Exploring Programming Misconceptions

An Analysis of Student Mistakes in Visual Program Simulation Exercises

Teemu Sirkiä
Aalto University
teemu.sirkia@aalto.fi

Juha Sorva
Aalto University
juha.sorva@aalto.fi

ABSTRACT

Visual program simulation (VPS) is a form of interactive program visualization in which novice programmers practice tracing computer programs: using a graphical interface, they are expected to correctly indicate each consecutive stage in the execution of a given program. Naturally, students make mistakes during VPS; in this article, we report a study of such mistakes.

Visual program simulation tries to get students to act on their conceptions; a VPS-supporting software system may be built so that it reacts to student behaviors and provides feedback tailored to address suspected misconceptions. To focus our efforts in developing the feedback given by our VPS system, UUhistle, we wished to identify the most common mistakes that students make and to explore the reasons behind them. We analyzed the mistakes in over 24,000 student-submitted solutions to VPS assignments collected over three years. 26 mistakes stood out as relatively common and therefore worthy of particular attention. Some of the mistakes appear to be related to usability issues and others to known misconceptions about programming concepts; others still suggest previously unreported conceptual difficulties. Beyond helping us develop our visualization tool, our study lends tentative support to the claim that many VPS mistakes are linked to programming misconceptions and VPS logs can be a useful data source for studying students' understandings of CS1 content.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Human factors

Keywords

Introductory programming education, CS1, misconceptions, program visualization, visual program simulation

Koli Calling '12, November 15–18, Tahko, Finland

1. INTRODUCTION

In this article, we investigate beginner programming students' use of an educational program visualization tool. There are two primary motivations behind this work.

First, we agree with those within computing education research who urge us to study all aspects of the 'didactic triangle' formed by the learners, the teachers, and the content of teaching (including tools that express that content) [2]. We believe that in order to create an educational tool that is helpful, it is useful to consider the tool not only in the ideal form in which the teacher or tool author envisions it. We should also study the relationship between learners and the tool in actual practice.

Second, and more specifically, we wished to explore the potential of usage logs collected from a program visualization tool in investigating students' misconceptions of introductory programming concepts. The particular form of program visualization that we studied, visual program simulation, encourages students to act on their conceptions and misconceptions as they trace program execution graphically. Visual program simulation logs afford a look at the mistakes that students make, many of which presumably reflect their misconceptions. Studying these mistakes has the potential to improve our knowledge of programming misconceptions and to improve our visualization tools and their associated pedagogies.

The next section describes related prior work: research on misconceptions and visual program simulation. Section 3 lists our research questions and Section 4 explains how we went about answering them. Section 5 presents an overview of our results, which are then discussed in more detail in Section 6. In Section 7, we briefly discuss the quality of our research design and some limitations of the study. Finally, Section 8 summarizes our contributions.

The empirical study described in this article has also been reported by the first author in his recent master's thesis [23].

2. RELATED WORK

2.1 Misconceptions

Studies suggest that introductory programming courses (CS1 courses) are not very successful in teaching students about fundamental concepts, and that the problem is not limited to a single institution nor caused by the use of a particular programming language [7, 12]. Since the 1980s, researchers have catalogued a vast number of misconceptions about computer behavior and basic programming constructs (see [3, 27] for reviews of these studies). For instance, stu-

dents commonly consider classes to be containers for objects or think of assignment statements as mathematical equations. Misconceptions can cause students to flail about unproductively and to produce buggy programs.

Depending on one's theoretical viewpoint, one may regard misconceptions as raw materials that can evolve into better understandings or as errors that should be confronted and supplanted [25, 14, 4, 17]. Either way, it is useful to promote *cognitive conflict*, that is, dissonance between the learner's existing understanding and what they observe.

Within introductory programming, many misconceptions, if not most of them, have to do with things that are not readily visible, but hidden within the execution-time world of the computer: references, objects, automatic updates to loop control variables, and so forth. One way of fostering cognitive conflict is to make these aspects visible.

2.2 Educational Program Visualization

Program animation is a form of program visualization in which the computer displays the execution steps of a program to the user, who may control the pacing of the visualization and determine which aspects of execution are displayed. It is the most common form of program visualization in tools that are meant for CS1 students. (See [27] for a comparison of the many program visualization systems that have been built for beginner programmers.)

Although program animation is attractive and can work, computing educators are increasingly aware that just showing things to learners is often not particularly effective [11, 16]. Builders of educational software visualizations have consequently sought new ways of cognitively engaging learners with visualizations. For instance, learners may be asked to answer prediction questions related to a visualization (e.g., [15, 18]), or required to manipulate or construct a visualization. A part of this movement is the recent work on visual program simulation.

2.2.1 Visual Program Simulation

In *visual program simulation* (VPS), the learner takes on the role that program animation leaves to the computer [27]. A VPS exercise challenges the learner to think about how a program works and to demonstrate their understanding in a concrete way: the learner is expected read code, make control flow decisions, and keep track of program state as they trace a given program's execution step by step.

As a learner does a VPS exercise, they use a visualization of computer memory as an external aid. To enable convenient practice on multiple VPS problems, a supporting software system is useful if not critical. A VPS-supporting program visualization system can provide the graphical elements that the learner manipulates to produce a visual trace of program execution. A VPS system may also provide feedback and grade learners' solutions automatically, which is especially useful in large-class scenarios.

Visual program simulation is a program-reading activity that is intended to promote a dynamic view of programs, to improve students' conceptual understanding of programming and the so-called *notional machine* [5, 28] that runs them, and to provide practice with tracing programs. Ultimately, albeit indirectly, VPS is intended to contribute to the skill of writing programs.

One way in which visual program simulation differs from program animation is that it encourages learners to act on their conceptual understandings of program execution as they choose simulation steps. When a learner has a misconception, a program animation can only show the learner 'how it really goes' and hope that the learner is sufficiently motivated to study the visualization to experience cognitive conflict. A VPS system, on the other hand, can require the learner to actively find a correct solution that conflicts with their misconceptions, and may also diagnose learners' behavior in order to provide guidance that is tailored to address particular misconceptions. (For a more detailed discussion of interactivity in the context of educational program visualization, see [27].)

2.2.2 UUhistle, a VPS system

UUhistle (pronounced "whistle") [27, 29] is a program visualization system for introductory programming education, which supports several modes of user interaction, including visual program simulation. Figures 1 and 2 show how the system visualizes a notional machine for the Python programming language. Relevant areas of computer memory such as the heap, a call stack, and areas for evaluating expressions are displayed diagrammatically. In a VPS exercise, the learner drags, drops, and clicks on the visual elements to carry out execution steps such as assigning to variables, creating frames on the call stack, and so forth. Table 1 shows some examples of VPS steps and the corresponding GUI operations in UUhistle.

UUhistle automatically and immediately notifies the learner when they make a mistake. At the present time, most of UUhistle's feedback is generic (e.g., "wrong kind of step"). However, work is ongoing towards making UUhistle better at providing context-sensitive feedback that reflects the particular mistake that the student has made and the misconceptions that such a mistake may reflect (cf. Figure 2).

This brings us to our research questions.

3. RESEARCH QUESTIONS

By conducting this study, we wished to find out:

1. What are the most common mistakes that students make during visual program simulation exercises in UUhistle?

2. What underlying reasons plausibly explain these mistakes? In particular:

   - Do the mistakes suggest usability problems in UUhistle?
   - Are there plausible relationships between the mistakes and known programming misconceptions?
   - Do the mistakes suggest any other misconceptions?

Investigating these questions helps us to focus our efforts in optimizing UUhistle's usability and feedback. Moreover, answering these questions, however preliminarily, puts us in a better position to consider the potential of VPS logs in studying novices' misconceptions of programming. We wished to see whether we would find relationships between students' VPS behavior and the existing misconceptions literature, and to tentatively search the VPS logs for evidence of new kinds of misconceptions.

Table 1: Examples of visual program simulation.

Python code	Example VPS tasks	In the UUhistle v0.2 GUI
`a = 10`	Create the variable `a`. Assign to it.	Click on the top frame to create the variable. Drag `10` to it.[a]
`b = a + 1`	Evaluate the arithmetical expression. Create `b`. Assign to it.	Drag the value of `a` into the expression evaluation area within the top frame, then the plus operator and the literal `1`; click on the plus to produce the sum in the evaluation area. Create `b` as above. Drag the sum to it.
`square(b + 1)`	Access the function definition. Evaluate the parameter expression. Create a new frame on the stack and pass the parameter into it.	Drag the function element into the evaluation area. Evaluate `b + 1` as above (within the function element; cf. Figure 1). Click on the stack to create a frame. Click on the frame to create the appropriate variable; drag the parameter value from the lower frame to it.
`o = MyObj(100)`	Create an object of class `MyObj` and produce a reference to it. Call a method to initialize the object. Create the variable `o` and assign to it.	Click on the *Data in heap* panel to create the object; drag it to the evaluation area to produce a reference. Call the initializer (much like a function, above). Create `o` as above and drag the reference from evaluation area to it.
`if a > 100:`	Evaluate the conditional. Move to a different line as appropriate.	Evaluate `a > 100` like the arithmetical expressions above. Click on the line that will be executed next.

[a]The process of creating a variable has been streamlined in later versions of UUhistle so that binding an initial value to a name is a single atomic simulation step which creates a variable of the appropriate name.

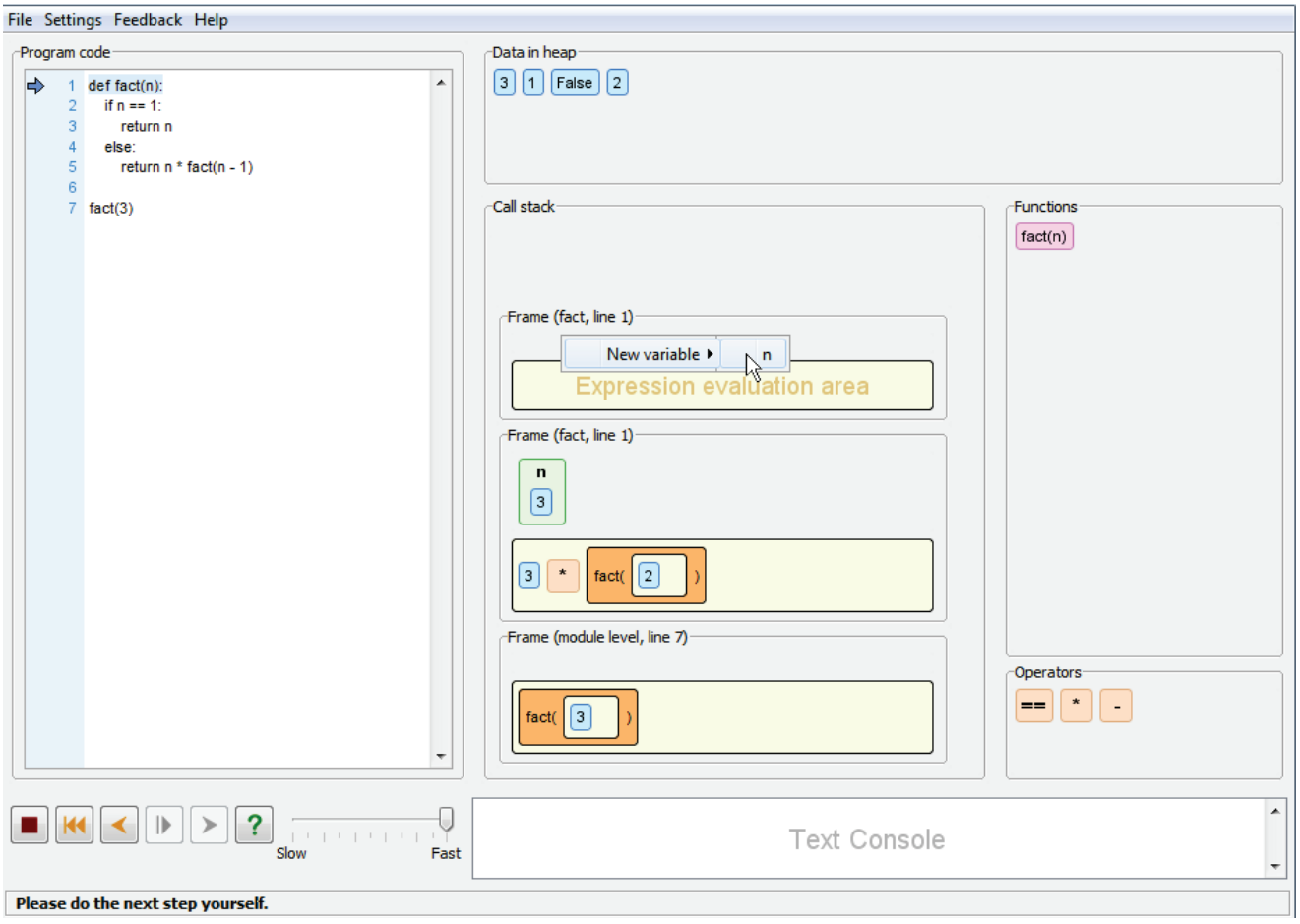

Figure 1: A visual program simulation exercise in UUhistle v0.2. The user, who is simulating the execution of a small recursive program, has just clicked on the top frame to create a variable there. After choosing the identifier `n` for the variable, he should drag the value 2 from the lower frame into the new variable.

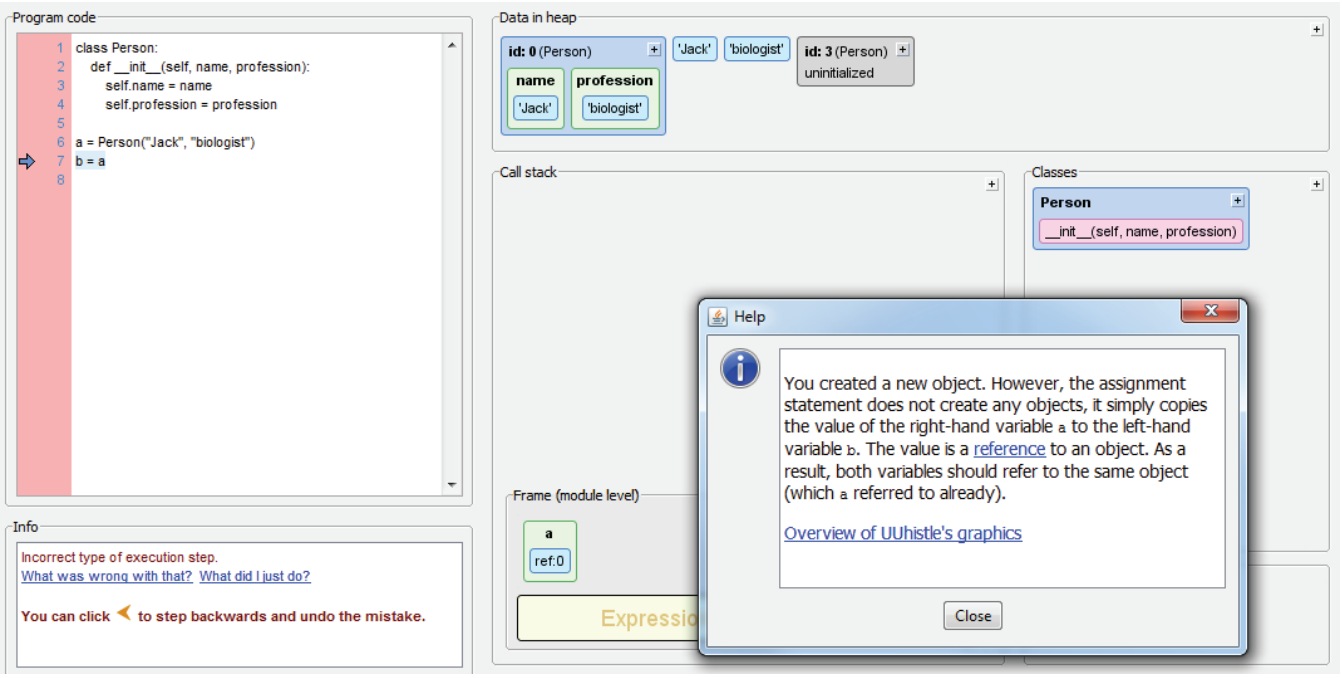

Figure 2: A VPS exercise in UUhistle v0.6. The user has created a new object when they should have formed another reference to an existing object. The Info box in the lower left-hand corner provides links to additional materials. Here, the user has clicked on "What was wrong with that?" to read UUhistle's feedback, which addresses a suspected misconception concerning object assignment.

4. METHOD

In this section, we outline the data we studied and the analysis process. The master's thesis that this article is based on provides more detail about aspects of our work and about the specific VPS exercises and versions of UUhistle that were used in the different course offerings that we studied [23].

4.1 Data Source

UUhistle's VPS exercises have been used since 2010 in one of the CS1 courses at Aalto University. This course is taught in Python, and is a fairly typical large-class CS1 on procedural programming; object-oriented concepts are touched upon near the end. 600 to 700 students enroll annually. The students do not major in Computer Science, but in various engineering disciplines, mostly Mechanical Engineering, Electrical Engineering, Transportation and Environmental Engineering, Energy and HVAC Technology and Chemical Technology. The majority have little to no previous programming experience.

Our data comes from students' solutions to VPS exercises submitted using various prototype versions of UUhistle between 2010 and 2012. These exercises, which covered topics such as assignment, selection, looping, references, and objects, were a part of the CS1 course alongside other types of assignments. Students were not strictly required to do the VPS exercises, but doing them contributed to their course grade and most students did participate.

Students submitted their solutions to online courseware. Each submission to a VPS exercise in UUhistle is a step-by-step log of a student's solution, including the missteps they have made. With three course offerings, a total of 18 different VPS exercises, and several hundred submissions per exercise, the total number of analyzed log files exceeds 24,000. Some log files were unusable due to technical complications in automatic data collection. Nevertheless, our analysis covers approximately 95 % of the all submissions made.

4.2 Analysis

The log files were first analyzed automatically using a custom-made computer program. Each log was compared against a file containing the model solution to the assignment. In this way, we acquired, for every stage of the model solution (which the students are expected to replicate) a listing of all the different steps that had been chosen by the students at that stage, and their frequencies.

It is possible for a student to take an incorrect step in UUhistle and then follow it up with another incorrect step. However, this is very infrequent in practice, since UUhistle instantly notifies the student of mistakes and encourages them to undo the incorrect step. We excluded such chains of incorrect steps from our analysis.

Students sometimes take a trial-and-error approach to VPS exercises [27]: mistake-undo-mistake-undo-mistake-etc. We thought that the first mistake made by a student at any given stage might be more indicative of their misconceptions than follow-up attempts, which might be more random. For this reason, we ran the analysis twice, including such post-undo attempts in one run, and excluding them in the other. As it turned out, while there was some variation in the frequency patterns produced by the two runs, the most common results were the same. The results reported below are from the run in which only the first attempt at each stage was counted.

The quantitative analysis, whose results appear in Section 5 below, identified the most frequent mistakes and answered our first research question. To answer our second question, we examined the more common mistakes more closely. For this part of the analysis, we used an informal method in which three researchers brainstormed for plausible explanations for the mistakes. We sought to relate the the common mistakes to the design of UUhistle's user interface, to known misconceptions of programming, to anecdotal evidence from the CS1 classroom, and anything else that seemed relevant. The main results of this process have been worked into the discussion in Section 6 below.

5. RESULTS OVERVIEW

The total number of simulation steps in each exercise set was about 700 (the exercise sets for the different years were nearly identical). Roughly 200 different mistakes were made by at least 10 students; mistakes that were made by fewer than 10 students we ignored entirely. 26 mistakes were identified as the most frequent ones and selected for further discussion.

We have listed the 26 most common mistakes, grouped thematically, in Tables 2 to 6. For each of the mistakes, the percentage of students who made that mistake is shown. For each exercise and each course offering, the total number of submissions (from which the percentages have been calculated) is in the hundreds; see [23]. The tables also provide fragments of the program code that the students were working on as they made the mistake, and very brief explanations of how the chosen VPS steps differed from the correct solutions. As the tables show, within this selection, too, there was considerable variation in the frequency of the mistakes, with some mistakes far more common than others.

6. A CLOSER LOOK AT SELECTED MISTAKES

In this section, we discuss in some detail what we consider to be plausible explanations for the relative commonness of some of the mistakes made by the students. We have grouped the mistakes into those which are suggestive of conceptual misunderstandings about the content of the visualization and those which are probably caused by UUhistle's user interface or other aspects of the VPS experience. Mistakes of the latter kind we would like to eliminate whenever possible. Mistakes of the former kind can be an important part of the learning experience, especially if the VPS tool is alert to them and provides useful guidance when the student makes them.

Our categorization is necessarily a simplification of a mesh of complex issues. Different students obviously will have different reasons for making the same mistake and many mistakes may be caused by a combination of usability issues and conceptual confusion. Sometimes students just misclick, too.

6.1 Misconception-Related Mistakes

We have grouped the potentially misconception-related mistakes into miscellaneous, function-related, and OOP-related mistakes. The emergence of functions and objects as major themes was not surprising: the implementation of functions and objects involves many 'hidden' aspects that UUhistle makes explicit; many misconceptions about these topics have been reported in the literature; and many students who answered a course-end survey have singled out functions and objects as the things UUhistle taught them most about [27].

6.1.1 Miscellaneous Basic Concepts: Assignment, Selection, Loops

Table 2 lists five common student mistakes that appear to be related to misconceptions about the fundamental programming concepts covered at the beginning of the CS1 we studied.

The first mistake listed, Misc1 (*Inverted assignment*), is familiar from the literature on misconceptions [e.g. 5, 19, 13, 22]. Research spanning several decades has shown that CS1 students struggle greatly – astonishingly much, many educators have complained – with the direction of the assignment statement (in many programming languages). Our data matches this observation. According to the VPS logs, students only make this kind of mistake in direct variable-to-variable assignment; when the right-hand side is a literal or a composite expression, we observed no analogous difficulty. Worryingly, the problem does not easily go away: anecdotal evidence from our CS1 classrooms suggests that our students often make the same mistake when they write programs of their own, even late into the course.

The next two mistakes, Misc2 *(Wrong branch)* and Misc3 *(Wrong False)*, in which students fail to react appropriately to the boolean value of a conditional expression, are difficult to diagnose reliably from our data. However, it is possible that they are linked to some of the misconceptions previously reported in the literature. Misc2 and Misc3 could both plausibly be committed by someone under the impression that 'then' clauses always get executed [24]. This hypothesis could be tested in the future by furnishing UUhistle with feedback that addresses this issue when students make this mistake. We expect that the commonness of Misc3 was moreover significantly affected by the learners being relatively inexperienced with the `not` operator and with conditional expressions that consist of a function call.

The mistake Misc4 *(Conditional into accessed variable)* appears to be easy to explain based on the literature. It is well documented that students confuse the `==` and `=` operators, which typically exhibits itself as overuse of `=` (in conditionals). Misc4 illustrates how students may also interpret `==` to indicate assignment. This is an another example of a mistake that would be easy to react to automatically in a VPS system by providing context-sensitive feedback that details the differences between the operators and provides links to further information about selection statements.

Despite the superficial similarity between Misc4 and Misc5 *(Conditional into loop control variable)*, we were more surprised to find that up to 8 % of students made the mistake Misc5, which to us seemed a rather exotic misstep. The mistake suggests a misconception regarding the role of variables in `while` loops, but our data does not lend itself to further explanation. This mistake is an interesting one that can be investigated further using other methods. We are puzzled as to why Misc5 did not occur at all during the 2011 offering.

6.1.2 Functions

Table 3 lists the most common mistakes which we interpreted as being related to misunderstandings of function calls.

Func1 corresponds to a family of misconceptions reported in the literature: it is thought that subroutines get executed

Table 2: Mistakes probably related to misconceptions about assignment, control structures, and/or booleans

Mistake	Percentage		
	2010	2011	2012
Misc1: Inverted assignment `first = second` *The student assigns the value of the left-hand variable to the right-hand side variable, rather than the other way around.*	27 %	26 %	38 %
Misc2: Wrong branch `if divisor == 0:` *Even though the conditional evaluates to* `False`*, the student jumps to the 'then' clause.*	11 %	17 %	17 %

Misc3: Wrong False
`if not information_ok(place, distance):`
`  return False`
As soon as the conditional evaluates to `False` *, the student proceeds to return* `False` *from the function.*
	52 %	35 %	45 %

Misc4: Conditional into accessed variable `if divisor == 0:` *After evaluating the expression, the student assigns its value to* `divisor`*.*	8 %	8 %	6 %
Misc5: Conditional into loop control variable `while i < 7:` *After evaluating the expression, the student assigns its value to* `i`*.*	8 %	0 %	3 %

Table 3: Mistakes probably related to misconceptions about functions

Mistake	Percentage		
	2010	2011	2012
Func1: Executing function instead of defining it `def ask_euros():` *The* `def` *command defines a new function: the student is supposed to store the function (object) in memory. Instead, the student starts executing the function body.*	17 %	15 %	16 %
Func2: Unevaluated parameters `result = calculate(result, result + 1)` *The student tries to start the function call before they have evaluated the sum.* *(The result from 2011 is missing due to a 'feature' in that year's version of UUhistle.)*	7 %	N/A	10 %
Func3: Parameter in the wrong frame `def calculate(first, second):` *The student creates parameter variables in the caller's frame, not in the callee's.*	6 %	7 %	11 %

Func4: Misplacing return value
`   return second * 2 + first`
# . . . 	
`result = calculate(3, 2)`
The student creates the variable `result` *in* `calculate`*'s frame instead of returning the value.*
	17 %	27 %	24 %

Func5: Return value into variable `return intermediate * intermediate` *The student assigns the result of the multiplication back into the variable* `intermediate`*.*	5 %	9 %	8 %
Func6: Failing to store return value `text = ask_euros()` *After returning a value, the student does not store assign the return value to the caller's local variable. Instead, they fetch the function call to the evaluation area again.*	20 %	16 %	29 %

as soon as program execution starts [24, 20]. Students' unfamiliarity with UUhistle's VPS exercises is likely to have contributed to the commonness of this mistake, but the mistake also appears in the later VPS exercises during which the simulation step for defining a function should be familiar.

The mistake Func2 *(Unevaluated parameters)* is suggestive of a misconception in which students think that unevaluated pieces of code get passed around as parameters (rather than their values). They may have a fragile grasp of the concept of expression evaluation. This interpretation matches some of the classroom experiences from the CS1 staff and is analogous to those reports in which objects are thought of as pieces of code [6] and assignment statements are thought of parts of equations [1, 26]. We feel that students' conceptions of expressions and evaluation are an important topic that merits further investigation.

The mistake Func3 *(Parameter in the wrong frame)* demonstrates that some students did not understand the purpose of a stack frame and how it relates to the variables that are available during a function call. This is consistent with the misconceptions of parameter passing that have been previously reported, in which students are confused about which

variables are accessible during which function call and about where the values of variables are stored in memory [8, 21]. The mistake Func4 *(Misplacing return value)* is similar, but involves a return value instead of parameters. (Parameter-passing-related difficulties have been further discussed in the context of UUhistle by the second author, who studied videos of students working on VPS problems [27].)

After the students returned a value, they often did not know what to do with it, as illustrated by mistake Func6 *(Failing to store return value)*. This mistake is possibly connected to the misunderstanding that a return value does not need to be stored even if one needs it later, which has been reported in the literature [10] and observed in our own classrooms. Students may think that they can use local variables of the called function even after returning.

6.1.3 Object-oriented programming

Table 4 lists mistakes related to object-oriented concepts.

The mistake OO1 *(Assignment copies object)* matches a well-documented student misconception according to which assigning objects to variables creates copies of them (even where references are actually used) [13, 26]. Both the literature and our experience from teaching introductory courses suggest that this misconception is very common. It is interesting to note that even though UUhistle's visualization prominently features object references, and even though the students were already used to dragging values from a variable to another when they simulate assignments, around 10 % of them still made this mistake.

The mistakes OO2 *(Failing to create second object of a class)* and OO3 *(Local variable becomes instance variable)* both seem likely to be related to the well-known and common difficulties that students have with understanding the relationship between classes and objects [9, 30, 20]. Students who make the mistake OO2 may believe that there can only be a single instance of each class, for example. OO3 appears related to the notion that a variable that references an object is a part of that object [9, 26, 21]. The mistake OO4 *(Instance variable becomes local variable)*, too, is suggestive of conceptual confusion between local and instance variables.

OO5 *(Method call without recipient)* is a common mistake that may be partially explained by students' unfamiliarity with objects: due to limited exposure to examples in which the order of evaluation is important, students fail to understand why an object reference is needed before the call can be formed. Students may intuitively feel that it is more 'logical' to first fetch the method, then the target object. (One of the students interviewed about UUhistle by the second author expressed this sentiment [27].) Python's explicit `self` parameters in method definitions may also contribute towards the commonness of this mistake.

6.2 VPS-Related Mistakes

Tables 5 and 6 list mistakes that have probably been mostly brought about by the students' unfamiliarity with UUhistle's user interface, unclear requirements in assignments, and/or usability problems.

6.2.1 Unfamiliarity with Requirements, VPS and UUhistle

The mistakes listed in Table 5 occurred either in the very first VPS exercises or when a novel concept first appeared in a an exercise. They tended to disappear later. We comment below only on a couple of cases that we have already addressed in UUhistle's design.

The mistake VPS1 *(Failing to assign)* featured only in the very first VPS exercise that the students did, and was clearly caused by the unfamiliarity of the tool and the requirements of a VPS exercise in UUhistle. Students who failed to pay sufficient attention to the instructions simply did not realize that they were expected not only to evaluate expressions but also to assign to variables. This analysis is supported by the fact that after the instructions for the first assignment were made clearer in 2011, this mistake does not appear in the subsequent logs.

The mistake VPS2 *(Non-standard order of evaluation)* was similarly reduced after instructions were improved. In some cases, this latter mistake was also brought about by students not always appreciating the fact that the order in which expressions are processed in the evaluation area is not arbitrary, and that UUhistle enforces a particular order that matches Python's notional machine. (This is evident from our related studies [27].) The newest versions of UUhistle give better feedback when a student does this.

6.2.2 Usability

The three common mistakes in Table 6 are ones that appear to be primarily related to UUhistle's usability. The most common and widespread mistake, which occurred across most of the VPS exercises in 2010 and 2011, is UI1*(Eager variable creation)*. This mistake has to do with the way UUhistle (like Python) treats variables as bindings of names to values: on the level of abstraction that is illustrated by UUhistle, a variable does not exist prior to being assigned an initial value. However, students tended to create the target variable of an assignment statement in advance (which would be acceptable if the notional machine we wished to visualize was a bit different). As this clearly resulted in considerable complications, we modified UUhistle so that creating a variable and assigning an initial value to it form a single atomic operation. (This had the added benefit of reducing the number of GUI operations needed for simulating assignment statements.) Making this mistake is therefore simply impossible in the most recent versions of UUhistle.

Students made the mistake UI2 *(Failing to jump upon selection)* because it was unclear to them what to do when they encounter a selection statement. While some occurrences of this mistake may be related to misconceptions and unclear requirements, we see this mostly as a flaw in usability. UUhistle tries to provide unintrusive signals (a flashing highlight and an Info-box text) to tell the student that they have to click the next line that will be executed, but it seems that students often did not notice these clues; presumably many thought instead that UUhistle takes care of all line changes automatically (as it does when there is no branching involved). Better guidance in the VPS exercises that feature selection is planned for the future.

There are likely to be different reasons behind UI3 *(Recalling a function)*, one of which is that UUhistle highlights an entire line of code at a time, and `def` lines get so highlighted in two different contexts: when defining a function and when setting up a function call. This can be confusing to the user. (Another reason is likely to be students' documented tendency to try to use visual elements that superficially resemble the current line of code [27].) One possible way of improving matters would be not to highlight the `def` line at

Table 4: Mistakes probably related to misconceptions about object-oriented topics

Mistake	Percentage		
	2010	2011	2012
OO1: Assignment copies object `car3 = car1` *The student creates a new object rather than copying a reference.*	12 %	6 %	14 %
OO2: Failing to create second object of a class `car2 = Car(60)` *A new object should be created, but the student instead makes a copy of a a reference to an existing object.*	16 %	7 %	18 %
OO3: Local variable becomes instance variable `car1 = Car(45)` *Instead of creating a new local variable to store the reference, the student creates an instance variable for the new object.*	7 %	10 %	11 %
OO4: Instance variable becomes local variable `self.__name = firstname` *Instead of creating a new instance variable for the object, the student creates a new local variable.*	8 %	7 %	3 %
OO5: Method call without recipient `car1.fuel(40)` *The student tries to call the method* `fuel` *from the* `Car` *class before accessing the variable for a reference to the target object.*	18 %	38 %	33 %
OO6: Assigning reference to uninitialized object `car1 = Car(45)` *The student assigns the reference to the variable before calling the* `__init__` *method.*	6 %	12 %	9 %
OO7: Returning self-reference instead of dereferencing it `return self.__profession` *The student returns* `self` *rather than the value of the instance variable* `__profession`*.*	6 %	15 %	15 %

Table 5: Mistakes probably related to unfamiliarity with VPS / UUhistle

Mistake	Percentage		
	2010	2011	2012

VPS1: Failing to assign
`celsius = 100`
`fahrenheit = 1.8 * celsius + 32`
In the very first VPS exercise:
Instead of assigning the value `100` *to the variable* `celsius`*, the student advances to the next line.*
	9 %	0 %	0 %

VPS2: Non-standard order of evaluation `fahrenheit = 1.8 * celsius + 32` (and similar expressions in other assignments) *The student tries to form the entire right-hand-side expression in the evaluation area. (Expected: multiply, then form the sum.)*	53 %	22 %	29 %
VPS3: Multiplying with a function call `fahrenheit = 1.8 * float(celsius) + 32` *The student tries to multiply the value* `1.8` *with the function* `float` *instead of evaluating the function call first.* *(Due to a technical issue, we have no data about this mistake from 2011/2012.)*	40 %	N/A	N/A
VPS4: Functions before literals `fahrenheit = 1.8 * float(celsius) + 32` *The student starts evaluating the expression by processing the function call (rather than the literal)* `1.8`*.*	9 %	22 %	21 %
VPS5: Inverse nesting `value = float(raw_input('Give a value:')))` *The student starts processing the line by fetching the* `raw_input` *function (which will indeed be* called *first), rather than first fetching* `float`*.*	20 %	17 %	13 %

all during calls (but to retain the highlight on the calling line during parameter passing).

7. LIMITATIONS OF THE STUDY

The validity of our study must be viewed critically. Our data is about student behavior, whose relationships to students' conceptions we may only conjecture and hypothesize about. However, as discussed above, there are what we perceive as credible links between our findings and the existing misconceptions literature. This strengthens our belief in the idea that VPS logs can contribute valid data about students' conceptual understandings. Triangulatory studies may be used in the future to test this claim.

We expect that some aspects of our results generalize to other contexts, but others are particular to our study. The distribution of student mistakes is affected by the background of the students, the pedagogy of the CS1 course, the particular VPS system we used, and the sequence of VPS exercises that the students did. The set of VPS exercises together with the system determines which mistakes it is even possi-

Table 6: Mistakes probably related to usability issues

Mistake	Percentage		
	2010	2011	2012
UI1: Eager variable creation `fahrenheit = 1.8 * celsius + 32` (and similar) *The student creates the target variable before evaluating the expression.* *This occurred in many assignments until a change was made to the system in 2012 (see Table 1 and its footnote).*	varies (but high)		0 %

UI2: Failing to jump upon selection
`if divisor == 0:`
`  print 'Chuck Norris divides by zero, you do not.'`
`else:`
`  print 1000 / divisor`
After evaluating the conditional, the student should click the line that control jumps to.
Instead, the student has started to execute a print statement without jumping.
	23 %	22 %	25 %

UI3: Re-calling a function
`def calculate(first, second):`
# . . . 	
`result = calculate(3, 2)`
Instead of passing parameters into the top frame they have just created, the student starts another call to `calculate` *within the top frame (cf. a recursive program).*
	29 %	33 %	45 %

ble to make. That being said, prior research indicates that learners have similar misconceptions of CS1 content across different contexts. If the same VPS exercises were used in a roughly similar setting, the list of the most common mistakes made would probably be similar. The forms of conceptual confusion that our study suggests presumably occur in other CS1 contexts as well, although we cannot comment on their commonness more generally.

8. CONCLUSIONS

In this article, we have explored the mistakes that students make in visual program simulation exercises within the UUhistle program visualization system. Students make a lot of different mistakes during VPS exercises, and knowing which ones are the most common allows us to focus our future efforts in developing the system. We identified 26 mistakes as relatively common; some of the mistakes appear to be related to usability issues and some others to known misconceptions about programming concepts. Together with related work [27], this study forms a part of an empirically-grounded tool development project in the context of CS1 education.

Visual program simulation is an unusual form of program visualization in that it tries to draw students to act on their conceptions. The present study lends tentative support to the claim that many VPS behaviors are linked to programming misconceptions.

Teachers, VPS systems, and computing education researchers can attempt to diagnose students' VPS behavior. Teachers may give feedback on the basis of students' mistakes. Similarly, developers of visualization tools may build VPS systems that are increasingly aware of misconceptions and provide guidance as signs of a possible misconception is apparent in a student's VPS trace. Researchers may find VPS logs useful as a data source that can be used to discover new misconceptions or to study the commonness of misconceptions within a population. Future studies might investigate, for instance, misconceptions of some less-studied CS1 topics, such as maps (dictionaries), exception handling, or first-class functions. Although VPS logs have their limitations, they are cheap and easy to collect in large quantities. Combining VPS logs with other data sources such as interviews seems a useful direction for future work: studying log files can produce hypotheses of misconceptions and suggest themes to be pursued in depth using other methods.

Our main motivation in doing this study has been to improve the feedback that UUhistle provides to learners when it suspects a misconception. In addition to it being useful to know what the most common student mistakes are and what reasons may lie behind them, we must consider the ways in which a visualization system may present the feedback to the learner. Tentative results from our research [23] agree with the general literature in that providing the option to read feedback (as UUhistle currently does) is not sufficient to get learners to reflect on their mistakes. In the future, we intend to explore other, more direct ways of presenting feedback on VPS mistakes as well as other pedagogical techniques that may help learners pay closer attention to the conceptual content of the visualization.

Acknowledgements

The authors would like to thank Lauri Malmi for supervising the first author's master's thesis, which this article is based on, and for participating in analyzing the data.

9. REFERENCES

[1] P. Bayman and R. E. Mayer. A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. *Communications of the ACM*, 26(9):677–679, 1983.

[2] A. Berglund and R. Lister. Introductory Programming and the Didactic Triangle. In T. Clear and J. Hamer, editors, *Twelfth Australasian Computing Education Conference (ACE 2010)*, volume 103 of *CRPIT*, pages 35–44. Australian Computer Society, 2010.

[3] M. Clancy. Misconceptions and Attitudes that Interfere with Learning to Program. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 85–100. Routledge, 2004.

[4] A. A. diSessa. A History of Conceptual Change Research: Threads and Fault Lines. In R. Sawyer, editor, *The Cambridge Handbook of the Learning Sciences*, pages 265–282. Cambridge University Press, 2006.

[5] B. du Boulay. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[6] A. Eckerdal and M. Thuné. Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory. *SIGCSE Bulletin*, 37(3):89–93, 2005.

[7] A. Elliott Tew. *Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner*. PhD thesis, School of Interactive Computing, Georgia Institute of Technology, 2010.

[8] A. E. Fleury. Parameter Passing: The Rules the Students Construct. *SIGCSE Bulletin*, 23(1):283–286, 1991.

[9] S. Holland, R. Griffiths, and M. Woodman. Avoiding Object Misconceptions. *SIGCSE Bulletin*, 29(1):131–134, 1997.

[10] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. *SIGCSE Bulletin*, 35(1):153–156, 2003.

[11] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[12] W. M. Kunkle. *The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts*. PhD thesis, Drexel University, 2010.

[13] L. Ma. *Investigating and Improving Novice Programmers' Mental Models of Programming Concepts*. PhD thesis, Department of Computer & Information Sciences, University of Strathclyde, 2007.

[14] F. Marton. Our Experience of the Physical World. *Cognition and Instruction*, 10(2):227–237, 1993.

[15] N. Myller. Automatic generation of prediction questions during program visualization. *Electronic Notes in Theoretical Computer Science*, 178:43–49, 2007.

[16] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bulletin*, 35(2):131–152, 2003.

[17] G. Özdemir and D. B. Clark. An Overview of Conceptual Change Theories. *Eurasia Journal of Mathematics, Science & Technology Education*, 3(4):351–361, 2007.

[18] A. Pears and M. Rogalli. mJeliot: A Tool for Enhanced Interactivity in Programming Instruction. In A. Korhonen and R. McCartney, editors, *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 10–15. ACM, 2011.

[19] R. T. Putnam, D. Sleeman, J. A. Baxter, and L. K. Kuspa. A Summary of Misconceptions of High School BASIC Programmers. *Journal of Educational Computing Research*, 2(4):459–72, 1986.

[20] N. Ragonis and M. Ben-Ari. A Long-Term Investigation of the Comprehension of OOP Concepts by Novices. *Computer Science Education*, 15(3):203 – 221, 2005.

[21] J. Sajaniemi, M. Kuittinen, and T. Tikansalo. A Study of the Development of Students' Visualizations of Program State during an Elementary Object-Oriented Programming Course. *Journal of Educational Resources in Computing*, 7(4):1–31, 2008.

[22] Simon. Assignment and Sequence: Why Some Students Can't Recognize a Simple Swap. In A. Korhonen and R. McCartney, editors, *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 16–22. ACM, 2011.

[23] T. Sirkiä. Recognizing Programming Misconceptions – An Analysis of the Data Collected from the UUhistle Program Simulation Tool. Master's thesis, Department of Computer Science and Engineering, Aalto University, 2012.

[24] D. Sleeman, R. T. Putnam, J. Baxter, and L. Kuspa. Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*, 2(1):5–23, 1986.

[25] J. P. Smith, III, A. A. diSessa, and J. Roschelle. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *Journal of the Learning Sciences*, 3(2):115–163, 1994.

[26] J. Sorva. Students' Understandings of Storing Objects. In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 127–135. Australian Computer Society, 2007.

[27] J. Sorva. *Visual Program Simulation in Introductory Programming Education*. Doctoral dissertation, Department of Computer Science and Engineering, Aalto University, 2012.

[28] J. Sorva. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education*, (accepted).

[29] J. Sorva and T. Sirkiä. UUhistle – A Program Visualization Tool for Introductory Programming Education (web site), n.d. `http://www.uuhistle.org/`.

[30] M. Teif and O. Hazzan. Partonomy and Taxonomy in Object-Oriented Thinking: Junior High School Students' Perceptions of Object-Oriented Basic Concepts. *SIGCSE Bulletin*, 38(4):55–60, 2006.