but the course arrangements and the backgrounds of the students are different in the summer and therefore we have used only the data collected in the spring semesters. The number of the participants has been approximately between 600 and 700 students. The exact numbers are reported in Appendix A.

In 2010, we used UUhistle for the first time and the UUhistle exercises were not mandatory for the students. The UUhistle exercises were also in a completely different system where the mandatory programming exercises were available. Because the UUhistle exercises could be seen as extra material but the points were required to get the highest grades, the number of students using UUhistle in 2010 was smaller than in 2011 although the number of the participants in the course was almost equal.

In 2011 and 2012, UUhistle exercises were in Goblin online assessment system instead of TRAKLA2 system together with the programming exercises and the UUhistle exercises were also a part of the course and therefore all the participants have at least tried UUhistle. To pass the course the students had to get at least the minimum points from each exercise round and usually the UUhistle exercises were an easy way to get some points and therefore there is more data available from these two years.

When comparing the students between these three years by using the background questionnaire the students filled when the course started, the answers are quite the same. In the spring 2010 when the UUhistle exercises were not mandatory, the students who used UUhistle were a bit more motivated and got better grades than the students who did not use UUhistle. In their previous knowledge there were no differences.

This might mean that there is a slight difference between the students in 2010 and the next two years which may affect the result presented in the next sections.

We analyzed only the submissions where the student had solved the exercise correcting all the errors because otherwise the analysis tool would not have worked correctly. There were also a small number of log files that needed to be skipped because the files could not be fixed in a reasonable time. In some situations, UUhistle had stepped two steps backwards after undo because of a bug in the tool. There were also steps that were logged twice and this caused problems. These problems were hard to recognize automatically during the analysis process and after this kind of problem the analysis tool can not trace the execution anymore correctly.

The number of analyzed log files is still very high, the average count of analyzed log files is 95 percent and there are only five cases where the percentage is below 90 percent. The lowest percentage is in exercise 4.4 in the year 2010, 81.6 percent. The average number of analyzed log files per exercise in each year is about 500 and the total number of the log files is over 24000. The exact numbers are available in Appendix B.

6.2 Common errors

We assume that there are four main reasons behind the errors:

- Simple mistakes
- Errors caused by the user interface

- Errors caused by the strict simulation order
- Errors caused by a misconception

Simple mistakes mean that if the simulation exercise consists of many steps, it is possible (and quite probable) that at some point a student just makes a mistake without any particular reason.

Errors caused by the user interface cover the errors that the students have made most probably because of some issues in the user interface instead of they have understood something incorrectly. The problem might be, for example, that they know how the execution should proceed but they do not know how to simulate the step in UUhistle.

Errors caused by the strict simulation order contain the errors that are likely caused by the nature of the simulation exercises. The simulation steps must be made in a very strict order and because the execution model is not that clear to all students, this may cause errors.

Errors caused by a misconception is the most interesting reason for the errors. We would like to show that we can find similar errors which have been reported as misconceptions in the literature. By finding the errors of this type we can suspect a possible misconception based on the earlier research but we can also state that the reported misconceptions exist.

The errors we report here are most likely caused by all of those four reasons, but we presume that the share of the simple mistakes is very low and the three other reasons are more likely. We have not reported all errors because the number is very high (normally there are at least two or three incorrect steps in every step) but we have chosen the most common errors we could also explain with the help of the literature and other knowledge.

In this section, we report the discovered common errors grouped to four categories based on the topic: basics, branches and loops, functions and object-oriented programming. Each subsection starts with a table which shows how many percent of the students have made the described error instead of the correct simulation step.

The percentage tells how many students have done the incorrect step as their first simulation step at that particular point. This means that if the first simulation step was incorrect, the students click undo and try again with a wrong step, only the first wrong step is taken account.

We assume that usually the first step is the one that the student really thinks to happen next and the logs reveal that if the students do not know what to do next,

they can try many wrong steps and test which simulation step might be the correct one. Because the students are not thinking as much as they should in this situation, counting all incorrect simulation steps might lead to inaccurate results. The same analysis was also done so that the algorithm took all the steps and not only the first one. This did not affect the most common errors but the number of different incorrect steps increased per one simulation step. This means that if the first step was not correct, the students tried many other incorrect steps before they finally found the correct one.

6.2.1 Basics

This subsection contains common errors about the basics of programming, such as the evaluation order and assignments. The most common errors are presented in Table 6.1.

Table 6.1: The most common errors in the basics of programming

Error	Percentage		
	2010	2011	2012

B1: Forgetting to assign (ex. 1.2)
`celsius = 100`
`fahrenheit = 1.8 * celsius + 32`
	9 %	0 %	0 %

B2: Starting always by creating a new variable (all exercises) `fahrenheit = 1.8 * celsius + 32`	varies		0 %
B3: Forgetting to evaluate multiplication (ex. 1.2) `fahrenheit = 1.8 * celsius + 32`	53 %	22 %	29 %
B4: Inverted assignment (ex. 1.3) `first = second`	27 %	26 %	38 %
B5: Multiplying a float with a function (ex. 1.6) `fahrenheit = 1.8 * float(celcius) + 32`	40 %	N/A	N/A
B6: Starting by dragging `float` function (ex. 1.6) `fahrenheit = 1.8 * float(celcius) + 32`	9 %	22 %	21 %
B7: Incorrect order in a nested statement (ex. 1.7) `value = float(raw_input('Give a value:'))`	20 %	17 %	13 %

B1. Forgetting to assign

```
celsius = 100
fahrenheit = 1.8 * celsius + 32
```

What should have been done? At the first line students should create a new variable and then assign the value `100` to the variable `celsius`. After that the execution automatically jumps to the next line.

What have students done? Many students have dragged the correct value `100` to the evaluation area but instead of assigning it to a variable, they have jumped to the next line or dragged the multiplication operator to the evaluation area.

What might have caused this? Because this is the first simulation exercise, the most probable reason is that the usage of UUhistle is not clear. Also the meaning of the assignment might not be fully understood at this point.

There were step-by-step instructions and a video available how to solve this first simulation exercise, but however, it seems that the instructions were not clear enough or the students did not read them carefully. The students were asked to fill a course feedback form and there was a question did they watch the videos explaining how to use UUhistle. According to the feedback about a half of the students have watched at least one video in 2010 and 2011.

How have we changed the exercise? After the year 2010 in addition to the instructions before the assignment, UUhistle also showed step-by-step instructions how to proceed throughout the exercise. This made it easier to start and the same mistake could not be found in 2011 and 2012.

B2. Starting always by creating a new variable

For example: `fahrenheit = 1.8 * celsius + 32`

What should have been done? If there is a code line which assigns the right-hand side to a new variable, the right-hand side must be evaluated before the new variable can be created. This is because in Python a variable is always bound to a value and therefore there can not be an uninitialized variable.

What have students done? Almost in all exercises there were many students who have always started simulating that kind of lines by creating the variable first. It seems that they did not learn that a variable can not be created if the next step is not an assignment because the same mistake existed throughout the exercises.

The percentage of this error was between 20 and 50 which describes the severity of this problem.

What might have caused this? Because the statements are evaluated from left to right, it is natural to start by creating a new variable first. It is also a language-specific case that in Python there can not be variables without a value.

How have we changed UUhistle? We noticed this was a major problem and a usability issue and therefore we changed UUhistle a bit before the course started in 2012. In the new version the step to create a new variable was removed and now variables are created by dragging the initial value to the variable area and selecting the name for the new variable. Now when students need the value to be assigned, they know better when to create a new variable, there is one step less and the user interface is more intuitive to use.

Because this error was very common with a high percentage and in 2012 this error could not occur anymore, this can also affect the percentages of the other errors.

B3. Forgetting to evaluate multiplication

```
fahrenheit = 1.8 * celsius + 32
```

What should have been done? After the multiplication `1.8 * celsius` has been dragged to the evaluation area, the multiplication should be evaluated before the addition operator is dragged next to the result of the multiplication.

What have students done? Many students have dragged the addition operator to the evaluation area before evaluating the multiplication.

What might have caused this? Students should know the correct calculation order (multiplication has higher precedence than addition) but they did not realize that the multiplication should have been evaluated already before adding the addition operator.

How have we changed the exercise? After the year 2010 in addition to the instructions before the assignment, UUhistle also showed instructions how to proceed while solving the exercise. This has reduced the number of mistakes but obviously not all the students have read the next step carefully because the percentage is still between 22 and 29.

B4. Inverted assignment

```
first = second
```

What should have been done? If there is an assignment where the left-hand side is a variable and the right-hand side is also a variable, the value of the right-hand side variable should be assigned to the left-hand side variable.

What have students done? The students have assigned the value from the left-hand side variable to the right-hand side variable.

What might have caused this? If there are only two variables separated by an assignment operator, it might not be clear which variable is the source and which the destination. It is interesting that still in the 9th round about 10 percent of the students made the same mistake. This error has also been quite common while the students have solved the coding exercises and is well known in the literature.

How have we changed the exercise? This exercise has not been changed at all. We do not know why the percentage has increased in 2012.

B5. Multiplying a float with a function

```
fahrenheit = 1.8 * float(celcius) + 32
```

What should have been done? To be able to evaluate the multiplication, both of the operands should be values. Therefore `float` type conversion function should have been evaluated before the multiplication.

What have students done? Students have tried to evaluate the multiplication so that the second operand is a function call.

What might have caused this? Because the type conversion functions only change the type without any concrete effects to the value itself, students may think that it is only a wrapper that tells the value should be handled as a float without knowing that `float` is a function. At this point, functions are also quite unfamiliar to the students.

The information about this error is not available from the years 2011 and 2012 because a new error dialog was added to prevent evaluating an operator with incorrect parameters. Unfortunately, UUhistle did not log that step at all.

B6. Starting by dragging `float` function

```
fahrenheit = 1.8 * float(celcius) + 32
```

What should have been done? Before adding `float` function call to the evaluation area, the value `1.8` and multiplication operator should be dragged to the evaluation area first.

What have students done? Students have dragged the `float` function to the evaluation area first without adding the preceding elements.

What might have caused this? When calculating this kind of statements with pen and paper, the evaluation order is not strict and evaluating a function first is usually a good place to start if the statement can be evaluated after that. Students may have tried to apply the same strategy here without realizing that the computer evaluates the statements always in a strict order and therefore the evaluation should be started by adding the value `1.8` to the evaluation area.

How have we changed the exercise? We have not changed this exercise. We cannot explain why the percentage have increased in 2011.

B7. Incorrect order in a nested statement

```
value = float(raw_input('Give a value:'))
```

What should have been done? If there are nested function calls, the outermost function call should be dragged to the evaluation area first and then construct the inner function call as a parameter. This corresponds to the way how the return value of the inner function will be a parameter for the outer function. In UUhistle this is also the only way to simulate this step.

What have students done? Many students have started this line by dragging `raw_input` to the evaluation area instead of `float`.

What might have caused this? The students might have thought that they can add the result of `raw_input` to `float` function later although this is not possible. The nested structures are also difficult to understand at this point and for some students they cause trouble throughout the course although the idea should be familiar from mathematics.

6.2.2 Branches and loops

This subsection contains the most typical errors when simulating `if` statements or `while` loops. The errors are presented in Table 6.2.

Table 6.2: The most common errors related to branches and loops

Error	Percentage		
	2010	2011	2012

BL1: Dragging `print` instead of jumping (ex 2.2)
`if divisor == 0:`
`  print 'Chuck Norris divides by zero, you do not.'`
`else:`
`  print 1000 / divisor`
	23 %	22 %	25 %

BL2: Jumping to wrong branch (ex 2.2) `if divisor == 0:`	11 %	17 %	17 %

BL3: Executing wrong branch without jumping there (ex 5.1)
`if not information_ok(place, distance):`
`  return False`
	52 %	35 %	45 %

BL4: Assigning result back to variable (ex 2.2) `if divisor == 0:`	8 %	8 %	6 %
BL5: Assigning result back to variable (ex 3.2) `while i < 7:`	8 %	0 %	3 %

BL1. Dragging `print` instead of jumping

```
if divisor == 0:
  print "Can't divide!"
else:
  print 1000 / divisor
```

What should have been done? After the boolean value is in the evaluation area as the result of the comparison, the student should click the next line because now there are two possibilities where the execution moves.

What have students done? Many students have dragged the `print` statement to the evaluation area before jumping to the correct line. The same effect with almost the same percentage is also visible with `while` statements.

What might have caused this? Normally, UUhistle jumps to the next line after completing the current line. However, there is an exception: if there is a branch, the student must click the next line in order to show that the idea of `if` or `while` statement is clear.

At this point, UUhistle flashes the line number bar on the left-hand side of the code and also shows a text in the status area the student should select the next line. Although UUhistle tells what to do next, it seems that many students did not notice this or understand the next step correctly.

BL2. Jumping to wrong branch

```
if divisor == 0:
```

What should have been done? Because in this case the result of the comparison was `False`, students should have jumped to `else` branch.

What have students done? Although the result was `False`, some students have proceeded to the following line as the result had been `True`.

What might have caused this? It seems that the students have not understood how the result of the `if` statement affects the control flow. Some students may have believed that all lines are always executed.

BL3. Executing wrong branch without jumping there

```
if not information_ok(place, distance):
  return False
```

What should have been done? Because the result of the return value was `True` and inverting it causes it to become `False`, the execution should pass the line inside `if` statement and continue below that. In this case there was not `elif` or `else` blocks.

What have students done? A high number of students have returned value `False` straight after evaluating `not` operator without even jumping to the line containing `return` statement.

What might have caused this? The combination of using the return value of the function and then inverting the result may have confused the students. But because the error is from the fifth round where functions and branches should be familiar, the percentage of the students seems to be very high.

BL4. Assigning result back to variable

```
if divisor == 0:
```

What should have been done? After evaluating the comparison, the student should jump to the correct line and continue the execution.

What have students done? Some students have assigned the boolean value back to the variable `divisor` replacing the original value.

What might have caused this? The comparison operator looks very similar to the assignment operator and it is possible to misinterpret the statement. Throughout the course there are students who do not remember that == should be used when comparing values and sometimes somebody tries to assign a value by using == operator. Also in mathematics there is normally only = operator which makes == operator hard to remember and understand the difference between the two operators.

BL5. Assigning result back to variable

```
while i < 7:
```

What should have been done? As in the previous case, after resolving the value of the statement, students should select the next line.

What have students done? The boolean value has been assigned back to variable `i`.

What might have caused this? Now the operator in the statement is < and therefore there should not be confusion with the assignment operator. Still some some students may have thought that this comparison changes the value of the variable or the result of the comparison should be saved. Most likely this error is caused by a simple mistake.

6.2.3 Functions

This subsections presents the most common errors that students have made in the exercises related to function calls. The errors are in Table 6.3.

Table 6.3: The most common errors related to functions

Error	Percentage		
	2010	2011	2012
F1: Executing function instead of defining it (ex. 4.2) `def ask_euros():`	17 %	15 %	16 %
F2: Constructing a new function call instead of assigning return value (ex. 4.2) `text = ask_euros()`	20 %	16 %	29 %
F3: Creating parameter variable to wrong frame (ex. 4.3) `def calculate(first, second):`	6 %	7 %	11 %
F4: Creating new function call instead of passing parameters (ex. 4.3) `def calculate(first, second):`	29 %	33 %	45 %
F5: Trying to start function call before evaluating parameters (ex. 4.3) `result = calculate(result, result + 1)`	7 %	N/A	10 %
F6: Creating a variable for return value to wrong frame (ex. 4.3) `return second * 2 + first`	17 %	27 %	24 %
F7: Assigning return value back to variable instead of returning it (ex. 4.4) `return intermediate * intermediate`	5 %	9 %	8 %

F1. Executing function instead of defining it

```
def ask_euros():
```

What should have been done? When the program starts, UUhistle goes through the functions in the same way as the interpreter. When coming first time to a line defining a function, the student should click the Functions panel on the right side of UUhistle and add the function there by selecting it from a drop-down menu.

What have students done? Some students have created a new frame for the function which is not even called yet or start to simulate the first line inside the function although the current line points to the definition of the function.

What might have caused this? The parsing phase is normally totally invisible to the students and therefore it can be hard to understand what should be done. Al-

though the visualization was changed so that the whole function is highlighted in the code panel instead of only the definition line and in the information area there is a help link available, these changes have not changed the behavior much.

F2. Constructing a new function call instead of assigning return value

```
text = ask_euros()
```

What should have been done? After returning to the line where the function call was started, the return value should be assigned to the variable on the left-hand side of the assignment statement.

What have students done? The students have started to construct the function call again by dragging the function call to the evaluation area or creating a new frame.

What might have caused this? It seems that the students can not understand the function call and its purpose totally. After jumping back the students think that line should be executed in a similar way as they arrived to that line at the previous time. They did not understand that execution of that line was interrupted and should be continued after getting the result from the function that was called.

How have we changed the exercise? Since 2011 there has been a help dialog available which explains how to simulate function call. The percentage of this mistake decreased in 2011 but we can not explain why the percentage has increased that much between 2011 and 2012 because the exercise is the same.

F3. Creating parameter variable to wrong frame

```
def calculate(first, second):
```

What should have been done? After calling the function the execution jumps to the line containing the definition of the function. At that point, the student should create a new frame for that function call.

What have students done? Instead of creating a new frame, the students have created the parameter variable to that frame where the function was called.

What might have caused this? The purpose of the frame and its relation to a function call seems to be unclear or the simulation step to create the new frame is unclear.

F4. Creating new function call instead of passing parameters

```
def calculate(first, second):
```

What should have been done? When a new stack frame is created, the next step is to pass the parameters to parameter variables. This is done by creating a new variable to the new frame and dragging the correct parameter from the function call to that variable as shown in Figure 6.1.

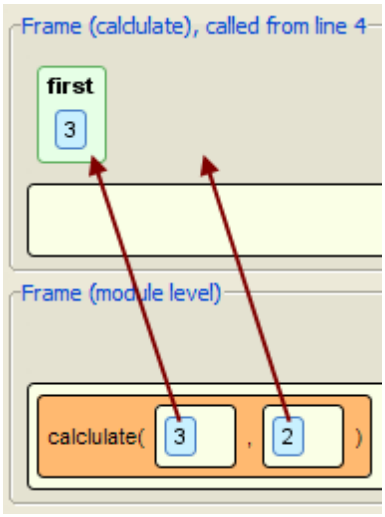

Figure 6.1: In UUhistle the parameter passing process is simulated by dragging the parameters from the function call and assigning them to the corresponding parameter variables in the frame above the function call.

What have students done? Some students have started to construct a new function call in the new frame instead of passing the parameters.

What might have caused this? The most likely reason is that the students did not know what they should do although in 2011 and 2012 there is a help dialog available. The students have dragged a new function call to the evaluation area because it is the most similar element to the current line.

How have we changed the exercise? Between the years 2011 and 2012 this exercise has not been changed. We do not have an explanation why the percentage have increased that much in 2012.

F5. Trying to start function call before evaluating parameters

```
result = calculate(result, result + 1)
```

What should have been done? The second parameter of the function was a statement that must be evaluated before starting the function call. When a function call starts, all the parameters must be single values.

What have students done? The students have tried to start the function call before evaluating the second parameter.

What might have caused this? The students may thought that they can pass a statement as a parameter and the value is evaluated later. A very common misunderstanding in the exercise sessions is that inside a function they can also access the local variables of the other functions. This indicates that the parameter passing and the role of the local variables are not clear.

F6. Creating a variable for return value to wrong frame

```
return second * 2 + first
```

What should have been done? After the return value has been evaluated, the return value should be dragged to the frame below on the top of the active call as shown in Figure 6.2. When the value has been dragged to the correct place, the topmost frame disappears and the function call is replaced with the return value.

What have students done? Instead of returning the value, some students have created a new variable for the return value in the active frame. The name of the variable is the same as where the return value should be assigned in the caller's frame. In the spring 2010 and 2011 UUhistle allowed to create new variables only to the topmost frame.

What might have caused this? The students have understood that the return value should be assigned somewhere but they have not realized in which frame the return value should be saved. Because UUhistle did not allow to create a new variable to the lower frames, the topmost frame might have been thought to be a good place.

This kind of mistake does not appear in the previous exercise where the return value taken from a variable instead of evaluating a statement. In that exercise, the students did not have problems to simulate this step and therefore it is not probable that the students did not know how to simulate the return step.

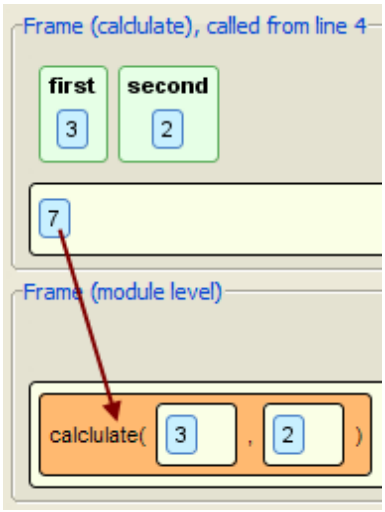

Figure 6.2: Returning the return value is simulated by dragging the return value from the upper frame on top of the active function call in the lower frame. After that the function call will be replaced with the return value and the topmost frame disappears.

F7. Assigning return value back to variable instead of returning it

```
return intermediate * intermediate
```

What should have been done? After evaluating the return value, it should be returned to the frame where the function was called.

What have students done? Some students have assigned the return value back to that variable which was used to calculate the return value instead of returning the value.

What might have caused this? In this exercise a local variable was used to calculate the return value. Some students may believe that the local variable should also be updated although it will not be read anymore inside the function and can not be accessed after the function call has ended.

6.2.4 Object-oriented programming

This subsection contains the most typical errors related to the object-oriented exercises. The errors are presented in Table 6.4.

Table 6.4: The most common errors related to object-oriented programming

Error	Percentage		
	2010	2011	2012
OO1: Calling method without reference (ex. 9.2) `car1.fuel(40)`	18 %	38 %	33 %
OO2: Creating new object instead of assigning reference (ex. 9.2) `car3 = car1`	12 %	6 %	14 %
OO3: Assigning reference to uninitialized object (ex. 9.2) `car1 = Car(45)`	6 %	12 %	9 %
OO4: Creating incorrectly instance variable (ex. 9.2) `car1 = Car(45)`	7 %	10 %	11 %
OO5: Dragging reference to existing object instead of creating new (ex. 9.2) `car2 = Car(60)`	16 %	7 %	18 %
OO6: Creating local variable instead of instance variable `self.__name = firstname`	8 %	7 %	3 %
OO7: Returning object reference instead of value of instance variable (ex. 9.4) `return self.__profession`	6 %	15 %	15 %

OO1. Calling method without reference

```
car1.fuel(40)
```

What should have been done? When starting to construct a method call, the first step is to drag to the evaluation area a reference to that object whose method is being called. After the reference is in the evaluation area, the method call will be dragged next to the reference.

What have students done? Some students have started this line by dragging the method call first to the evaluation area.

What might have caused this? There are two possible explanations for this mistake. Students have not understood the difference between function and method calls or the Python way of declaring a method with `self` parameter is misleading although the evaluation order is similar to the code where the method is called. In addition to these, students may have not thought that it is not possible to know

from which class the method will be found before the reference is available.

How have we changed the exercise? We have not changed the exercise. Since the year 2011 UUhistle offers more help with methods but still the percentage of the error has increased. The reason for this is unknown.

OO2. Creating new object instead of assigning reference

```
car3 = car1
```

What should have been done? Create a new variable `car3` and assign to that variable the same object reference that is assigned to the variable `car1`.

What have students done? Some students have started this line by creating to the heap a new object although this line does not create any new object instances.

What might have caused this? The concept of references can be unclear to the students although in the previous exercise rounds lists and dictionaries have been handled via references. Some students may have thought that this line would generate a similar object as in `car1` to variable `car3` by creating a new object not having realized that the variable contains only a reference.

OO3. Assigning reference to uninitialized object

```
car1 = Car(45)
```

What should have been done? After the object is created to the heap, the reference should be dragged to the evaluation area and start to call `__init__` method which initializes the created object.

What have students done? The students have created a new variable `car1` after they have created a new object to the heap instead of starting to call `__init__` method which should be executed before the reference to the object will be exposed.

What might have caused this? The code line does not have any kind of clue that `__init__` method is called at this point because this is normally taken care by the interpreter. Students might think that after the object is created, the right-hand side is evaluated and the reference can be assigned to the variable. However, this does not explain what the students would do with the parameter `45` in order to initialize the object.

OO4. Creating incorrectly instance variable

```
car1 = Car(45)
```

What should have been done? As in OO3, after the object is created to the heap, the reference should be dragged to the evaluation area and start to call `__init__` method which initializes the created object.

What have students done? After the new object is created to the heap, instead of creating a new local variable as in OO3, they have created a new instance variable `car1` to the object. In this exercise the internals of the class were not shown and therefore the students did not have to create or modify any instance variables.

What might have caused this? The roles of instance variables and local variables might not be clear at this point because the instance variables will be presented in the next exercises. Some students may have believed that the variable in the left-hand side is somehow a part of the created object.

OO5. Dragging reference to existing object instead of creating new

```
car2 = Car(60)
```

What should have been done? At this line, the student should create a second `Car` object instance to the heap and initialize the created object using `__init__` method.

What have students done? Some students have dragged a reference to the first `Car` object to the evaluation area instead of creating a new instance.

What might have caused this? The students may have not realized that this kind of line creates a new object always. The students may have thought that because they already have one `Car` object they could initialize it somehow again.

OO6. Creating local variable instead of instance variable

```
self.__name = firstname
```

What should have been done? A new instance variable `__name` should be created to the new object that is being initialized. The value in the parameter variable `firstname` is assigned to that instance variable.

What have students done? Instead of creating a new instance variable, some students have created a local variable to the stack frame.

What might have caused this? The concept of classes, objects and instance variables are very different compared with the exercises in the previous eight rounds and it might be hard to understand first how the object-oriented world is constructed.

How have we changed the exercise? In the version which was used in the spring 2012, there is a clear text inside an uninitialized object that the instance variables should be created here. This new text most probably explains the small percentage in 2012 although also in 2011 there was a pop-up window that explained how to create a new instance variable when a student jumped to that line. This pop-up window was also used in 2012 together with the new text.

OO7. Returning object reference instead of value of instance variable

```
return self.__profession
```

What should have been done? In this line students should follow the reference in variable `self` and return instance variable `__profession` from that object. The method is a normal *getter* which contains only this one line.

What have students done? Some students have returned the reference to the object instead of returning the value of the instance variable.

What might have caused this? Most likely the students have not realized the purpose of this method or it is also possible that the simulation sequence is not clear. The reference should only be followed, not to be dragged anywhere although this will not explain why the students have returned the reference to the object instead of adding it to the evaluation area.

6.3 Explanation texts as a part of the exercises

The log files also give a possibility to see how many students have read the feedback and explanation texts UUhistle provides. In 2012 we added some new dialogs which explained why some simulation steps are incorrect if the students did some incorrect simulation steps that we recognized to be a possible caused by a misconception. These dialogs were not shown automatically but students could

click a link and read the explanation in a pop-up window. Figure 6.3 shows an example of the situation after an incorrect step where UUhistle offers an explanation why the step was incorrect.

When we analyzed how many students read those explanations, the percentage seems to be extremely low. We made this analysis only to the data collected in 2012 because before that UUhistle did not give that much feedback.

In program animation exercises where students only watch the animation, UUhistle gives a possibility to open an explanation containing an automatically generated explanation of the previous step. A total number of students using this feature at least once during the whole course is 219, which is 37 % of the students. This means that over a half of the students did never read any of these explanation texts.

While a function call is simulated, UUhistle always shows a link to a help text explaining how a function call works and how the different steps are simulated. The number of students who read this help text is exactly the same as the previous number, 219. Figure 6.3 shows a similar link to the help for method calls.

Those explanations were read much more often if compared with the explanation texts related to the possible misconceptions. We analyzed four different situations which were easy to track from the log files and where UUhistle provides an explanation why the simulation step was incorrect.

One of the most common simulation errors is to execute a single statement in a wrong order. We added an explanation which tells why the operators must be evaluated and dragged to the evaluation area in a strict order. The link to this explanation will become visible, for example, after the error B3 *(Forgetting to evaluate multiplication)*. Totally 594 students made this mistake during the whole course at some point but only 35 students (6 %) ever read the explanation why the simulation step they just did was not correct.

The percentages of the three other explanations for a possible misconception are even lower. 159 students made the error F3 *(Creating parameter variable to wrong frame)* during the course but only one student was enough interested why it was not possible to create the parameter variable to the same frame where the function was called and read the explanation.

Two possible misconception related to object-oriented programming covered the errors OO1 *(Calling method without reference)* and OO2 *(Creating new object instead of assigning reference)*. 275 students encountered the first error and 66 students the second error. There were only 3 students who read the explanation for the first error and nobody read the explanation for the second error.

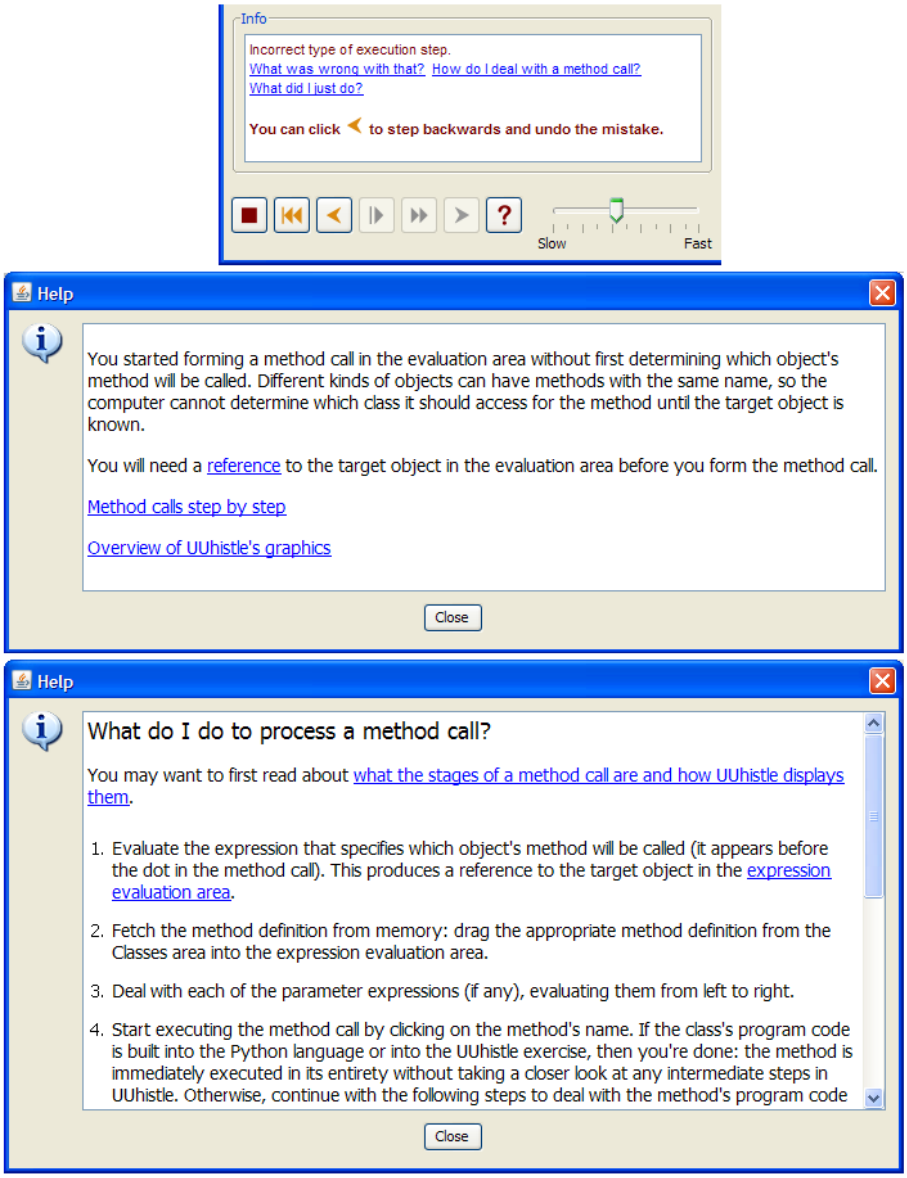

Figure 6.3: After an incorrect simulation step which might indicate a misconception UUhistle has a few built-in explanations discussing what was wrong with that step and why. The first link in the status area opens the explanation of the mistake and the second link shows a general help for simulating a method call which is always available while simulating method calls.

Chapter 7

Discussion

In this chapter we discuss the results we got, compare the results with the literature and consider how to improve the visual program simulation exercises and UUhistle in order to help students realize they have understood something incorrectly.

7.1 Reasons for the errors

For the future development of UUhistle and visual program simulation exercises and VPS tools in general, it is important to recognize whether the errors have occurred because of the UUhistle's user interface, are the errors related to simulation or is there a reported misconception behind the error.

We do not want to accomplish a situation where students would not make any mistakes because then there would not be anything new to learn. However, what we would like to achieve, is to minimize the number of the errors caused by the UUhistle's user interface. If students do not have to struggle with the minor issues related to the UI, they have more time to concentrate to the important aspects of the program execution that we would like to teach them.

7.1.1 Errors caused by the user interface

Many of the errors in the category Basics have most likely occurred because of the new environment. All errors in this category except one come from the first exercise round which means that the knowledge of programming in general is

low and the tool is still unfamiliar. UUhistle requires to simulate the first as the all exercises in a strict order. And when the students are still learning the basics of programming and how to use UUhistle, it is quite natural that students make mistakes.

The error B1 *(Forgetting to assign)* in Exercise 1.2 is a good example of an error that disappeared after improving UUhistle. Since year 2011 UUhistle guided the students in the first simulation exercise and told step-by-step what to do next as shown in Figure 7.1. This helped to understand how to use UUhistle and the percentage of the error B1 dropped from 9 percent to zero.

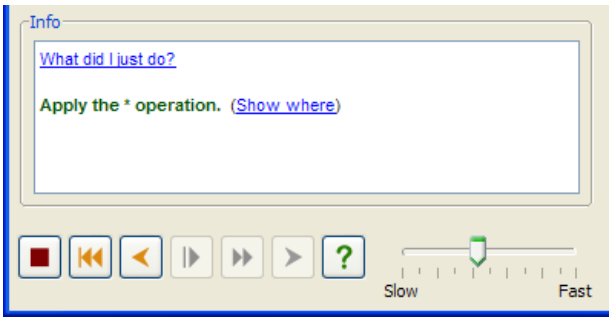

Figure 7.1: UUhistle in tutorial mode showing always what to do next. If a student clicks *Show where* link, UUhistle highlights the element.

This means that when a student should do something new, it is a good idea to give a hint what to do to avoid unnecessary mistakes. After the simulation step is told to the student, we can assume in the later exercises the student should know how to simulate the step.

The error B3 *(Forgetting to evaluate multiplication)* in Exercise 1.2 is an example where the given hints do not always help. Since the year 2011 UUhistle told in the status area to evaluate the multiplication at that point when the students have dragged the addition operator to the evaluation area. The percentage has dropped from 53 % to 22 – 29 % but not below that. This possibly means that all students do not pay enough attention to the status area which contains important information throughout the exercise such as instructions, help links and links to explanations which explains why some simulation steps are not correct. We also noticed that the students do not click the links to open the help and explanation dialogs UUhistle offers. The visibility of this area and its contents should be improved in the future.

The error BL1 *(Dragging print instead of jumping)* in the Exercise 2.2 seems to be also related to the user interface. At this step students should first time select to which line the execution jumps. After evaluating `if` statement there are

two possible lines and UUhistle requires the student to choose the line instead of jumping automatically. If there is a branch, the line number bar flashes and in the status area there is an instruction to select the next line. One fourth of the students may have not noticed this because they have proceeded the execution as the jump would have already happened. In addition to the current visual and textual indication a new pop-up dialog may help students. When a student must do this simulation step at the first time, UUhistle could show a message what to do next and explain how UUhistle indicates this.

One of the biggest changes is related to the error B2 *(Starting always by creating a new variable)*. This error is a combination of the UI and the evaluation order. The old version allowed to create an empty variable if the next step was an assignment which was not exactly correct because in Python a variable is always bound to a value and the step should therefore be atomic. The students did not know when it was allowed to create a new variable and they thought it is allowed to create a new variable always at the beginning of a line. This problem did not disappear after the first exercise rounds and almost a half of the students made this mistake in some exercises. We recognized this problem and it was fixed in the version used in 2012 as described in the Chapter 6. This helped to solve one of the major errors related to the user interface and made it more intuitive to use.

7.1.2 Errors related to previously reported misconceptions

Basics

The visual program simulation involves very different strategies than the usual coding exercises. This means that the errors the students make might be completely different than those when they write code by themselves. But however, VPS exercises seems to expose many same misconceptions that have been reported earlier in another context.

The first error we reported in Chapter 6 that is present in the misconception literature is the error B4 *(Inverted assignment)*. This misconception is already reported in the 1980s by Du Boulay [4] and Putnam et. al [20]. One of the later studies of the same misconception is made by Ma et al. in 2007 [13]. Simon has also discussed how hard assignment statements can be for the students [26]. He found, for example, 71 % of the students who had completed a diploma in IT thought the statement *an assignment statement always assigns from right to left* is false.

This misconception is interesting because students do not have any problems to understand assignment statements if the right-hand side is an expression to be

evaluated or a literal. But if the right-hand contains only a name of a variable, students get confused. It is also interesting that there were students in the courses we analyzed who did not learn the right order during the course and did the same mistake still in the last exercise round. Of course it is possible to make a simple simulating mistake but because the misconception is well known, the percentage of the students making this mistake is quite high (27 – 38 %) and this case is easy to recognize, it would be a good idea to add to UUhistle a specific explanation which would be shown if a student made this mistake.

Branches and loops

The idea of `if` clause is pretty simple: there is a condition which is evaluated and according to the result, the execution will jump to branch A or B where the A is the code block inside the `if` statement and B is `else` block or the code below `if` statement if `else` block does not exist.

There are two errors, BL2 *(Jumping to wrong branch)* and BL3 *(Executing wrong branch without jumping there)* where the students have not understood which line is to be executed next.

Sleeman et al. have studied the problems the students had on an introductory course where Pascal programming language was used. They have reported a few misconceptions related to `if` statements. In the first error, the students have thought that both `THEN` and `ELSE` are executed and in the second error that `THEN` statement is always executed whether the condition is true or false [27].

In our exercise where the error BL2 occurred, the condition was `False` and therefore the students should have jumped to `else` block in order to avoid dividing by zero. Nevertheless 11 – 17 % of the students have jumped to inside `if` statement as the condition had been `True`. This means that they have not understood correctly how `if` statement is used to control the program execution. The error may be related to the observations Sleeman et al. have reported.

In the error BL3 `if` statement did not have `elif` or `else` blocks and therefore the students should have jumped over the code inside `if` statement because the condition is first `True` but after evaluating `not` operator the result is `False`. 35 – 52 % of the students have decided to execute `return` statement inside `if` statement without even jumping to that line. Somehow the students have misunderstood how the execution continues or `not` operator have confused them.

The error BL4 *(Assigning result back to variable)* shows a possible confusion with equality and assignment operators. This is a real problem because while

students are solving coding exercises too many students can not remember that they should use == operator in `if` statement. This misconception is well known in the literature. The reason for BL4 is probably that the students have found `operator` and evaluated the result but after that interpreted the meaning of the operator incorrectly and made an assignment.

We have no evidence that the students have a misconception where the result of the condition is always assigned back to the variable or otherwise saved although the error BL5 is very similar to BL4 but instead of `if` statement there is `while` statement. The percentage of BL5 is much lower (0 – 8 %) and can be explained as a pure simulation mistake.

In order to react these errors containing possibly misconceptions, UUhistle could give better feedback in these situations. Currently, students can read a static text after jumping to wrong line but this could be changed to be more interactive and have different cases for `if` and `while` statements. If a student assigns a value to a variable after evaluating == operator, this can be easily recognized and UUhistle could show an explanation why the step was not correct by explaining the difference between = and == operators.

Functions

In the first error related to functions, F1 *(Executing function instead of defining it)*, students do not understand properly the phases when an interpreted program is started. The interpreter starts to scan the code and normally finds functions and below all functions a call to `main` function. During this process, the interpreter knows which functions are defined but the functions are not executed because they have not been called yet. In the error F1 some students have thought that when the program starts, the first function in the code will be executed and have started to execute the code inside the function instead of just defining the code although UUhistle have instructed to do so since spring 2011. UUhistle showed a pop-up dialog when the execution started stating that first the functions are only defined by clicking the function area in the GUI and the functions are not executed yet. Nevertheless about 15 percent of the students did a simulation step as they were executing the function.

Sleeman et al. noticed that there are two misconceptions related to the same error as in F1. Some of their students thought that the functions (or actually at this case procedures in Pascal) are executed immediately when a program starts. Some students thought that they are executed first when the program starts and then again when the functions are called [27]. Ragonis and Ben-Ari have reported a similar

misconception where students think that the methods will be called according to the order they are defined in the program code [21].

The rest of the errors in this category we reported in Chapter 6 are related to process how a function call works: evaluating parameters, passing parameters, evaluating the function, returning the return value and possibly assigning it to a variable. The reasons behind the errors F2 – F7 can also be related to UUhistle's user interface but most likely there are misconceptions and misunderstandings.

Functions have always been one of the hardest things to learn in the basic course of programming we investigate in this thesis. It takes a lot of time to understand the concepts and still there are many students having different misconceptions although functions have been used in several rounds. We are not alone with this problem as Madison and Gifford [14] and Fleury [5] have reported many misconceptions they have noticed. They have used Pascal procedures where are no return values. A procedure can have variables that can be seen as references to the original variable and in that way a procedure can communicate with the caller. In Python, as in many other languages, there is a return value that is returned and the return value must be saved to somewhere or used immediately as a part of a statement.

The error F3 *(Creating parameter value to wrong frame)* shows that the students did not know the purpose of a stack frame and how the frame defines the scope of the variables that are available during a function call. When students are solving programming exercises, a very common misconception is that the variables are somehow global and the local variables of a function can still be accessed although the execution of a function call has already ended. The error F4 *(Creating new function call instead of passing parameters)* shows that for the students the parameter passing phase is hard to understand. They do not know what to do after they have created a new stack frame.

One of the errors, the error F5 *(Trying to start function call before evaluating parameters)*, is related to the parameter passing. This error can be caused by the strict simulation order but it is possible that we have found a misconception that is not yet reported in the literature. If students do not understand that each parameter is a single value that is evaluated before the function call starts, they may think that the parameter can be an expression that is used somehow inside the function. In the programming sessions, some students also think that the name of the variable is passed and the function could use the value which is stored in the caller's frame. We do not have currently data to investigate this possibility further but this is an important finding which should be investigated more in the future. UUhistle shows an error message currently if a student tries to begin a function call although the parameters are incorrect, ie. missing or not a single

value. This simple error message should be improved to explain the nature of the passed parameters.

When a function ends and the return value should be returned, the place where the return value belongs is unclear. The errors F6 *(Creating a variable for return value to wrong frame)* and F7 *(Assigning return value back to variable instead of returning it)* show that the students did not understand that the value should be returned to the lower frame and if the return value is saved to a variable, the variable can not be in the frame of the function which generated the value.

After the students have returned the return value and the function call has been replaced with the value, the students do not know what to do with the value. The error F2 *(Constructing new function call instead of assigning return value)* shows that the students can not make the difference between when jumping to the line the first time and jumping back to that line after a function call. Hristova mentions that sometimes students call a method in Java without saving the result although they should [9]. We have noticed the same mistake that the students do not write an assignment statement when they call a function as they should but instead of that they think that they can still use the local variable of the called function containing the return value.

These errors points out that students do not understand how a function call works although there is a program visualization exercise which shows all the important steps. Sorva noticed very similar problems when he analyzed the videos where students where using UUhistle [29]. Because functions are a very integral part of programming and obviously hard to understand, the VPS exercises should concentrate more on these errors. The visualization should help the understanding but if there were more interactive tutorial explaining carefully the steps and their purpose, students might learn better. Currently, it is possible to see the animation containing all the important steps related to functions but if there are students who do not understand the steps and do not read the explanation texts which need to be opened after each step separately, there is a big risk that they can not understand what they are doing while simulating the rest of the exercises.

Object-oriented programming

Object-oriented programming is the field of the recent research. Many CS1 courses are taught with object-oriented languages such as Java and therefore this is a natural research field. However, the course where UUhistle has been used is mostly procedural programming, but the last exercise round covers the very basics of object-oriented programming.

Although there is only one round about objects, we found a few object-oriented misconceptions that are already reported. One of these is the error OO2 *(Creating new object instead of assigning reference)*. In this error the students did not realize that a variable holds only a reference to the object and if that reference is assigned to another variable, no new objects will be created.

Ma used in a experiment reported in his thesis [13] very similar code as we had in the exercise 9.2. Ma created different kind of categories what student thinks that will happen when a code line such as `a = b` is executed, if there is an object reference in the variable `b`. We noticed that 6 – 14 % of the students created a new object instead of just assigning the reference to another variable. Ma has named this as *Assign model*, where a copy of the object is assigned to the variable. If the students using UUhistle had continued the execution after the error, they would most probably have initialized the object they created to be a copy of the original and assigned that to the new variable. Sorva has also reported the same misconception in his article [28]. There is already an explanation of the mistake which can be opened after this error but the number of the students reading the explanation was extremely small. This kind of dialogs could always be opened and the dialog could contain an option not showing it anymore after the student has understood the idea. In this way, more students would probably read the explanation and realize the purpose of the line correctly.

The second misconception is the error OO4 *(Creating incorrectly instance variable)* where the students have created an instance variable with the same name as the local variable where the reference should be assigned. Holland et al. discuss *Identity/attribute confusion* in which students think the local variable (or its name) is part of the object [8]. Ragonis and Ben-Ari [21] have also have noticed that some students think that an instance variable can be used as the identifier or the identifier is one of the instance variables. Sorva reported that some students think that the object is somehow named or the name is part of the object [28]. In the research by Sajaniemi et. al the students were asked to draw a picture of a given program at certain point. In these pictures it was also visible that the students had placed the name of the local variable inside the object [23]. Better feedback in this situation would also be helpful. Currently, the explanations contain only text and hyperlinks but in this kind of situation a picture could also help to understand the idea of the reference.

The third error, OO5 *(Dragging reference to existing object instead of creating new)*, might be related to the misconception where students do not know the difference between a class and an instance. Holland et al. call this as *Object/class conflation* [8]. Sanders and Thomas investigated this and noticed that some students defined three similar classes and instantiated each one once instead of in-

stantiating one class three times [24]. This indicates that students may think that one class can be instantiated only once. In UUhistle the students tried to use the previously created object when they were supposed to create a new object. They may have thought that they should use somehow the same instance already existed and did not realize that the purpose of the corresponding line is to create a new instance and after that there are two instances of the same object in the heap.

7.2 Exercise solving strategies

We asked in the course feedback form what kind of strategies the students have used to solve the simulation exercises. The students could select many suitable options. One of the options was a strategy where the student just tries everything in order to proceed without thinking much before or after simulation steps. In 2011, about 30 % of the students indicated that they have used this strategy. Another option was similar but in that strategy the students stopped in order to think why the step was correct after they found it. The share of the students using this strategy was about the same. More details are in [29].

These results show that for many students the original idea of encouraging students to think what to do does not hold. If students use trial-and-error approach it can explain why they are not interested the additional information as described in Section 6.3. For some students it is also hard to understand the link between UUhistle and the program execution. They can think UUhistle as some kind of puzzle without understanding what they should learn. The log files also showed that there were really students who have solved the whole exercise with trial-and-error approach.

Currently, UUhistle shows only an error message if a student makes an incorrect step. If each step seems to produce this very general message, it will not motivate students. There are only a few more detailed error messages but in the future we should implement at least to the most common errors a detailed message which explains why the step was incorrect and tries in that way correct a possible misconception. If UUhistle can give good feedback, students will hopefully understand that there is something behind the logic. Of course the feedback must be presented in such way that students will read or otherwise it will not help.

To motivate students better UUhistle could be used more frequently in lectures to explain and visualize different concepts together with theory and other examples. In this way UUhistle would not be only a platform providing simulation exercises without a clear connection to the contents of the course.

7.3 Trustworthiness of the results

The percentages we reported in the Chapter 6 are gathered from the log files that were collected during three years in Aalto University. The log files were mostly automatically processed but a small percentage of files needed to be fixed or were skipped. The average number of the analyzed files is above 500 which covers on average over 95 percent of the total number of files that were available. The total number of the analyzed files is over 24000. As we have analyzed a very large number of files, the results should be accurate and reflect on the actual usage.

The same numerical results should be obtained if the same log files are processed with a similar analysis tool. The analysis tool we used is written by the author but no other persons have reviewed it so there is a small probability that the analysis tool has worked incorrectly. To minimize this risk the tool was written so that if it fails to follow the log file the analysis will stop and the incorrect file must be fixed manually or added to the list of the files to be ignored. The results the tool generated reflects well on the other observations we have made and we have no reason to believe that the numerical results are incorrect.

The possible causes for the error presented in Chapter 6 are based on our own experiences and the literature. We have developed the system and been also involved in the course arrangements. Therefore we have quite a good vision of how the students have used UUhistle, what they have learned and what kind of problems they have with UUhistle exercises or programming exercises. We have used this knowledge to create the most likely reasons behind the errors but we must admit that we do not have data that could prove our interpretations to be correct. Our knowledge also affects the interpretations and therefore it is possible that other researchers could make different interpretations from the same numerical results based on their knowledge or other literature references that we have used.

It is also likely that if UUhistle was used in a different kind of context, for example, if the exercises were different or it had been used in a different way as a part of the teaching the results would be different. If another VPS tool was used, it would also affect the results and therefore the results we have presented can not be seen as general results of this kind of simulation exercises.

Chapter 8

Conclusion

The main objective of this master's thesis was to analyze the log files we had collected during three years and see what kind of mistakes the students have made.

The log files contained sequences of the steps the students had made and although the log files were not originally designed to be source to this kind of analysis, the information they contained was very valuable.

We wrote a tool which analyzed over 24 000 log files and after that we could manually recognize 26 common errors from the results the tool generated. This showed that collecting data is useful and provides a good way to analyze later the usage of the educational tools and how students solve exercises. The data we have collected has still potential to be used to create new kind of analysis that are not part of this master's thesis.

The results we got proves that students have problems when they need to think about the smaller or bigger details of the program execution. A part of the common errors we noticed is most probably caused by UUhistle's user interface but many errors are very similar to those which have been reported in the literature earlier.

This shows that because VPS exercises make students think about how the execution continues, program simulation exercises make the misconceptions visible and provide a good way to provide feedback and try to correct misconceptions in early phase of the learning process.

The optimal situation is where a VPS tool itself does not cause problems to students with its user interface. We noticed that some problems can probably be fixed by improving the user interface with some changes. When students do not have to

struggle with the UI, they have more time to concentrate on the important things we want to teach them.

The VPS exercises should be designed so that they reveal the possible misconceptions. We do not want to accomplish a situation where students make no mistakes but instead of that we want them to learn better. VPS tools should give better feedback that explains why the simulation step was incorrect and, if possible, somehow explain or show what would happen if the execution continues as the student thought. If a VPS tool can show the concept the student has causes problems, the student can accept the correct model more easily compared to a situation where the tool only tells that the step was not correct without any explanation.

We also noticed that although the tool could show more information about the incorrect steps, only a very small number of the students read the explanations.

In the future VPS tools should encourage students to think more especially when they make mistakes and avoid the current trial-and-error approach where the students just try to guess the next step instead of thinking about the process. If VPS exercises contained more dialog between the student and the tool, students might see the exercises more attractive instead of a program that complains about ”everything”.

A good start to create better VPS exercises is to improve UUhistle's user interface to avoid the error caused by the UI and write new explanation texts for those errors that were the most common ones. If UUhistle can give more detailed feedback in a way that students read it, they will hopefully learn why the step was incorrect and notice that the tool can give useful feedback which helps to understand the difficult concepts and make easier to learn.

Bibliography

[1] Bayman, P., and Mayer, R. E. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM 26* (September 1983), 677–679.

[2] Bonar, J., and Soloway, E. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction 1*, 2 (June 1985), 133–161.

[3] Donmez, O., and Inceoglu, M. M. A web based tool for novice programmers: Interaction in use. In *Proceedings of the international conference on Computational Science and Its Applications, Part I* (2008), ICCSA '08, Springer-Verlag, pp. 530–540.

[4] Du Boulay, B. Some difficulties of learning to program. *Journal of Educational Computing Research 2*, 1 (1986), 57–73.

[5] Fleury, A. E. Parameter passing: the rules the students construct. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education* (1991), SIGCSE '91, ACM, pp. 283–286.

[6] Fleury, A. E. Programming in Java: student-constructed rules. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education* (2000), SIGCSE '00, ACM, pp. 197–201.

[7] Hiisilä, A. Kurssinhallintajärjestelmä ohjelmoinnin perusopetuksen avuksi. Master's thesis, Teknillinen korkeakoulu, Espoo, Finland, 2005.

[8] Holland, S., Griffiths, R., and Woodman, M. Avoiding object misconceptions. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education* (1997), SIGCSE '97, ACM, pp. 131–134.

[9] Hristova, M., Misra, A., Rutter, M., and Mercuri, R. Identifying and correcting java programming errors for introductory computer science students.

In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (2003), SIGCSE '03, ACM, pp. 153–156.

[10] Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education* (2010), SIGCSE '10, ACM, pp. 107–111.

[11] Kollmansberger, S. Helping students build a mental model of computation. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (2010), ITiCSE '10, ACM, pp. 128–131.

[12] Korhonen, A., Helminen, J., Karavirta, V., and Seppälä, O. Trakla2. In *Proceedings of the 9th Koli Calling International Conference on Computing Education Research* (November 2010), A. Pears and C. Schulte, Eds., University of Joensuu, pp. 43–46.

[13] Ma, L. *Investigating and improving novice programmers' mental models of programming concepts*. PhD thesis, University of Strathclyde, Glasgow, United Kingdom, 2007.

[14] Madison, S., and Gifford, J. *Parameter Passing: The Conceptions Novices Construct*. Distributed by ERIC Clearinghouse, 1997.

[15] Mayer, R. E. The psychology of how novices learn computer programming. *ACM Comput. Surv. 13* (March 1981), 121–141.

[16] Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. Visualizing programs with Jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces* (2004), AVI '04, ACM, pp. 373–376.

[17] Myller, N. Automatic prediction question generation during program visualization. In *Proceedings of the 4th Program Visualization Workshop* (2006), University of Florence, pp. 89–93.

[18] Online tutoring system (OTS) web page. Accessed March 2012. `http://www.kolls.net/ots/`.

[19] Pea, R. D. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research 2* (1986), 25–36.

[20] Putnam, R., Sleeman, D., Baxter, J. A., and Kuspa, L. K. A summary of misconceptions of high school basic programmers. *Educational Computing Research 2*, 4 (1986), 459–472.

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

[22] Rajala, T., Laakso, M.-J., Kaila, E., and Salakoski, T. VILLE - a language-independent program visualization tool. In *Conferences in Research and Practice in Information Technology* (2007), L. Raymond and Simon, Eds., vol. 88, Australian Computer Society, Inc.

[23] Sajaniemi, J., Kuittinen, M., and Tikansalo, T. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. In *Proceedings of the 3rd international workshop on Computing education research* (2007), ICER '07, ACM, pp. 1–16.

[24] Sanders, K., and Thomas, L. Checklists for grading object-oriented CS1 programs: concepts and misconceptions. In *ITiCSE* (2007), pp. 166–170.

[25] Sheil, B. A. The psychological study of programming. *ACM Computing Surveys 13* (March 1981), 101–120.

[26] Simon. Assignment and sequence: why some students can't recognise a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (2011), Koli Calling '11, ACM, pp. 10–15.

[27] Sleeman, D., Putnam, R. T., Baxter, J., and Kuspa, L. An introductory Pascal class: A case study of students' errors. In *Teaching and Learning Computer Programming: Multiple Research* (1988), pp. 237–257.

[28] Sorva, J. Students' understandings of storing objects. In *Proceedings of 7th Baltic Sea Conference on Computing Education Research (Koli Calling)* (2007), R. Lister and Simon, Eds., vol. 88 of *CRPIT*, ACS, pp. 127–135.

[29] Sorva, J. *Visual Program Simulation in Introductory Programming Education*. PhD thesis, Aalto University, Espoo, Finland, 2012.

[30] Sorva, J., and Sirkiä, T. Uuhistle: a software tool for visual program simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (2010), Koli Calling '10, ACM, pp. 49–54.

[31] Sorva, J., and Sirkiä, T. Context-sensitive guidance in the UUhistle program visualization system. In *Proceedings of the 6th Program Visualization Workshop* (2011), G. Rößling, Ed., Technische Universität Darmstadt, pp. 77–85.

[32] Spohrer, J. C., and Soloway, E. Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (1986), CHI '86, ACM, pp. 183–191.

[33] UUhistle web page. Accessed May 2012. `http://www.uuhistle.org`.

[34] ViLLE web page. Accessed March 2012. `http://ville.cs.utu.fi/`.

Appendix A

Students and their backgrounds

The following table shows how many students participated to the programming course where UUhistle was used. Because in 2010 the UUhistle exercises were voluntary, not all students did the exercises. In 2011 and 2012 the UUhistle exercises were a part of the course as the other exercises and therefore all students have solved at least one UUhistle exercise during the course.

Table A.1: The total number of the students

year	participants	students using UUhistle
2010	691	571
2011	688	688
2012	597	597

The previous programming experience of the students who used UUhistle is presented in the following table. The question in the background questionnaire was how long was the largest program they had coded before the course started. Most of the students did not have any or only a very limited previous experience. N/A means the student did not know the answer or did not want to answer.

Table A.2: Previous programming experience measured on the length of the largest program the students had written before the course

year	0 lines	below 100	below 500	below 5000	longer	N/A
2010	44 %	28 %	11 %	6 %	1 %	8 %
2011	45 %	25 %	9 %	6 %	1 %	11 %
2012	47 %	29 %	6 %	3 %	1 %	10 %

Appendix B

The number of the analyzed log files

The following table shows how many log files were successfully analyzed. The average percentage of skipped files is about 5 % but there are only five cases where the percentage is less than 90 %. The lowest percentage is 81.6 % in the exercise 4.4 in 2010.

Table B.1: The number of the analyzed log files per year and exercise

year	ex 1.2	ex 1.3	ex 1.5	ex 1.6	ex 1.7	ex 2.2	ex 2.3	ex 3.2
2010	523	513	503	493	383	516	493	476
2011	671	644	654	646	490	654	659	635
2012	589	565	569	567	428	571	579	579

year	ex 4.2	ex 4.3	ex 4.4	ex 5.1	ex 5.2	ex 8.3	ex 9.2	ex 9.4
2010	454	464	470	455	414	372	344	331
2011	634	621	581	581	632	93	518	431
2012	555	539	493	506	547	56	434	371

The major drop in the exercise 8.3 is caused by the course arrangements. In 2010 the exercise 8.3 was worth of 50 points. In 2011 and 2012 this exercise related to recursion was changed to be a voluntary exercise without any points because recursion was not a part of the course. Only the most motivated students have done the exercise in 2011 and 2012.

Appendix C

UUhistle exercises

Exercise 1.1 (animation)

```
marks = 200 + 250
euros = marks / 5.94573
```

Exercise 1.2: (VPS)

```
celsius = 100
fahrenheit = 1.8 * celsius + 32
```

Exercise 1.3: (VPS)

```
first = 10
second = -20
temp = first
first = second
second = temp
```

Exercise 1.4: (VPS)

```
first = 3
first = first + 1
first = 1 + first
first = first + first
```

Exercise 1.5: (animation)

```
name = raw_input('What is your name?')
print 'What a lovely name,', name
```

Exercise 1.6: (VPS)

```
celsius = raw_input('Enter temperature in Celsius:')
fahrenheit = 1.8 * float(celsius) + 32
print fahrenheit
```

Exercise 1.7: (VPS)

```
marks = float(raw_input('Enter amount in marks:'))
print 'It is', marks / 5.94573, 'euros.'
```

Exercise 2.1: (animation)

```
line = raw_input('Enter a number:')
number = float(line)
if number > 5000:
    print 'Quite a big fish!'
else:
    print 'Was that the biggest number you got?'
print 'The game is over.'
```

Exercise 2.2: (VPS)

```
divisor = 4
if divisor == 0:
    print 'Chuck Norris divides by zero, you do not.'
else:
    print 1000 / divisor
```

Exercise 2.3: (VPS)

```
years = 25
if years >= 0:
    adult = years >= 17
    if adult:
        print 'Adult: ', years - 18
    else:
        print 'Yet a child: ', 18 - years
    print adult
else:
    print 'One for the future.'
print years
```

Exercise 3.1: (animation)

```
REPEATS= 3
i = 0
sum = 0
while i < REPEATS:
    line = raw_input('Enter temperature:')
    sum = sum + float(line)
    i = i + 1
print 'Average:', sum / REPEATS
```

Exercise 3.2 (year 2010): (VPS)

```
i = 0
while i < 7:
    i = i + 2
    print i
    i = i + 1
print i
```

Exercise 3.2 (year 2011 and 2012): (VPS)

```
value = 1
print 'The first numbers in the series:'
while value < 5:
    print value
    value = value * 2
print 'The end!'
```

Exercise 4.1: (animation)

```
def greet(name, lucky_number):
    print 'Hi,', name
    return lucky_number + 1

result = greet('BB-Esa', 7)
print result * 5
print greet('Idols-Ari', 666)
```

Exercise 4.2: (VPS)

```
def ask_euros():
    line = raw_input('Enter euros:')
    return line

text = ask_euros()
print float(text)
```

Exercise 4.3: (VPS)

```
def calculate(first, second):
    return second * 2 + first

result = calculate(3, 2)
result = calculate(result, result + 1)
```

Exercise 4.4 (year 2010 and 2011): (VPS)

```
def calculate(first, second):
    result = first - second
    return result * result

def main():
    val1 = int(raw_input('Enter an integer:'))
    val2 = 10
    result = calculate(val1, val2)
    print result + 2

main()
```

Exercise 4.4 (year 2012): (VPS)

```
def calculate(first, second):
    intermediate = first - second
    return intermediate * intermediate

def main():
    val1 = int(raw_input('Enter an integer:'))
    val2 = 10
    intermediate = val1 + val2
    result = calculate(val1, val2) + intermediate
    print result

main()
```

Exercise 5.1 (year 2010): (VPS)

```
def info_ok(name, age):
    return len(name) >= 0 and age >= 0

def print_info(name, age):
    if not info_ok(name, age):
        return False
    print 'Name:', name
    print 'Age:', age
    return True

if print_info('Johan', 15):
    print print_info('Peewit', -1000)
else:
    print 'Serve you right!'
```

Exercise 5.1 (year 2011 and 2012): (VPS)

```
def info_ok(destination, distance):
    return len(place) > 0 and distance > 0

def send_package(destination, distance):
    if not info_ok(place, distance):
        return False
    print 'Destination:', destination
    print 'Distance:', distance
    return True

if send_package('Helsinki', 20):
    print 'Shipment successful!'
else:
    print 'Shipment failed!'
```

Exercise 5.2: (VPS)

```
def calculate(first, second):
    return second * 3 + first

def double(value):
    return value * 2

print double(double(5))
print calculate(double(3), 2 + calculate(5, 1))
```

Exercise 6.1 (year 2011 and 2012): (animation)

```
def find(source, results):
    for value in source:
      if value > 8:
        results.append(value)

def main():
    number_list = [2, 5, 4, 9, 16]
    number_list[1] = 6
    first = number_list[0]

    bigger_values = []
    find(number_list, bigger_values)

    if len(bigger_values) > 0:
      print 'Numbers bigger than 8:', bigger_values
    else:
      print 'There were not bigger numbers than 8.'

main()
```

Exercise 7.1 (year 2011 and 2012): (animation)

```
def add_numbers(phonebook):
    phonebook['Ville'] = '050-123456'
    phonebook['Matti'] = '040-765432'
    phonebook['Liisa'] = '045-132465'

def main():
    phonenumbers = {}
    add_numbers(phonenumbers)

    print 'End with an empty string.'
    print 'Remember capital letters!'

    name = raw_input('Whose number you want to get?')
    while name != '':

       if name in phonenumbers:
          number = phonenumbers[name]
          parts = numero.split('-')
          print 'Area code:', parts[0], 'Phone number:', parts[1]
       else:
          print 'The name', name, 'does not exist.'

       name = raw_input('Whose number you want to get?')

    print 'The program ends.'

main()
```

Exercise 8.1 (year 2011 and 2012): (animation)

```
f = open('stock.txt', 'r')

for line in f:
    line = line.rstrip()
    parts = line.split(';')
    count = int(parts[0])
    if count < 10:
       print parts[1]

f.close()
```

Exercise 8.2: (animation)

```
def count_sum(sum, left):
    if left > 0:
         newest = float(raw_input('Enter value:'))
         return count_sum(sum + newest, left - 1)
    else:
         return sum

print 'The sum is:', count_sum(0, 3)
```

Exercise 8.3: (VPS)

```
# Calcuates the factorial of a positive integer.
def factorial(n):
    if n < 3:
        return n
    else:
        return factorial(n-1) * n

print 'Result:', kertoma(5)
```

Exercise 9.1: (VPS)

```
car1 = Car(50)
car1.fuel(40)
liters = car1.fuel(60)
print liters
car1.drive(10)
print car.get_fuel()
car2 = Car(60)
car2.fuel(10)
```

Exercise 9.2: (VPS)

```
car1 = Car(45)
car1.fuel(15)
car2 = Car(60)
car3 = car1
car3.fuel(20)
print car.get_fuel()
print car3.get_fuel()
car3 = car2
print car3.get_fuel()
```

Exercise 9.3: (VPS)

```
class Car:
  def __init__(self, tank_size):
      self.__tank_size = tank_size
      self.__gas = 0

  def fuel(self, liters):
      added = min(liters, self.__tank_size - self.__gas)
      self.__gas = self.__gas + added
      return added

  def drive(self, consumption):
      if self.__gas < consumption:
          return False
      self.__gas = self.__gas - consumption
      return True

  def get_fuel(self):
      return self.__gas

car1 = Car(50)
car1.fuel(40)
liters = car1.fuel(60)
print liters
car1.drive(10)
print car1.get_fuel()

car2 = Car(60)
car2.fuel(10)
```

Exercise 9.4: (VPS)

```
class Person:
    def __init__(self, firstname, profession):
        self.__name = firstname
        self.__profession = profession

    def greet(self, other):
        return self.__name + '. Nice to meet you, ' + other.__name

    def get_profession(self):
        return self.__profession

first = Person('Babar', 'doctor')
print first.get_profession()
second = Person('Safiira', 'biologist')
print first.greet(second)
```