HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Juha Helminen

Jype – An Education-Oriented Integrated Program Visualization, Visual Debugging, and Programming Exercise Tool for Python

Master's Thesis

Espoo, April 23, 2009

Supervisor: Professor Lauri Malmi

Instructor: Docent Ari Korhonen

HELSINKI UNIVERSITY OF TECHNOLOGY Faculty of Information and Natural Sciences Degree Programme of Computer Science and Engineering		ABSTRACT OF MASTER'S THESIS
Author	Juha Helminen	Date 05.03.2009 Pages 84
Title of thesis	Jype – An Education-Oriented Integrated Program Visualization, Visual Debugging, and Programming Exercise Tool for Python	
Professorship	Software Technology	Code T-106
Supervisor	Professor Lauri Malmi	
Instructor	Docent Ari Korhonen	

Learning to program is difficult. In international studies, students have demonstrated a surprisingly definite lack of programming skills after passing their introductory programming courses. The underlying cause appears to be their fragile knowledge of elementary programming and overall insufficient understanding of control flow and program state.

Ultimately, learning to program requires practice. On large programming courses automatic assessment can be used to provide students individual feedback on programming exercises without sacrificing quantity. In addition, visualization is often used in a supporting role in order to facilitate learning of abstract concepts. However, existing research indicates that mere visualizations are not effective in learning but the learners must be engaged beyond passive viewing. Furthermore, visualization tools are typically separate from automatic assessment environments, which prevents students from easily utilizing them simultaneously. An integrated environment could be used to efficiently deliver focused assignments that target the observed deficiencies in elementary programming.

To address this, we developed a web-based easy-to-use tool for visualizing and debugging Python programs. The tool can be used to deliver automatically assessed Python programming exercises that are solved directly within the environment with the help of an integrated reversible debugger and automatically generated visualizations of program state. The tool was also integrated into the TRAKLA2 course management system where students' points and submissions can be recorded and tracked.

Keywords	program visualization, automatic assessment, computer science education, python, jython

TEKNILLINEN KORKEAKOULU Informaatio- ja luonnontieteiden tiedekunta Tietotekniikan koulutusohjelma	DIPLOMITYÖN TIIVISTELMÄ

Tekijä	Juha Helminen	Päiväys	05.03.2009
		Sivumäärä	84
Työn nimi	Jype – Integroitu visuaalinen virheenjäljitin, visualisointi- ja ohjelmointiharjoitustyökalu Python-ohjelmoinnin opetukseen		
Professuuri	Ohjelmistotekniikka	Koodi	T-106
Työn valvoja	Professori Lauri Malmi		
Työn ohjaaja	Dosentti Ari Korhonen		

Ohjelmoinnin oppiminen on vaikeaa. Kansainvälisissä tutkimuksissa on havaittu, että ensimmäiset ohjelmointikurssinsa suorittaneiden opiskelijoiden ohjelmointitaidoissa on suuria puutteita. Syy näyttäisi piilevän heidän epätarkoissa ja epätäydellisissä ohjelmoinnin alkeiden tiedoissa sekä kaiken kaikkiaan riittämättömässä ohjausvuon ja ohjelman tilan ymmärtämyksessä.

Viime kädessä ohjelmoinnin oppiminen vaatii harjoittelua. Suurilla ohjelmointikursseilla automaattista arviointia voidaan käyttää antamaan opiskelijoille henkilökohtaista palautetta ohjelmointiharjoituksista uhraamatta määrää. Lisäksi tukena käytetään usein visualisointia abstraktien käsitteiden oppimisen helpottamiseksi. Tutkimusten perusteella näyttäisi kuitenkin, että pelkät visualisaatiot eivät ole tehokkaita opetuksessa, vaan passiivisen katselun sijaan oppijat on aktivoitava. Lisäksi visualisointityökalut ovat tyypillisesti erillisiä automaattisen arvioinnin ympäristöistä, mikä estää opiskelijoita hyödyntämästä näitä helposti samanaikaisesti. Integroitua ympäristöä voitaisiin käyttää tehokkaasti alkeisohjelmoinnin puutteisiin kohdistuvien tehtävien teettämiseen.

Tämän johdosta kehitimme web-pohjaisen helppokäyttöisen ohjelmatyökalun Python-ohjelmien visualisointiin ja virheenjäljitykseen. Työkalua voidaan käyttää automaattisesti arvioitujen Python-ohjelmointiharjoitusten teettämiseen, jotka ratkaistaan suoraan järjestelmässä käyttäen apuna integroitua virheenjäljitintä, joka tukee myös koodissa taaksepäin askeltamista, sekä automaattisesti tuotettuja ohjelman tilan visualisaatioita. Työkalu integroitiin myös TRAKLA2-kurssinhallintajärjestelmään, jota voidaan käyttää opiskelijoiden pisteiden ja palautusten tallentamiseen sekä seurantaan.

Avainsanat	ohjelmien visualisointi, automaattinen arviointi, tietotekniikan opetus, python, jython

Acknowledgements

This thesis presents results on work carried out in the Software Visualization Group in the Department of Computer Science at the Helsinki University of Technology during years 2008 and 2009.

I am thankful to my supervisor Lauri Malmi and instructor Ari Korhonen for providing the opportunity to do this work and the guidance to carry it through. Additionally, I would like to extend my gratitude to Ville Karavirta, Otto Seppälä, Tapio Auvinen, Lasse Hakulinen, and the rest of SVG for their help, ideas, and comments. Furthermore, I thank Moti Ben-Ari, Mike Joy, Andrés Moreno, Klaus Müller, Tom Naps, Hermann Schloss, and Ángel Velázquez for their invaluable feedback on the finished software.

Much of this work is based on Matrix and Jython, the developers of which I would also like thank for creating these useful software libraries.

Finally, I wish to thank my family for the tremendous support over the years.

Otaniemi, April 23, 2009

Juha Helminen

Contents

Chapter 1

Introduction

1.1 Background

It is widely agreed that learning to program is extremely difficult. Beginning programmers tend to have serious difficulties in grasping the abstract concepts and notations that programming involves. However, as with any abstract constructs, we can use illustrations to try to convey information about them. Software is commonly visualized in an attempt to facilitate program comprehension and support software engineering activities.

The aim of software visualization is to enable making sense of software structure, dynamics and processes by delivering illustrative and intuitive graphical representations of software concepts and constructs. Depending on the specific task at hand the visualization can, for example, be at a higher level of abstraction with reduced details than the actual data, thereby making the information easier to navigate, analyze and absorb. Software visualization can help with software development activities in facilitating collaborative development, debugging and testing, and with software maintenance activities in facilitating fault and quality defect detection, refactoring, re-engineering, reverse engineering, software configuration management and code reviews. A significant application area of software visualization is programming education. There software visualization techniques are utilized to help students learn programming concepts and algorithms. This area of software visualization will be the focus of this thesis.

Educational software visualization tools are used to give students graphical representations of aspects of software systems that are inherently intangible and hidden from

them, such as, the exact control flow, side effects in expression evaluations and data dependencies. When students are given the ability to visually explore programs and algorithms we expect them to be able to better make sense of program executions and programming concepts. There is some controversy and conflicting research on whether visualizations, such as animations of program executions, actually improve learning. Studies indicate that the level of interactivity is a significant factor in the learning outcome when it comes to educational software visualization [53, 62, 96]. As opposed to passive animations, more engaging visualizations that activate the students appear to be more beneficial. This result has encouraged researchers to create systems that integrate software visualization and automatic assessment: students are given tasks related to a visualization and their answers are automatically evaluated for correctness to give them immediate feedback. This way, the systems are able to engage students more effectively. The assignments can, for example, be pop-up questions based on an algorithm animation [66, 95, 111, 112] or a code animation [80, 94], which then forces the students to really stop and think about what is happening, for example, by asking them to predict the next step in the algorithm or the value of a variable. Another example is automatically assessed *visual algorithm simulation* assignments, where students construct algorithm animations by simulating the steps of an algorithm's execution. In TRAKLA2 [89], students interact with visualizations of data structures by clicking and dragging various graphical representations of data, and the correctness is evaluated by comparing the data structures' state transitions against those created by running an actual implementation of the algorithm. In MA&DA [78] students invoke operations via pop-up menus which are then compared to the correct sequence of operations to evaluate correctness and give feedback. In PILOT [21] graph algorithms are simulated by clicking edges of graphs to express traversal.

1.2 Objectives

Currently, there are several educational tools available that employ software visualization and automatic assessment to complement the teaching of programming. They vary much in features and approaches. On the one hand, there are many systems that are geared to simply creating animations of program executions to provide insight into the inner workings of memory and data structures and, on the other hand, there are many systems built for automatic programming exercise assessment. Generally, to make good use of both of these technologies, students have to use two

separate tools. They might install a stand-alone application for visualizing their code, such as an education-oriented *integrated development environment* (IDE), and separately submit and get feedback on their answers over a web-based assessment system. This is especially burdensome if we specifically want to drill their elementary programming skills with many small assignments focusing on language constructs and concepts, control flow, data flow and algorithms. In addition, many program visualization systems are primarily intended for use on lectures or in lecture notes and cannot create visualizations directly from running actual code in a visual debugging manner, but require manually annotating the code with animation instructions. However, especially for the Java language, there are also implementations that create visualizations automatically or on-the-fly from source code [27, 86, 93, 100, 120].

The objective of this thesis is to build an engaging interactive learning tool for novice programmers at the CS1 and CS2 levels that focuses on unit-level Python programming and basic data structures and algorithms. The tool will incorporate and integrate automatic software visualization strategies and automated assessment in creating a programming exercise system intended for rehearsing basic program comprehension, tracing and writing skills. The assignments are to range from a few lines of code to a few dozen in size and they are meant to teach and test the understanding of basic control and data flow. As opposed to most existing systems the solving of exercises and debugging of erroneous programs is to be supported with a closely-integrated debugging and visualization functionality. Furthermore, the new tool is to be integrated into a course management system to be deployed on introductory programming courses.

Our hypothesis is that by providing transparency into the execution of programs through software visualization and practice material in the form of automatically assessed programming exercises, the tool will aid in developing beginning programmers' skills in basic program comprehension and prevent misconceptions related to control and data flow by facilitating the construction of a viable model of program execution.

1.3 Structure of the Thesis

The structure of the thesis is as follows. Chapter 2 gives an introduction to the software visualization and automatic assessment fields of research as they relate to the teaching of programming. We define the basic concepts and review common visualization and assessment techniques being employed in tools aimed at teaching and learning programming. In Chapter 3, we draw upon this analysis to come up

with a set of requirements for our system and describe the design goals, constraints and choices, as well as our argumentation for them. We also review some significant existing systems that partially meet our requirements. In Chapter 4, we discuss the implementation details of the system and in Chapter 5 the finalized system is evaluated against the set objectives. Finally, Chapter 6 concludes the thesis with a discussion of possible directions for future work and a summary.

Chapter 2

Programming Education

2.1 Introduction

This thesis builds on an extensive existing body of research on *computer science education* (CSE). In this chapter we review some of the most important results that motivate and form the basis of our research. As discussed in Chapter 1, we intend to build a software tool to support the teaching and learning of programming, which will be the focus of discussion. Furthermore, our specific scope is the visualization and automatic assessment of small programs that serve to exemplify elementary programming concepts and constructs and lay the foundation for learning to program on introductory programming courses.

2.1.1 Programming and Program Comprehension

Programming is a core competence in computer science (CS) and is usually the first step in a CS curriculum. At the heart of programming is the concept of *program comprehension*, which refers to the process of understanding programs and software. Program comprehension research is twofold in that it deals with the theories of how programmers comprehend software and with the tools and methods used to assist in comprehension-related tasks. In essence, what happens in programming is that a programmer constructs a *mental model*, an internal representation of their understanding about a program's intent, its data and execution, through some cognitive processes, the study of which is the center of program comprehension research. It is good to note that program comprehension in itself is not an end goal, but rather a necessary step in all programming activities. [130]

Programming is a difficult cognitive skill to learn. It requires knowledge and skills in many areas, such as the syntax, semantics and pragmatics of programming languages, creative problem solving, development environment and a multitude of software tools (*e.g.* IDE, compiler), algorithms and data structures, programming concepts and paradigms (*e.g.* object-orientation, functions, variables), program design and programming patterns. Above all, what beginning programmers require, is a conceptual understanding of the computer's execution model, the *notional machine*, that is implied by the programming language's constructs [38]. On introductory programming courses students are expected to acquire this type of new knowledge, strategies and practical skills in a time span of only a few months and as programming dominates the beginning of any CS curriculum this difficulty can also easily discourage learners from continuing in this field. Indeed, "none of these issues are entirely separable from the others, and much of the 'shock' ... of the first few encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once." [108] Consequently, many kinds of software tools have been built to aid in developing programming and program comprehension skills. Most of these incorporate some form of visualization in an attempt to better communicate the abstract notions and constructs inherent in programming, and in so doing improve and speed up program comprehension.

2.1.2 Educational Software Visualization

Software visualization (SV) is "the visualization of artifacts related to software and its development process" [35], and is used in the presentation, navigation and analysis of software systems. This wide definition includes, but is not restricted to, the visualization of program code and data, requirements and design documentation, source code changes, bug reports, software quality and other metrics, and testing results. Specific areas of SV, that have been widely applied to programming education, and are therefore most relevant to our discussion, are the branches of program and algorithm visualization. *Program visualization* (PV) refers to the visualization of the *source code*[1] and data of a program [105]. *Algorithm visualization* (AV), on the other hand, is understood to mean the visualization of algorithms and programs on a higher level of abstraction, that is, on a more conceptual level [105]. While intuitively program and algorithm visualizations seem like powerful teaching methods this is not necessarily true. Their effectiveness has been suggested to depend on the

[1] "Source code is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form." [16]

chosen level of abstraction and how clean the visual presentation is [109]. More recent research on educational AV also indicates that the degree of user interaction with the visualizations, *i.e.*, the learner involvement, is a major factor as far as actual learning is concerned. In an extensive study on existing research on algorithm visualization Hundhausen et al. [62] concluded that the most beneficial uses of AV are those that activate the student, for example, with questions or exercises, as opposed to plain viewing. They suggest that AV is most effective when used in a supporting role of some engaging activity. Naps et al. [96] go as far as to say that educational SV is of little value if it does not engage the students in an active learning activity.

2.1.3 Constructivism in Programming

In his articles [11, 12], Ben-Ari applies the dominant theory of cognitive learning, constructivism, to CSE, and concludes that "Given the central place of constructivist learning theory and its influence on pedagogy, computer science educators should ... analyze their educational proposals in terms of constructivism". According to *constructivism* a learner actively constructs knowledge instead of passively absorbing complete models of knowledge from lectures and books. In essence, the theory claims that "all learning involves the interpretation of phenomena, situations, and events, including classroom instruction, through the perspective of the learner's existing knowledge" [125]. In other words, a learner builds knowledge by combining observations with their pre-existing models. When applied to programming, this implies what is also intuitively sensible, that in the end, learning to program requires practice along with adequate guidance to avoid *misconceptions* and to keep learners on the right path to building a *viable model* of programming knowledge. To a large extent, rote learning is not possible and while programmers can arm themselves with an array of pattern approaches that can be applied in many situations, ultimately, each problem will have a unique solution composed of several basic building blocks. Learners must first gain an understanding of the basic mechanics and then practice logical reasoning by combining and applying their mental models to solve problems in a variety of contexts.

The models are viable if they prove "adequate in the contexts in which they were created" [141]. That is, they allow the learner to accurately and consistently explain the mechanics of the constructs. The constructivist view indeed suggests that as learners each construct their own meanings from the materials and instruction they receive, they frequently end up with misconceptions, and that "the goal of instruction

should be not to exchange misconceptions for expert concepts but to provide the experiential basis for complex and gradual processes of conceptual change" [125]. This indicates that a programming course must have many programming assignments to drill the students and let them properly evolve their knowledge, and also that when designing programming assignments, attention should be paid to that they contribute effectively to the learning of viable models of programming constructs.

2.1.4 Computer-Assisted Assessment

Typical tasks for a beginning programmer include writing, extending or modifying a simple program or piece of code. Going through and grading these students' submissions is a time-consuming and mostly monotonous endeavour which quickly becomes a major burden on the teaching staff with their often strict resource constraints. This is where *computer-assisted assessment* (computer aided assessment, CAA) comes into play. CAA refers to software solutions of fully or partially automating the assessment, feedback and grading of assignments [4]. With the help of *automatic assessment* (AA) of programming assignments, even the workload on large courses can be kept manageable while still providing students with a reasonable level of guidance in developing their skills through hands-on experience with practical programming tasks.

Next we will take a closer look at topics brought up in this introduction.

2.2 Difficulties of Novice Programmers

2.2.1 Students' Performance on Introductory Courses

High drop-out rates are not atypical on first programming courses. In a recent study on the reasons behind this problem among CS minors at their institution Kinnunen and Malmi [71] reported a rate of 26 percent on their course, and in general the rate at many institutions is estimated to be at 20–40 percent. In 2007 Bennedsen and Caspersen [14] carried out a survey to internationally quantify failure rates on introductory programming courses. Across the 65 institutions that provided data for the study the rate for CS1 was 33 percent. This is obviously a significant waste of resources if the students have to take the course over and over to be able to pass. So any means of combating this problem are definitely worth of research. Along the same lines, multi-national studies have found that even novice programmers that have

passed their introductory programming courses have great difficulty in implementing and understanding even the simplest of programs [124, 135]. In an ITiCSE[2] 2001 working group study by McCracken et al. [90], a sample of 216 students from 4 universities were assessed after taking their first CS courses. On average they scored a discouraging 22.89 points out of 110 according to the evaluation criteria designed to meet the learning objectives of a first programming course. The overall result of the experiment was that students definitely cannot program at the expected level of competency when they have completed their introductory programming courses.

A commonly suggested explanation for the weak performance is that the students lack adequate skills in problem-solving [108]. The McCracken group described five steps that take place in this process: (1) abstract the problem from its description, (2) generate sub-problems, (3) transform sub-problems into sub-solutions, (4) re-compose the sub-solutions into a working program and (5) evaluate and iterate. Another explanation is that they lack required knowledge and skills precursor to this process. An ITiCSE 2004 working group led by Lister [85] set out to find out if, in fact, the real root of the problem is students' *fragile knowledge* of basic programming constructs, such as arrays and recursion, and their consequent deficiencies in understanding code. Fragile knowledge means that while a student might possess the knowledge to answer direct questions about particular programming items he is not able to apply that knowledge in solving a problem on his own. "[This] may take a number of forms: missing (forgotten), inert (learned but not used), or misplaced (learned but used inappropriately) [knowledge]." [108] Indeed, the working group discovered that many students are hindered by their inability to trace and understand code. That is to say, much of the problem resides in their weak skills in program comprehension which then turns out to be an essential factor even when writing novel code from scratch. Intuitively it does make sense that the cognitive processes of program comprehension always interleave the creative activities. Lister et al. [85] go on to note that, while experts generally employ higher-level skills in the comprehension of real programs, even they resort to meticulously simulating the code if they are unable to grasp the program's behaviour otherwise. By higher-level skills they refer to, for example, utilizing domain knowledge or *beacons*, which are specific recognizable programming patterns, such as the three-step-swap shown in Figure 2.1 [34]. Consequently, they suggest that it might be appropriate to first teach beginning programming students to properly systematically trace code.

In their study of introductory programming course drop-outs, Kinnunen and Malmi

[2]Innovation and Technology in Computer Science Education, an international CSE conference.

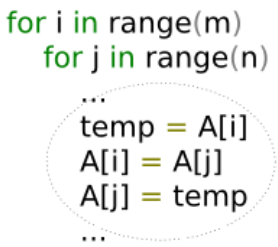

Figure 2.1: An example of a beacon. Instead of systematically reading the three lines to decipher their meaning, an expert will recognize the three-step pattern that is commonly used to swap the values of two variables.

reported that the students viewed finding run-time errors as the most difficult programming-related issue [71]. With run-time errors even experts tend to resort to systematically tracing code as a debugging strategy, so in light of the discussion above, it is not surprising that students find these particularly hard to debug. The severity of the issue that students find it hard to trace errors to their causes is demonstrated by Kinnunen and Malmi [70], who also reported that students named the difficulty of tracking down even simple errors as one of the reasons behind their decision to drop out. What discouraged them was that finding even a minor error could take hours. Based on the above discussion we can postulate that students are unable to find the errors because they lack the required precise understanding of basic control flow elements to be able to trace through their code. The results of a follow-up study to the Lister working group seem to support this theory. Based on the working group's data collected from the questions that required students to read code, Fitzgerald et al. [43] looked into novices' code tracing strategies. They found that students often – in principle – employed good strategies and most of them used walk-throughs, that is, systematic line-by-line tracing. However, their success varied greatly, which indicates a fragile knowledge of programming constructs. A related recent study on novices' debugging capabilities concluded that locating a bug is the real challenge in the debugging process [42], which is in accordance with earlier studies [69]. Fitzgerald [42] observed that once found, students were able to quickly fix bugs. The most challenging bugs to find were those related to loop conditions, conditional logic, arithmetic errors, and data initialization and updating. With regard to debugging strategies employed, the observations were similar to those of the code tracing study: there was no obvious correlation between the quality of strategies used and the success at debugging. So it appears to be more important how effectively the strategies are utilized. As an example of the ineffective use of a

strategy she explains how some students inserted print statements that printed the same fixed string in two different places, which can be accounted to an incomplete understanding of basic control and data flow.

2.2.2 Misconceptions in Programming

On the matter of what are the particular topics that students struggle with, the extensive literature review by Robins et al. [108] concludes that loops, conditionals, arrays and recursion are language features that are especially problematic for novice programmers. One example given of this is the failure to understand how the loop control variable is automatically incremented in a for-loop. Similarly, in a wide survey of educators about what they felt are the most difficult topics in CS1, in the category of general programming, the instructors mentioned such concepts as parameter passing, arrays, and recursion [30].

Underlying misconceptions about programming constructs are often offered as an explanation of students' difficulties. For example, students might apply the analogy of a box to variables and based on this metaphor believe that a variable may simultaneously contain many values [12]. However, the role of misconceptions is disputable.

Spohrer and Soloway [127, 128] conducted empirical studies with Pascal where they reported that bugs in students' programs are not primarily caused by misconceptions but instead stem from problems with plan composition. More recently, on courses taking the objects first approach, object-orientation-related misconceptions are not uncommon as discussed, for example, by Ragonis and Ben-Ari [106], and Sanders and Thomas [118]. In her studies of misconceptions in Pascal's parameter passing and in Java, Fleury [44, 45] noted that the incorrect self-invented rules constructed by the students predicted the behaviour as expected for a large set of programs. Their rules failed and the misconceptions became apparent only in some specific cases. Indeed, "most, if not all, commonly reported misconceptions represent knowledge that is functional but has been extended beyond its productive range of application" [125]. This brings up the point that even if misconceptions do not in typical cases manifest themselves as bugs, when they exist they are especially harmful. Eventually, when invoked, misconceptions result in bugs that are impossible to debug by the student because they follow from a fundamental flaw in their thinking. For example, in the working group report by Lister at al. [85], the second most difficult question for the students was the only one which involved invoking a return statement inside a loop.

They concluded this was due to students having a misconception about the semantics of return, that is, about the fact that the execution of the function would terminate immediately when the statement is run. In his analysis of constructivism within the context of CSE, Ben-Ari [12] expresses the same constructivist idea: "Teaching how to do a task can be successful initially, but eventually this knowledge will not be sufficient. ... The teacher must guide the student in the construction of a viable model so that new situations can be interpreted in terms of the model and correct responses formulated." Milne and Rowe [91] studied what are the most difficult topics in learning C++ programming, and equally stressed the importance of viable models. They concluded that they believe "the most difficult topics are so ranked because of the lack of understanding by the students of what happens in memory as their programs execute ... the students will struggle ... until they gain a clear mental model of how their program is working".

2.2.3 Supporting and Improving Program Comprehension

The implications of the discussion in this chapter are that, at least in the very beginning of learning to program, the hidden aspects of data and control flow should be made explicitly visible to the novices to let them properly learn to trace program state, to find errors effectively and to prevent fragile knowledge from evolving into misconceptions about basic programming constructs. This implies that the learning process should be supported with appropriate program visualizations. Also, while demonstrating programming principles in a passive fashion by explaining code on lectures and in textbooks might let the students absorb some structure of fragile knowledge of the programming constructs, this leaves space for fundamental misconceptions and generally is not enough to teach them to systematically apply this knowledge in tracing and writing code in practice on their own. We believe that to properly learn the basics of programming, students need to solve many small programming assignments that are purposefully designed to demonstrate different aspects of programming, and which require them to read, write and modify programs in a repetitive manner.

2.3 Program Visualization

2.3.1 Classifications of Software Visualization

Task-Oriented Taxonomy of Software Visualization

Based on the reference model of visualization given in Figure 2.2 adapted from Schneiderman et al. [123], Maletic et al. [87] define a task-oriented taxonomy of software visualization consisting of five different dimensions as listed below.

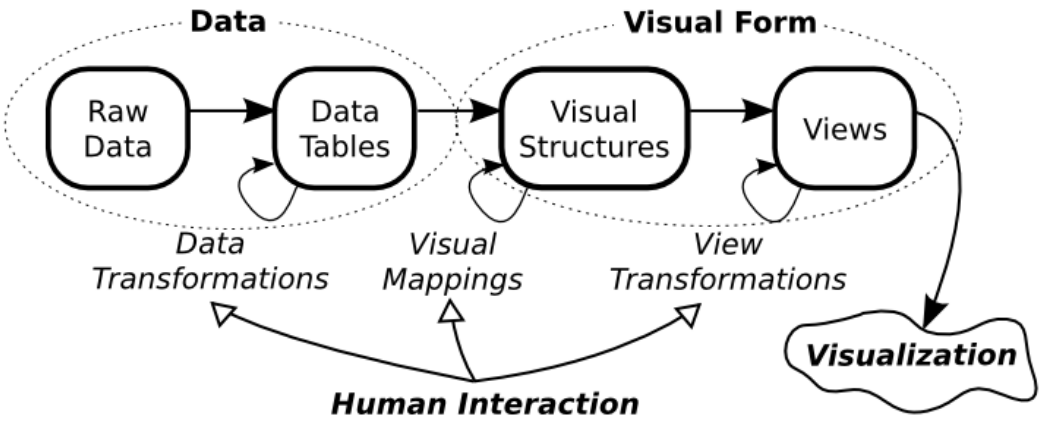

Figure 2.2: Reference model of visualization.

1. Tasks – why is the visualization needed?
2. Audience – who will use the visualization?
3. Target – what is the data source to represent?
4. Representation – how to represent it?
5. Medium – where to represent the visualization?

The model of visualization presented in Figure 2.2 defines visualization as a mapping of data to visual form that can also support interaction for adjusting the data processing step, the visual mappings, or the properties of the final visualization elements. This pipeline can be directly applied to software. Raw data is the source code, execution data or other software artifact. This data can directly or via transformations into

more abstract forms, such as parse trees or dependence graphs, be mapped into graphical elements through the use of selected *visual metaphors* to represent aspects of the data, where a visual metaphor is "an analogy which underlies a graphical representation of an abstract entity or concept with the goal of transferring properties from the domain of the graphical representation to that of the abstract entity or concept" [35]. The resulting visual elements are then combined into views of the data which can support visual navigation such as panning and zooming. For example, a metaphor for representing objects could be to show them as named rectangles and references between them as connecting lines. A view of a complete software would then be a graph of all the objects, which we might, for example, be able to scale to better tell apart the different components. [87]

A software visualization is then characterized in terms of the following five dimensions. The *task* dimension refers to what specific tasks are to be supported by the visualization. It could, for example, be meant to enhance understanding for simply educational purposes, debugging or reverse engineering. The *audience* dimension defines the attributes of the intended users of the visualization. The type of users, whether it be students, instructors or expert developers, has great significance in how the system should be designed: what kind of expertise do we expect the user to possess and how is this reflected in the type of visualizations and controls provided? The *target* dimension refers to what aspects of the data are to be presented. These can be attributes such as the architecture of a software system, the source code, an algorithm or software metrics. One of the most important elements of a software visualization is the choice of *representation*. This dimension defines how and what type of visualizations are derived from the target data to present the information. Finally, the *medium* dimension refers to the environment through which the visualization is made available, such as a single monitor, multiple monitors or a virtual reality environment. [87]

Price Taxonomy of Software Visualization

Another popular framework for describing software visualizations is that of Price et al. [105], in which visualizations are classified in terms of six basic categories as listed below. The taxonomy is somewhat outdated in that it is very much focused on the visualization of actual small programs and does not take into account all the aspects of SV that it has evolved to encompass since then, such as, visualizing software evolution (*e.g.* [24, 41, 40, 140]). Still, put together with the task-oriented

taxonomy it provides a good basis for explicating and evaluating different aspects of SV systems as it offers a somewhat different view.

1. Scope – What is the range of programs that the SV system may take as input for visualization?
2. Content – What subset of information about the software is visualized by the SV system?
3. Form – What are the characteristics of the output (the visualization) of the system?
4. Method – How is the visualization specified?
5. Interaction – How does the user of the SV system interact with and control it?
6. Effectiveness – How well does the system communicate information to the user?

Scope refers to the generality and scalability of the visualization system. That is, how wide range of programs in what environments and how large data sets can it display. For example, an educational visualization system aimed at beginning programmers might be restricted to a specific set of programming language constructs and few data items as opposed to arbitrarily complex software. *Content* defines what is visualized at what level of abstraction. For example, a system could be designed to visualize run-time behaviour at the conceptual level of showing line-by-line execution and the values of variables. The *form* category refers to the properties of the actual visualizations, such as, what medium is used, what types of graphical objects are there and how are they combined into views. *Method* category concerns itself with how the visualization is specified. In other words, is it automatically generated from program code or is there a specific language or library for creating it. *Interaction* defines what kind of controls there are to modify and navigate the visualizations. For example, there could be controls for spatial or temporal navigation of the visualizations. Finally, *effectiveness* refers to the system's ability to effectively convey the intended information and thus satisfy its purpose. [105]

As the learning of programming starts by observing the structure and flow of and by writing small programs, we can try to, and should as suggested by discussion in Section 2.2, facilitate understanding by visualizing the programs' structure and behaviour. As mentioned in the introduction of this chapter, this means using the closely-related methods of program and algorithm visualization. In terms of the

taxonomies described above, the target and content are the source code and behaviour of a program or algorithm, its data and control flow. The terms are not well-defined; Diehl [35] states that "this distinction is very blurry" and Price et al. [105] consider the definitions "ambiguous". However, in this work we regard the method category of the Price taxonomy as the differentiating factor. A program visualization is generated (automatically) from an actual written code of a program and as such the tools are also generally language-dependent, whereas with AV we do not base it on a plain program implementation of an algorithm but instead the visualization is described, for example, with an *algorithm animation language* [67] or as a code implementation that contains additional annotations for carrying out the visualization (*e.g.* [19, 25, 33, 129]). A type of annotation is providing special libraries with visualization capabilities that replace the standard ones to provide visualization somewhat transparently and automatically (*e.g.* [10, 75, 81, 131, 134]. These approaches are generally used to create more abstract visualizations which can, however, be linked to a visualization of an actual code or pseudo code implementation of the algorithm (*e.g.* [113]) or can be parametrized (*e.g.* [68]) what gives the impression of a real program visualization. Similarly, program visualizations can include elements typical of algorithm visualizations by providing visualizations of abstract concepts such as data structures that are identified by analyzing the program's data patterns (*e.g.* [27, 59]). We will focus on the techniques of program visualization, as we've defined it, because our intent is to visualize actual student-created code on-the-fly. In the teaching of programming, this type of visualization is made use of, for example, in novice-oriented integrated development environments (IDE) (*e.g.* [26]) and visual code tracing tools that are designed to support debugging or aid in demonstrating program behaviour on lectures (*e.g.* [80, 93]).

In terms of the representation dimension or the Price's method category, program visualizations can be divided into those created via *static analysis* and those generated through *dynamic analysis*. Dynamic analysis refers to the run-time inspection of a program as opposed to static, compile-time analysis [35]. With static program analysis techniques we analyze the source code so as to compute predictions on the set of values or behaviours that arise dynamically at run-time when executing the code [97]. The input data and environment of a particular execution is not and cannot be taken into account but the result must hold for all executions of the program, which means that approximations must be made [16]. In addition, based on static analysis we can visualize features of the code itself, and not the resulting program, such as the language's syntax and semantics. On the other hand, with dynamic methods we look to

give representations for the state and execution of running code, *i.e.* the realized data and control flow in one execution instance. The challenge in software visualization is devising metaphors that effectively map concepts to graphical representations. The aim is to "evoke mental images to better memorize concepts and to exploit analogies to better understand structures" [35]. Next we'll review representations that have been used to illustrate small-scale programs for educational purposes. Because of our scope we will not go into class-level and component-level architectural visualizations but restrict ourselves to techniques relevant to the visualization of the structure and execution of short pieces of code ranging from a few to a few dozen lines of code.

2.3.2 Static Analysis

As it's written in a formal language, a program's source code has a very specific structure and semantics that depend on the programming language. In static methods, based on this well-defined structure, the code is analyzed to visualize features of it and the program it describes.

A widely used visualization technique is *pretty printing*, which originally refers to "the use of spacing, indentation and layout to make source code easier to read in a structured language" [105] but nowadays attributes such as colors and fonts can also be adjusted. In essence, the code is parsed or matched against patterns defined with regular expressions to identify the syntactic or semantic roles of lexical components or groupings of components, which are then illustrated accordingly, for example, by showing string literals in a specific style, by indenting function bodies or other blocks of code in a certain manner or by graphically annotating specific fragments of code, such as beacons or other identifiable patterns. A ubiquitous form of pretty printing is *syntax highlighting*, where the keywords and literal values of the programming language are colored in distinctive ways to make it easier to discern syntactic constructs. The textual visualization can also be enhanced with visual navigation of code groupings in order to enhance legibility with large programs, for example, by using *code folding*. Figure 2.3 shows an example of pretty printing and code folding.

The second approach is to augment or replace the textual representation by carrying out program analyses, such as *control flow or data flow analysis*, to extract information about the defined program. In data flow analysis a program is thought of as a graph, where "the nodes are the elementary blocks (of the program) and the edges describe how control might pass from one elementary block to another" [97] and "the purpose

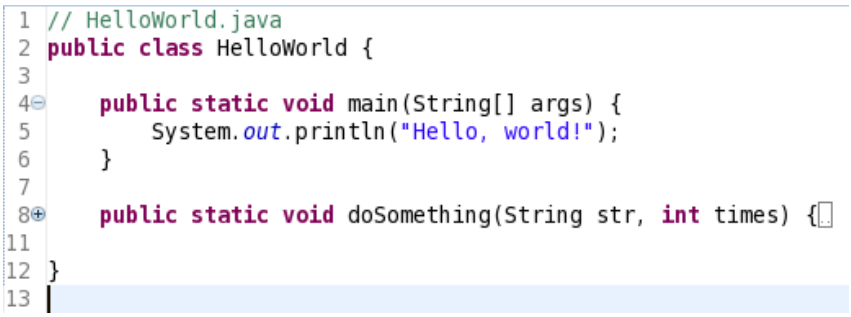

Figure 2.3: Java syntax higlighting in Eclipse (http://www.eclipse.org). Keywords, such as, "class" and visibility modifier "public", comments and the string literal are presented in different styles. The plus and minus signs next to the row numbers indicate that the "main" method is collapsed, *i.e.* fully visible, whereas "doSomething" has been folded and only its signature is shown.

of control flow analysis is to determine information about what elementary blocks may lead to what other elementary blocks" [97]. The results are typically shown in some diagrammatic graph-like form that attempts to illustrate data dependencies and possible execution paths. Examples of such visualizations are the *Jackson diagram*, *control-flow graphs*, *structograms* (Nassi-Schneiderman diagram) and *control-structure diagrams* [35]. Figures 2.4, 2.5, 2.6 and 2.7 show these diagrams when applied to a simple implementation of computing the factorial[3].

2.3.3 Dynamic Analysis

The state of a running program changes as the execution progresses so the visualization is not fixed. Therefore the visualization is typically an *animation*, that is, "a sequence of images which are shown one after another" [35]. The transition between images can be continuous, which is also referred to as smooth animation. Alternatively, information can either be accumulated or plotted along a time axis. For example, we might count the times a specific method is called or record what lines have been executed. The visualizations derived from dynamic analyses can be divided into two categories: data and code visualization. *Data visualization* focuses on memory contents, the data flow, and *code visualization* refers to the visualization of instruction executions, the control flow. [35] Figure 2.8 shows a screenshot of the ViLLE[4] visualization tool that features both data and code visualization.

[3]The code is constructed the way it is so that it features branching, looping and sequential execution. The examples are adapted from Diehl [35]

[4]http://ville.cs.hut.fi

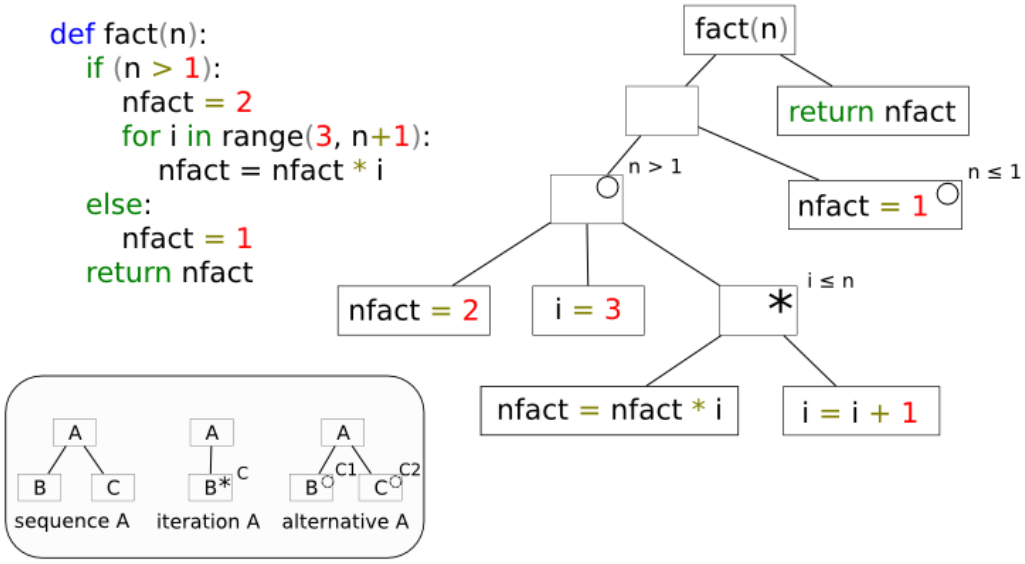

Figure 2.4: Jackson diagram of a factorial function. In the lower left corner are the building blocks of a Jackson diagram. The first diagram shows a sequence A that consists of actions B and C in that order. The second diagram shows an iteration A that repeats action B while condition C is true. The third diagram shows a branching construct A that consists of actions B and C which are taken according to their respective conditions C1 and C2.

Data Acquisition

An important aspect with dynamic analysis techniques is the *data acquisition* method, *i.e.*, how the run-time information about a program is extracted. Generally, the program is either *instrumented*, *i.e.*, additional instructions for recording the state are added within the original code (*e.g.* [86]), or the program is run in a special environment whose changes can be observed, such as a virtual machine or an interpreter which supports *reflection* well, where reflection is "the ability of a program to manipulate as data something representing the state of the program during its own execution" [18]. For example, the JIVE (Java Interactive Visualization Environment) visualization tool [47] collects information about the execution from the Java virtual machine via the Java Platform Debugger Architecture (JPDA)[5] application programming interface (API), and the Jeliot 3 [93] visualization tool executes programs with a Java interpreter, DynamicJava[6]. [35]

[5] http://java.sun.com/javase/technologies/core/toolsapis/jpda/
[6] http://koala.ilog.fr/djava/

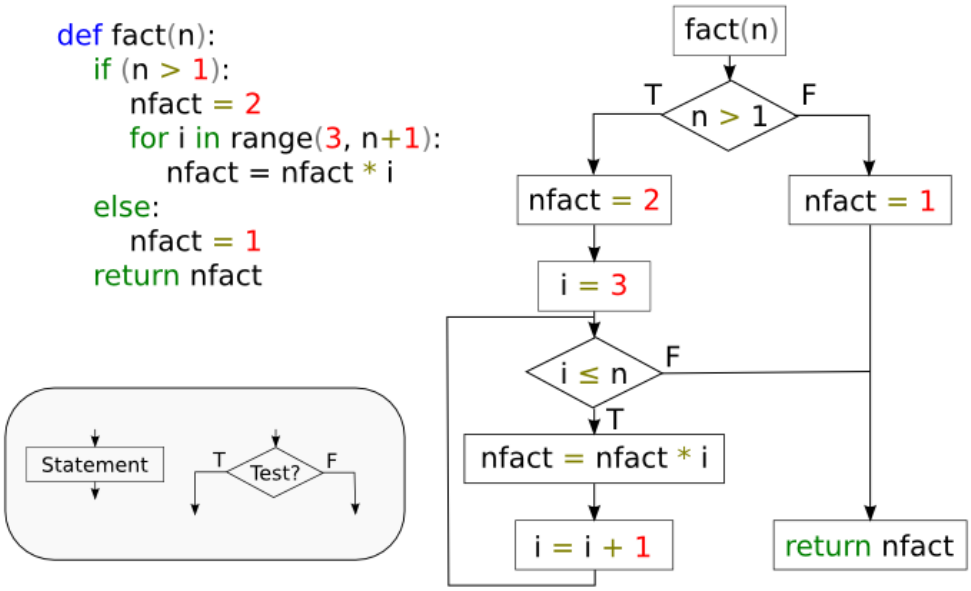

Figure 2.5: Control-flow graph of a factorial function. In the lower left corner are the building blocks of a control-flow graph. The first diagram represents a statement which can be combined into sequences by connecting them with arrows. The second diagram represents a branching conditions where T and F indicate the transition that follows from a true or false condition.

Visual Metaphors

The visual representations of a program's dynamic state are largely tool-specific and in Section 3.2 we give more concrete examples of existing solutions, whereas here we only examine some common approaches. In current algorithm and program visualization systems, the code, *i.e.* the progress of control flow, is typically visualized by simply highlighting the parts of the textual representation of the code's statements and expressions that have been or are being executed (*e.g.* [27, 80, 93, 139]), although all the diagrammatic illustrations discussed above would also be possible. With procedural languages we also usually have a visualization of the *execution stack* which is used to carry out function invocations. Program data is generally visualized as table-like views of variables' values or as a graph of data components where nodes are structures and programming constructs such as classes, objects and variables, and arcs represent connections and references between them. More conceptual visualizations of running programs that attempt to capture the same level of abstraction as typical in hand-crafted algorithm visualizations are also possible. These are accomplished either by traversing and analysing memory structures to identify specific patterns (*e.g.* [77])

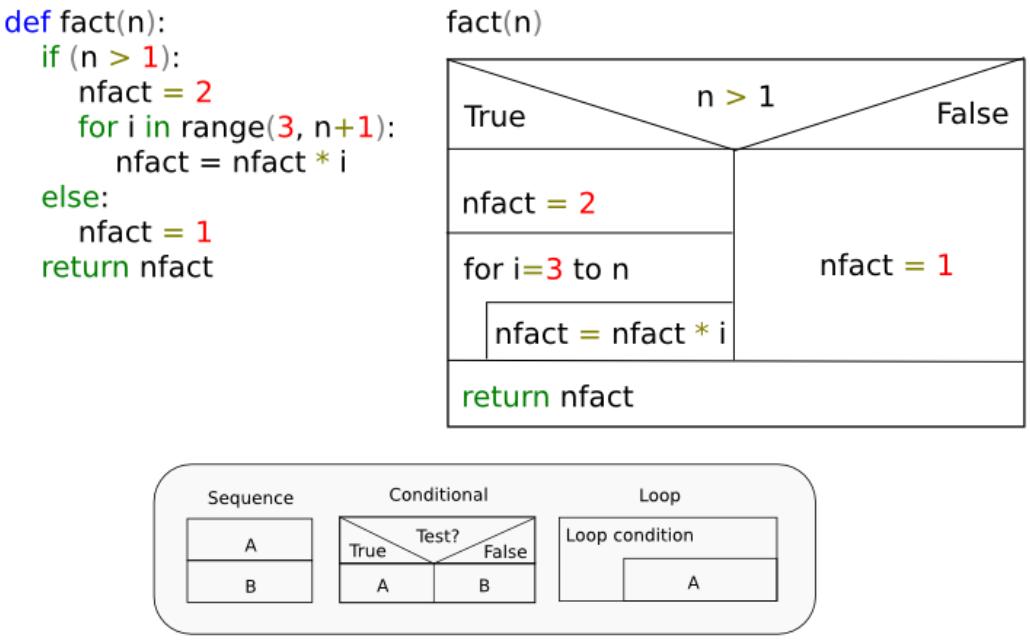

Figure 2.6: Structogram (Nassi-Shneiderman diagram) of a factorial function. In the bottom are the building blocks of structograms. The first diagram represents a sequence where A is before B. The second diagram shows a branching construct where the A branch is taken if the test is true and vice versa. The third diagram shows a looping construct where A is repeated as defined by the condition.

or simply by recognizing the use of specific interfaces, classes or modules for which the system can provide abstract visualizations (*e.g.* [57]). In algorithm visualization an algorithm's execution is typically visualized as an *algorithm animation* where the chosen sequence of execution states, the *interesting events*, is mapped to appropriate images of data structures creating a visual representation of how the algorithm works [35]. Program visualization tools that dynamically inspect run-time structures to identify data structures can seemingly create algorithm animations on-the-fly with no extra annotation or other visualization instructions.

Visual Debuggers

An important application of dynamically generated program visualizations are *visual debuggers*. These are tools that "reflect code-level aspects of program behavior, showing execution proceeding statement by statement and visualizing the stack frame and the contents of variables" and "are directed more toward program development

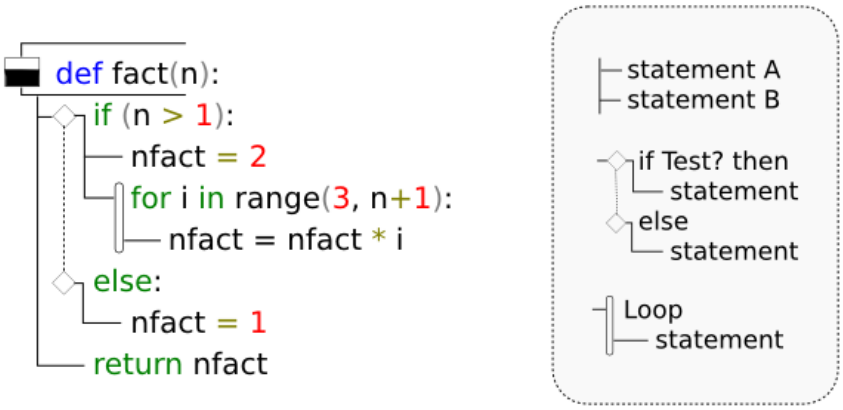

Figure 2.7: Control-structure diagram (CSD) of a factorial function. On the right are some building blocks of CSDs. The first diagram represents a sequence where B is executed after A. The second diagram shows a branching construct. The third diagram represents an iteration where the statement is repeated as defined by the looping condition.

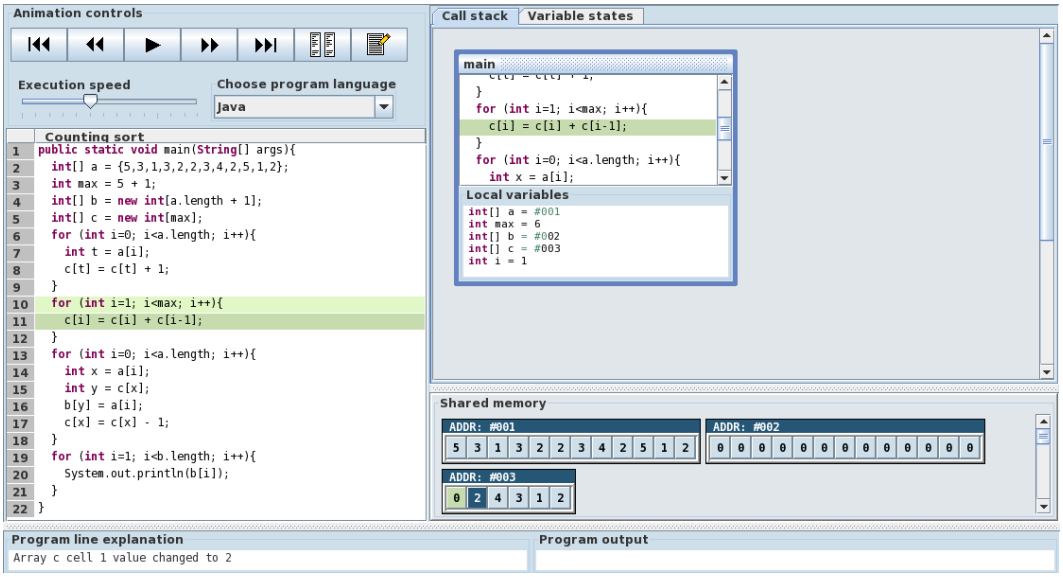

Figure 2.8: A screenshot of the ViLLE visualization tool. On the left is the primary code visualization where the next and the previously executed lines are highlighted. On the right there is a call stack visualization showing the values of local variables and below that there are visualizations of arrays.

rather than understanding program behavior". [103] In general, the goal of the process of debugging is "to detect the existence of errors in a program, to locate their position or cause, and, finally fix them" [35] and in visual debugging, graphical

representations of data and execution are used to help locate errors by allowing visual observation, exploration, and navigation of program state. These types of tools typically let the programmer execute software in steps while allowing them to easily follow the flow of data. Figure 2.9 has a screenshot of a typical visual debugger. It shows a Pydev[7] debugging session in the Eclipse IDE[8].

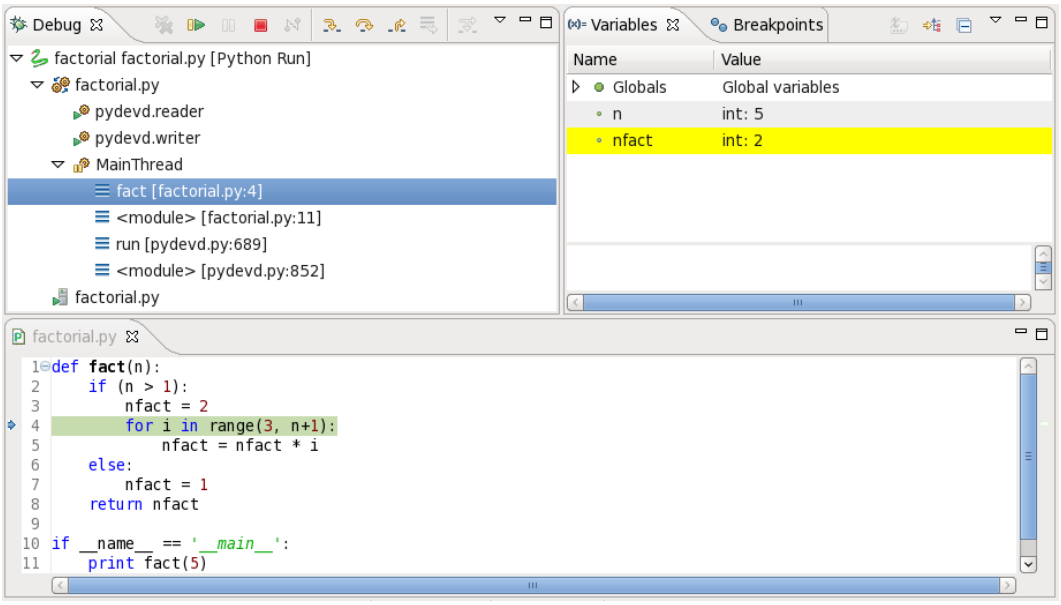

Figure 2.9: A Pydev debugging session in Eclipse. On the left we have representations of the execution threads. The states of variables are shown on the right. The yellow highlighting indicates that the previously executed line changed the value of the variable nfact. The code is on the bottom. The line that will be executed next is highlighted in green.

2.4 Automatic Assessment

Traditionally, the limiting factor in the number of actual programming assignments given to students on a course has been the amount of resources available to manually assess them. Because of this there usually are not that many of them. Of course, we can simply provide optional exercises that are voluntary to complete and are not checked by the course staff. However, the applicability of this approach is very limited. In their discussion of assessing programming assignments on large courses, Alamutka and Järvinen [3] note the tendency of students to try to minimize their workload, which

[7] http://pydev.sourceforge.net
[8] http://www.eclipse.org

renders optional activities in the learning process ineffective. Woit and Mason [144] carried out a five year study comparing different assignment strategies, where two experiments contained optional assignments. Along the same lines, they reported that their students mostly ignored optional tasks, which was also directly reflected in poor midterm exam results. In other words, the programming exercises must be mandatory to really make a difference, in which case we also need a way of, at least on some level, checking that the result is acceptable. In computer assisted assessment (CAA) the evaluation process is supported with software tools that fully or partially automate the tasks involved in order to reduce effort and speed up the process. Examples vary from facilitating rapid and consistent grading with feedback authoring tools (*e.g.* [1]) to *intelligent tutoring systems* (ITS, *e.g.* [5, 92, 133]) that simultaneously monitor and model the progress of and guide and give intelligent feedback on the student's learning.

2.4.1 Assessing Features of Programs

Generally, the assessment of programming assignments is based on the end artifact, the resulting programs and documentation, instead of, for example, the process observed while the student carries out the programming task. We require correct functionality, suitable design and good programming style. Typically, with automatic methods the *correctness* of the program, that is, its adherence to a given functional specification, is evaluated by examining the program's dynamic behaviour. In addition, we can also target many other non-functional properties and features of the code and the program it describes, such as, readability, efficiency and general programming style. For example, the program could be subjected to an analysis designed to uncover unnecessary uses of global variables, poor naming of identifiers or improper indentation. However, many of these properties are not readily assessable with automatic methods because they cannot be easily defined formally and require higher level reasoning or are a matter of taste, such as, the sufficiency and quality of comments in a program or the idiomatic use of language constructs and libraries. Thus, often only correctness is tested. In Section 2.2 we made the case that our tool must support the delivery of many short programming assignments. Indeed, especially with small assignments, where we only look to teach elementary programming, we can settle for assessing correctness.

2.4.2 Approaches to Evaluating the Correctness of a Program

Early automatic assessment systems were based on special tools, such as, modified versions of compilers and operating systems, whereas in all current systems solutions are submitted over the Internet via a web-based interface to a remote machine for assessment [37]. There are two primary approaches of automating the assessment of the correctness of programming assignments, for which we will use the terms program *output comparison* and *unit test based* assessment. In the first approach the functionality is checked by running the student's code with test data and comparing its output to the expected output of a working model solution (*e.g.* TRY [107], ASSYST [63], Goblin[9], RoboProf [32], BOSS [65], CourseMarker [60], Online Judge [23]). A typical way of implementing this is requiring the student to print computation results onto the standard output stream in a pre-specified format, and when the code is executed the produced character strings are matched against the set of expected output. In the other approach we drill the student code in a unit testing manner by calling parts of the code and comparing the results to the expected behaviour (*e.g.* Scheme-robo [114], BOSS [65], Ludwig [121], Javala [83], WebTasks [110]). In a sense, these correspond to the *black box* and *white box* (glass box) approaches in testing. In black box testing the test object is viewed as a black box that transforms input to output. We have no knowledge of the inner workings and testing is performed via examining the output generated in response to given valid and invalid input. In white box testing the tests are designed to exercise specific paths of the program fully aware of the internal structure. Therefore, white box testing can lead to more comprehensive tests but at the same time places restrictions on any later code restructuring or other changes. Similarly, the output comparison method allows for somewhat more freedom for students in the implementation details. Typically, each test data or test case set is designed to verify a specific aspect of the functionality, which is then expressed as feedback if a test fails. Similarly, grading or scoring can be based on the number of passed tests, which should reflect the extent of correctly implemented functionality.

Web-CAT [39] takes a somewhat different approach from the other systems. It implements a test-driven approach for automatic assessment. Students both write a program and unit tests for it themselves. The grade is based on three factors: test validity, test coverage and test results. Their tests are run against a reference implementation to assess validity, which is done by comparing the correct and

[9] http://goblin.tkk.fi/goblin/

students' expected results, and completeness, which is estimated with a test coverage analysis. Then the tests are run against their own implementation. The final grade is a combination of these results. The reasoning is that if the tests are valid and complete then their program must also be correct if the tests succeed.

2.4.3 The Pros and Cons of Automatic Assessment

The advantages of fully automated web-based assessment systems are immediate feedback, the availability of the service regardless of time and place and the objectivity and consistency of assessment. Furthermore, teachers can concentrate on aspects of the curriculum that most require direct interaction with students as opposed to the tedious task of scanning through hordes of more or less trivial programming submissions. Besides its immediacy, automated assessment and feedback may also have other less apparent benefits. Odekirk-Hash and Zachary [99] carried out an experiment where one group of students did programming exercises with a tutoring tool and another group did the same exercises without the help of automatic assessment. Both groups had access to teaching assistants (TA). They observed that while both groups performed more or less equivalently in terms of spent time and in a subsequent test, those who received automatic feedback spent much less time asking questions from the TAs. This implies that automatic assessment can reduce the need for contact learning and hence in tandem with the freedom of time and place of a web-based solution facilitate *distance learning* well. A related study comparing classroom exercises to automatically assessed exercises had similar results [74]. Also, TAs have more time for struggling students. In the discussion of their automated tools, Ala-Mutka and Järvinen [3] also note that a specific benefit of any, even rudimentary automatic assessment of programming assignments, is checking the format and contents of a submission. That is, even if the system may not flawlessly evaluate the correctness or quality of the solution, and some aspects have to be evaluated or proper feedback has to be compiled manually, there are significant gains with AA. They state that previously when submissions were done directly by e-mail 20 percent of all submissions had basic problems with content and form, such as, the program failing to compile.

An important intrinsic property of automatic assessment is that in order to programmatically implement automatic evaluation of an assignment the requirements must be formalized, which puts emphasis on proper design of assignments. So in theory, we come up with better thought out descriptions and specifications for the exercises. During the specification process, we might even gain more insight into the

the pedagogic and cognitive aspects of the assignment. Furthermore, formal requirements result in consistent and impartial assessment, whereas manual graders often have differing views on good programming style and, more importantly, interpret evaluation criteria differently no matter how exact and unambiguous they might be according to one person.

Current AA systems invariably also incorporate submission management utilities that enable the tracking, monitoring and reporting of students' progress with regard to the assignments [37]. This allows instructors to early identify students that are having problems or especially difficult subjects, and take action as needed.

Another didactic factor in the use of AA systems is the selection of an assessment policy, that is, how many times an assignment may be submitted for evaluation and how does this affect possible grading. This decision greatly affects the working strategies that students adopt with AA systems. An argument for allowing a submission-evaluation type of *formative feedback* loop with unlimited or lots of available submissions instead of one-time *summative feedback* is that this lets a student incrementally improve their solution and, hopefully, to eventually come up with a correct mental model of the problem, thus, supporting constructivist learning. Students also get more individual feedback than is otherwise possible. The flip side is that this type of instant feedback may encourage them to submit often and with less consideration to the correctness of their solution, leading to a trial-and-error interaction with the system. Ala-Mutka and Järvinen [3] mention one approach to combating this problem: in order to force students to analyze their errors their systems only approximately point out the problem in a solution. This is obviously an issue of balance: how do we construct the feedback in such a way that it guides the students in a formative manner in the right direction and prevents them from getting stuck, while at the same time does not reveal too much and that way let them pass the evaluation with the proposed fix without really fully understanding the problem themselves. In addition to limiting the number of submissions, the issue can be mitigated by only allowing a resubmission after some minimum time has passed since the last one. The reasoning is that the student is forced to and will use this time to better understand the problem and the probable fix. However, the downside is that if the student is able to quickly come up with the correct solution based on the feedback he still has to wait for the timer to run out. Depending on the submission policy the use of AA may also have positive effects on student motivation. Malmi et al. [88] analyzed statistics on submissions of their automatically assessed algorithm simulation exercises during 9 years and found that while it appears most

students are not prepared to do extra work if this does not directly affect their course grade, there seem to be fairly many students that are willing to do more in order to score maximum points for exercises even if this does not increase their grade.

The major weaknesses of using automatic assessment are the generally low quality of feedback and strictly analytical grading. These stem from the fact that it is difficult to analyze the students' errors automatically in such a way that we are able to identify misconceptions, the root causes of their errors, and grade incomplete or erroneous submissions on a scale relative to their degree of correctness. On the other hand, on large courses where the teaching staff systematically uses rubrics in order to carry out consistent and objective mass grading of programming assignments, the quality of feedback can easily degrade in a similar fashion, as individual feedback is strictly constructed from a prearranged set of replies.

Another special issue with all systems that execute *foreign code*, that is, outside code not included by the system developers, is security. This was noted early on in the research of AA: Hollingsworth [61] reported that a problem with their automatic grader for programs written on punching cards was that the student programs could modify the grader itself which they then had to try to detect. Indeed, when run to evaluate dynamic behaviour, a student's submission may inadvertently or intentionally cause undesirable effects in the AA system if it has not been carefully designed to address this problem. Damage can range from disclosure of private data to completely bringing down the system.

Plagiarism

Finally, a significant problem with all non-supervised electronic submission systems is of course *plagiarism*, that is, passing somebody else's work as your own. Many studies have shown that cheating is quite commonplace among university students and this is something that must be taken into account in the design of an AA system. For example, Sheard et al. [122] surveyed two universities and 34%/28% of the respondents admitted that they had copied a majority of an assignment from a friend and 53%/42% had worked collaboratively on tasks intended individual. When we are not in full control of the environment where the exercises are done, this can never be fully solved but there are some ways to discourage this type of behaviour. For example, Shaffer [121] explains that in their online system programming assignments are meant to be written using their text editor which is implemented as a Java applet. The applet does not allow the student to paste data from the clipboard in the editor,

which is intended to restrict their ability to easily copy solutions.

While we cannot prevent someone from copying a solution by retyping it, we can try to detect plagiarism. The primary approach to detecting program copies is *pairwise comparison* [82]. The pool of programs is analyzed in pairs to find commonalities in them. For each pair we get an estimate of the degree of similarity between the programs. Those pairs that much resemble each other in terms of the properties we assume to define the similarity of programs are potential instances of plagiarism. The comparison can make use of various types of software properties. Early systems based the comparison on simple style and software metrics, such as, the number of assignment or loop statements, or the size of a program (*e.g.* [15, 36, 52, 101]). For each program we get a tuple of these values called a feature vector the closeness of which in an n-dimensional space indicates similarity. The problem with even complex such numerical metrics is that most of the structural information is inevitably lost and, in fact, these types of metrics have been shown to be ineffective [82]. Obviously, another straighforward approach is to carry out different types of pattern and string matching on the character data. For example, in addition to exact duplicates, the Dup [9, 7, 8] system attempts to find sections that have gone through systematic substitution but are otherwise equal in a process they call parametrized matching. Currently, the most effective systems are based on first transforming the code into a representative form of canonical string tokens which are then used as the basis for matching [82] (*e.g.* YAP3 [143], JPlag [104] and Plaggie [2]). These types of systems are not that easily fooled by simple lexical or structural transformations commonly used to try to hide plagiarism.

An obvious shortcoming of pairwise comparison methods is that they are prone to false positives the shorter the programs are. One can argue that with a language, such as, Python that is known for its clean syntax and use of whitespace as a meaningful syntactical element, this is even more likely to become a problem. Daly and Horgan [32] implemented an alternative approach based on *fingerprinting* student assignments with digital watermarking. They added in every assignment a unique identifier coded into the whitespace, for example, after a function definition given in the boilerplate code for the exercise. This contained information about the exercise and who it was given to. Therefore, if a submission contained the fingerprint from somebody else's assignment it had been copied. The obvious drawback of this method is that the minute it becomes common knowledge it becomes useless because it is simple to just remove or not copy the whitespace. Surprisingly though, despite the authors' attempt to discourage plagiarism by informing the students of a plagiarism

detection system and possible severe penalties [32], according to the watermarks 149 out of the 283 students either copied at least one solution or allowed theirs to be copied [31].

Chapter 3

Design

3.1 System Goals and Constraints

In brief, we aim to create a tool for distributing small focused programming assignments effortlessly on introductory programming courses that are built around the Python programming language. Python is chosen because it will be the language used on future introductory programming courses at the Helsinki University of Technology. The purpose of the exercises is to gradually build students' skills and confidence in programming and to ensure that they become well-versed in elementary programming before moving on to issues of program design and architecture in subsequent projects and courses. Our hypothesis is that this will also help reduce attrition. This is a direct response to students' issues in introductory programming as described in Chapter 2. It can be argued that such short programming assignments which generally have no real-world application context do not motivate students well. However, as discussed in Sections 2.1.3 and 2.2.3, the reality of it is that some things you just cannot learn but through repeated practice, and with a large set of smaller assignments we can cover a wider range of programs that target a variety of misconceptions and program mechanics than we could with full-blown programming assignments that each take up so much time that there can only be a few. After these types of smaller assignments that provide an efficient platform for practicing basic program reading, writing and tracing skills the focus should eventually shift in the direction of program architecture and composition with larger programming exercises.

Another related matter is the choice of pedagogy on a CS1 course. There is still an ongoing debate in the CSE community on whether the first programming course should start with an object-driven approach that focuses much more on abstract

object-oriented concepts (a so-called *objects early/first* course) or with *imperative* (or *procedural*) programming. After the rush to convert programming courses to use the objects-first paradigm there has been some backlash. Sajaniemi confronts the objects-first camp – "This shift has been motivated by educators' desire to please information technology industry and potential students; it is not motivated by psychology of programming nor by computer science education research – there are practically no results that would indicate that such a shift is desirable, needed in the first place, or even effective for learning programming." [115] – and suggests a paradigm where early object-orientation would be combined with a strong initial start in procedural programming [116]. Similarly in an empirical study on the use of BlueJ, which is a popular strictly object-oriented educational tool, on an introductory programming course the authors suggest that "a minimal amount of imperative (procedural) programming initially would benefit the objects-first approach" [55].

Based on the review and discussion in Chapter 2 to effectively support the learning of programming we set the following five primary goals for our tool. We present the arguments for selecting these goals below.

1. Facilitate the development of an accurate mental model of program state and execution through consistent automatically generated visualizations.
2. Aid in tracking down the causes of programming errors and possible underlying misconceptions with reversible visual source code level debugging functionality.
3. Provide automatically assessed programming assignments to enable and support the learning of programming, in the sense of actually writing code, by practice and repetition.
4. In achieving the goals 1-3 add as little overhead as possible to the actual process of writing program code.
5. Minimize the barrier to entry and facilitate wide adoption by implementing the system as an easy-to-use web application, which also allows it to be easily updated and distributed, and to fully support distance learning.

3.1.1 Goal 1 – Visualization

As discussed in Section 2.1.1, at the center of learning to program is having an accurate understanding of what constitutes a program's state and how a program is executed.

In Section 2.2 we described how beginning programmers generally have great difficulty in forming this understanding. Goal 1 is a response to these observations and the general approach to achieving it is utilizing program visualization as discussed in Section 2.3. Also, in Section 2.2.1 it was discussed that novice programmers do not seem to know how to read and trace actual written code. Ultimately, our goal is to aid in learning to read and write code so we limit our approach to visualizing written programs as opposed to the reverse approach of *visual programming* where programs are constructed visually from different types of diagrammatic representations and which is also commonly used to teach basics of programming. In terms of the task-oriented taxonomy of software visualization [87] described in Section 2.3, the task is to support learning by facilitating program comprehension and debugging, the audience is beginning programmers, the target is the program's source code and dynamic state, the medium is a regular color monitor, and the representation depends on how and what properties we choose to visualize as described later. As discussed in Section 2.2.2 the visualizations must illustrate how such elementary constructs as loops, conditionals, arrays, recursion and parameter passing affect the control and data flow of a program. We are to provide a visual representation of the execution that both conveys the principles of the concept well and still is exact in its representation so that it is a good basis for forming a viable mental model of the subject.

3.1.2 Goal 2 – Reversible Debugging

The visualization can also aid in reaching Goal 2 when it is tied together with debugging functionality. Here, reversible execution is paramount to help with the typical "what just happened" reactions of novice programmers. After all, in the beginning students do not fully understand even the basic building blocks of programs and it can be difficult to follow the progress of a program execution, so we should attempt to reduce the cognitive load by giving them the possibility of backtracking when they get lost. Students' difficulties in debugging were discussed shortly in Section 2.2.1. While it is commonplace in algorithm visualization systems, this type of stepping back in the execution is something that conventional debuggers lack and we feel is very important for being able to properly keep track of the changes in the state of a program. Especially if a tool uses dynamic animations, you are prone to miss something and should at least be able to repeat the last step, that is, review the visualization of the last execution step. Furthermore, while it is generally far less mentally demanding and therefore tempting for students to resort to a trial-and-error strategy in order to overcome a problem, the easily accessible debugging features

coupled with the visualizations should allow students to instead more often prefer tracing the program's state as a strategy for locating errors. Related studies on students' doodles, that is, the notes and annotations written on a scratch paper, in multiple choice questions dealing with programming have found that when students use tracing as a problem solving strategy there is a high probability that they are able to answer correctly [85, 142].

3.1.3 Goal 3 – Automatic Assessment and Feedback

Goal 3 brings up the issue that learning to program requires practice but generally there are not resources for arranging adequate amounts of that. The requirement can, however, be met with automatic assessment as discussed in Section 2.4, which then eliminates the problem of resources with checking and giving feedback on massive amounts programming assignments. Furthermore, in combination with Goal 5 automatic assessment ensures the availability of the service regardless of time and place allowing students much flexibility in practicing programming with the tool. They can solve an exercise here and there slowly but surely building their programming skills. In addition, as discussed in Section 2.1.2 educational software visualization has been found to be more effective when combined with an active component. Our intent is to engage the students into using the provided visualizations in order to understand how the programs they are given/have written work. Of course, with automatically assessed programming exercises we expect them to specifically explore the visual representations of the execution to find out why their code does not work as expected.

3.1.4 Goal 4 – Ease of Use

Furthermore, being aimed at beginning programmers places some specific constraints on the system. As stated in Goal 4, neither the visualization nor assessment aspects should add any extra complexity to the process of programming, which would obviously be counter-productive. In other words, generating visualizations or carrying out automatic assessment must not require any additional annotations or use of special libraries on the part of the student. In terms of the Price taxonomy of visualization [105], described in Section 2.3, this means limiting the method to the automatic generation of visualizations, and also, we intentionally restrict the scope of programs and visualizations to small units with a few dozen lines of code. In particular, we do not aim to use this tool to deliver assignments and visualizations on component-

or class-level programming concepts and design-level challenges, or I/O and concurrent programming. Instead, the focus of representation is on basic language constructs, control statements, looping, recursion and basic algorithms and data structures. All in all, the tool must be simple enough for beginning programmers to immediately make good use of. Typical IDEs do well to ease development by integrating a specialized code editor with functionality for streamlined compilation, building and running of programs. They are, however, aimed at experienced professionals and expose an overwhelming amount of powerful features that easily confuse a beginner. If the interface is too complex, students will have to make a conscious effort to learn it, which distracts them from the actual process of designing and writing a program to solve a problem.

3.1.5 Goal 5 – Low Barrier to Entry

The tool should be lightweight in nature and easy to use which is the essence of Goal 5. In this age of ubiquitous rich Internet applications (RIA) people simply are not generally used to, or inclined to install any stand-alone software. Indeed, we should try to let students get quickly started with actual programming via writing simple programs to solve simple problems instead of having them early on get stuck on trying make sense of compilers and build tools. Learning to properly use these types of tools must of course follow at some point but when the students have first quickly got a sense of achievement in writing actual code we believe they are more inclined to put in the time to learn more.

Furthermore, this goal can be viewed from the instructors' point of view. A 2003 survey of SIGCSE[1] Technical Symposium conferences found that as much as 22% of papers published between 1984 and 2003 had presented software tools for teachers and students [138], and educational software continues to be an active area of research. However, in spite of this abundance of research few tools have gained wide adoption outside of their home institutions. Within the context of visualization tools it has been documented that one significant deterrent is that instructors simply feel that integrating a tool into their curriculum incurs too much overhead to be worthwhile [96]. Therefore, not only should the tool be easy to learn and use but effortless to deploy on a course as well.

Next we will give an overview some selected existing tools in light of the requirements we have set above.

[1]The ACM Special Interest Group on Computer Science Education

3.2 Existing Systems

3.2.1 Educational Program Visualization

Jeliot 3

Jeliot 3[2] is a Java program visualization tool. It is the fourth incarnation in a family of visualization tools (Eliot [81], Jeliot I [132], Jeliot 2000 [13] and Jeliot 3 [93]). It uses the DynamicJava[3] interpreter to extract run-time information which is then converted into a representation of the program's execution in an assembly-like language specifically developed for describing the visualizations (MCode). The visual representations are then built from this. The primary feature of Jeliot 3 is that it is able to show the progress of execution in finer steps than typical visual debuggers – it shows a smooth animation of variable assignments, object instantiation, method calls and every step in the evaluation of expressions. It also creates a call tree of the execution and there is a separate history view with static snapshots of the previous states as well. Furthermore, Jeliot 3 can be integrated into Moodle[4] and BlueJ[5]. Finally, there is proof-of-concept implementation of integrating prediction questions related to the program execution to better engage learners [94]. Figure 3.1 shows a screenshot of the tool.

jGRASP

jGRASP[6] [26, 27, 56, 57, 58, 28, 64] (Graphical Representations of Algorithms, Structures, and Processes) is an education-oriented IDE for Java, C, C++, Objective-C, Ada, and VHDL languages. The more advanced functionality is only available for Java so we will focus on that. Keeping this in mind, jGRASP is a fairly powerful tool that features many visualizations to support program comprehension: syntax highlighting, control-structure-diagrams, UML class diagrams and object viewers. It also provides an integrated debugger. In this extensive functionality is also where lies its greatest weakness. It has a lot of controls, buttons, many different views and several menus full of options which can be overwhelming and even daunting for a beginning programmer. The primary strength of jGRASP is its object viewer

[2] http://cs.joensuu.fi/jeliot
[3] http://sourceforge.net/projects/djava/
[4] Moodle is a popular learning management system (http://moodle.org/).
[5] BlueJ is a popular education-oriented development and object interaction environment for Java (http://www.bluej.org/).
[6] http://www.jgrasp.org

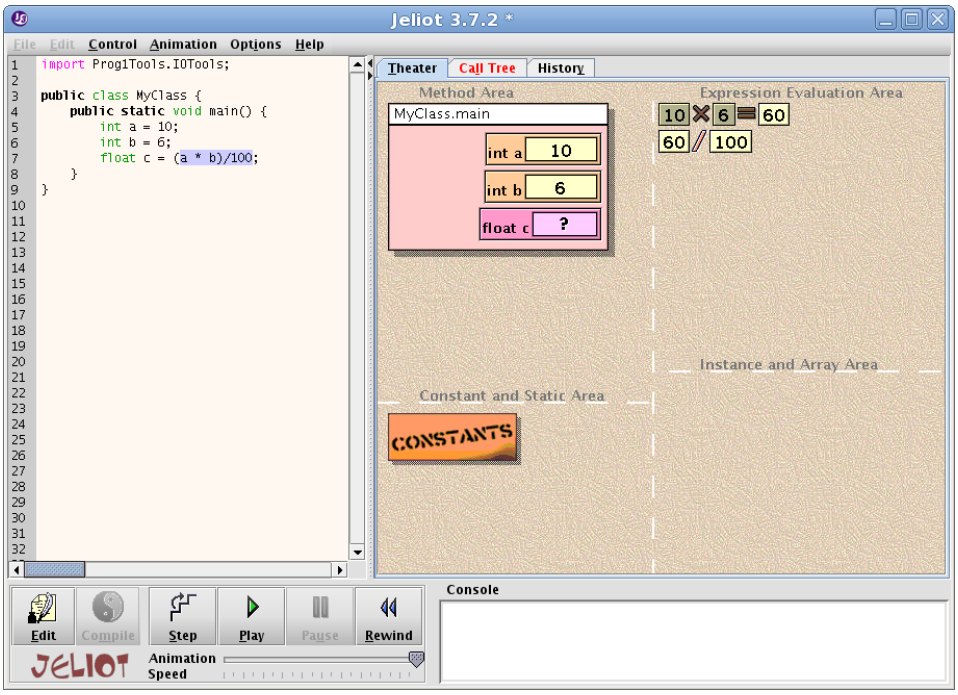

Figure 3.1: A screenshot of the Jeliot 3 program visualization tool.

system that is able to identify data structures such as stacks, queues, linked lists, binary trees, and hash tables and display them with a typical textbook-like abstract visualization. Another major feature is the so-called object workbench which lets the user create instances of classes and invoke their methods. jGRASP is written in Java and uses the Java Debugger Interface (JDI) and Java Reflection API to implement the debugging functionality and extract information for visualization.

ViLLE

ViLLE[7] [79] (the visual learning tool) is a program visualization tool whose primary features are programming language independency and the ability to create animations that include multiple choice questions at specific steps of the animation. Animations can be generated automatically from a small subset of Java source code[8]. This animation can then be viewed in a variety of languages including Python. This language translation is accomplished via a set of pre-designed string substitution rules for converting the supported subset of Java source code to other languages. At the time of writing this approach proved to be quite prone to errors and only worked

[7] http://ville.cs.utu.fi

[8] For information about the restrictions see http://ville.cs.utu.fi/media/ViLLE_supported_java_features.pdf.

in very simple cases. For example, in the Python animation empty or wrong lines were highlighted as there was no equivalent constructs to the Java statements in the Python version. New languages can also be added by writing a set of parametrized string replacement rules for the new language. Figure 2.8 on page 22 shows a screenshot of ViLLE.

JIVE

JIVE[9] [29, 48, 49, 47, 46, 50] (Java interactive software visualization environment) is a Java program visualization tool integrated within the Eclipse IDE[10]. The main features are its additional visualizations to the Eclipse debugger and reverse debugging functionality. The run-time state is visualized with an object diagram that shows the object structure and method activations. There is also a sequence diagram that depicts the execution history. The reverse stepping of programs is implemented by recording the visualization events that were created based on the information collected via the Java Platform Debugging Architecture. The animation of the execution history can then be navigated backwards as if executing in reverse. When the user inspects past states, the execution of the client program is suspended and it is resumed when the user steps past the stored history.

LEONARDO

LEONARDO[11] [25] is a C program visualization environment that is especially interesting due to its implementation of reversible execution. The code is compiled and then executed on a virtual CPU that implements the reversible execution functionality and structures can be visualized by specifying a mapping from structures to visual representations using a special declarative language. However, the tool is only available for the MacOS on PowerPC processors.

DynaLAB

DynaLab[12] [20] is a Pascal program visualization tool that is interesting because of its reversible execution functionality. The Pascal code is compiled to a representation

[9] http://www.cse.buffalo.edu/jive/
[10] http://www.eclipse.org/
[11] http://www.dis.uniroma1.it/ demetres/Leonardo/
[12] http://www.cs.montana.edu/ dynalab/

that can be run within their virtual machine implementation called the Education Machine which implements the reversible execution functionality.

VIP

VIP[13] [139] (Visual InterPreter) is program visualization tool for C++ whose concept is very similar to this work. It is implemented in Java and is therefore portable and can be run over the web as an applet. The code is executed and run-time information is collected from an embedded interpreter[14] which can process basic C++ constructs but not, for example, classes. VIP can provide two types of content: example programs and small exercises that are checked for correctness with included tests. It is able to textually visualize the evaluation of expressions step-by-step showing the values of operands for each operation. Figure 3.2 shows a screenshot of VIP.

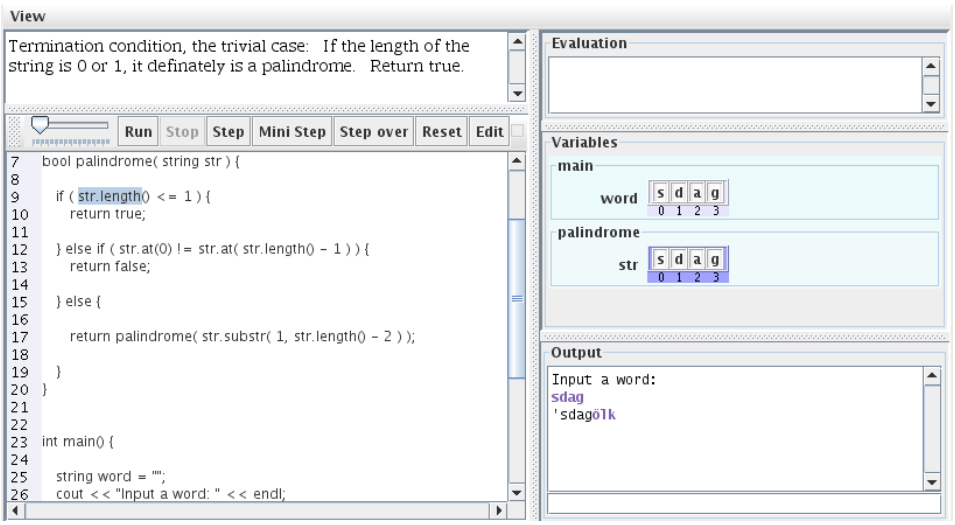

Figure 3.2: A screenshot of VIP program visualization tool.

3.2.2 Education-Oriented Programming Environments

In addition to program visualization tools there also exist other types of programming environments that are directed to beginning programmers. BlueJ [73] is a popular education-oriented IDE for Java. Its primary features are visualization of the class structure as a static UML diagram and the ability to create objects and call methods on them via a simple GUI in order to examine their behavior. It is designed strictly

[13] http://www.cs.tut.fi/ vip/en/
[14] CLIP, http://www.cs.tut.fi/ vip/clip/clip_english.html

for the objects-first approach to teaching. Ox [126] is another tool that provides a similar type of object workbench. Another type of tools are tiered language tools in which novices can use more sophisticated versions of a language as they learn more (*e.g.* ProfessorJ[15] [51]). Finally, there exist a variety of tools that provide a microworld with which learners can familiarize themselves with programming concepts by manipulating the objects in the world with programs written using a limited set of simple commands such as "go one step forward" or "turn right" (*e.g.* Jeroo[16] [117], JKarelRobot [22], Greenfoot[17] [72], PigWorld [84]).

3.2.3 Automatic Assessment of Programming Assignments

As discussed in Section 2.4 there currently exist many systems for the automatic assessment of programming exercises. Most of these follow a very similar workflow. First the student acquires the instructions and code for the exercise via the system. Then the student writes a solution in a separate environment of his choosing. Finally, the student possibly packages his solution appropriately or directly uploads it to the system which then runs analyses on the student's submission and gives feedback. Ala-Mutka [4] gives a general overview of automatic assessment systems. Most relevant to the scope of this work, however, are systems that provide a leaner workflow in order to allow the easy distribution of a larger set of smaller exercises that students can solve with little overhead. We will next review systems with this focus.

JavaBat

JavaBat[18] [102] is a web site that provides a large set of small Java programming problems. Solutions are written in a text box directly in the browser and can be submitted to get immediate feedback. Assessment is done using unit tests. Figure 3.3 shows a screenshot of the site.

WebTasks

WebTask [110] is a web-based platform for small Java programming exercises. Feedback is given based on unit tests and a static analysis tool Checkstyle[19]. Authors

[15] http://www.professorj.org/
[16] http://home.cc.gatech.edu/dorn/jeroo
[17] http://www.greenfoot.org/
[18] http://www.javabat.com/
[19] http://checkstyle.sourceforge.net/

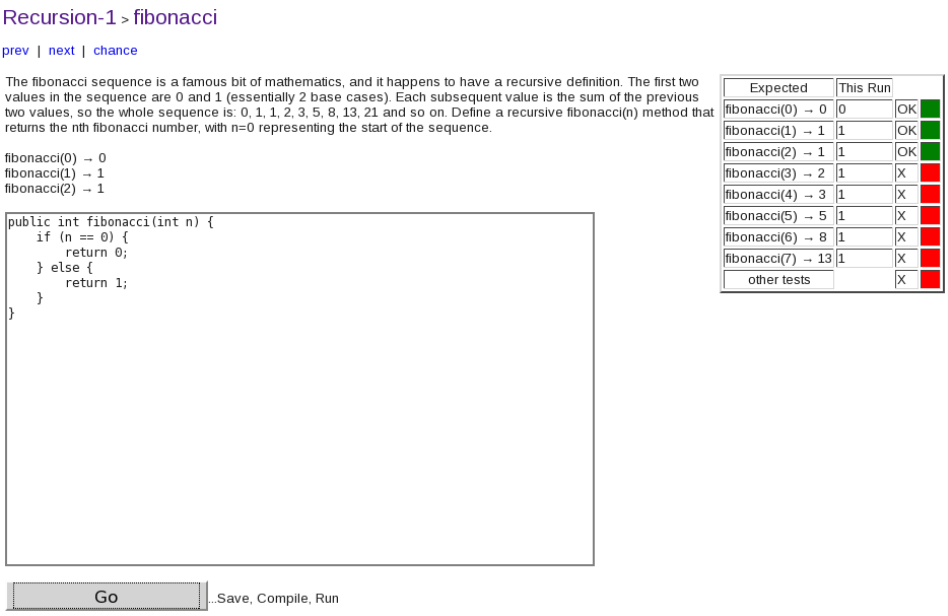

Figure 3.3: A screenshot of the JavaBat site. On the right is the feedback for this submission.

state that the purpose of the system is not to track students' progress for grading purposes but to "provide a rich and risk-free training platform for learning how to program" and motivate the system by referring to common complaints that "there are far too few "easy" programming tasks to get some hands-on experience in programming". Exercises can be multiple choice questions, 'type in the missing method body' -type tasks solved directly in the browser or writing and uploading a single class. Furthermore, accepted solutions are published for users who have passed the same task and they can comment on them.

Javala

Javala[20] [83] is a web environment for learning Java programming. It consists of tutorials and embedded exercises. The 'fill in method body' -type assignments are solved directly in the browser by writing into a text box. Assessment is accomplished with accompanied tests that are run on the server when an exercise is submitted. A unique feature in Javala is its game-like scoring of users. For each correct solution the user receives Javala points and there is a global high score table of Javala users. Furthermore, users have a rank in the system (*e.g.* Java Tourist, Java King) based on

[20] http://javala.cs.tut.fi

their score and every time a user is promoted this is shown in a column on the Javala web page. The authors report that these addictive elements cause "some students to spend nearly five hour long continuous periods in Javala with only 10 minute breaks in between". Figure 3.4 shows a screenshot of an exercise in Javala.

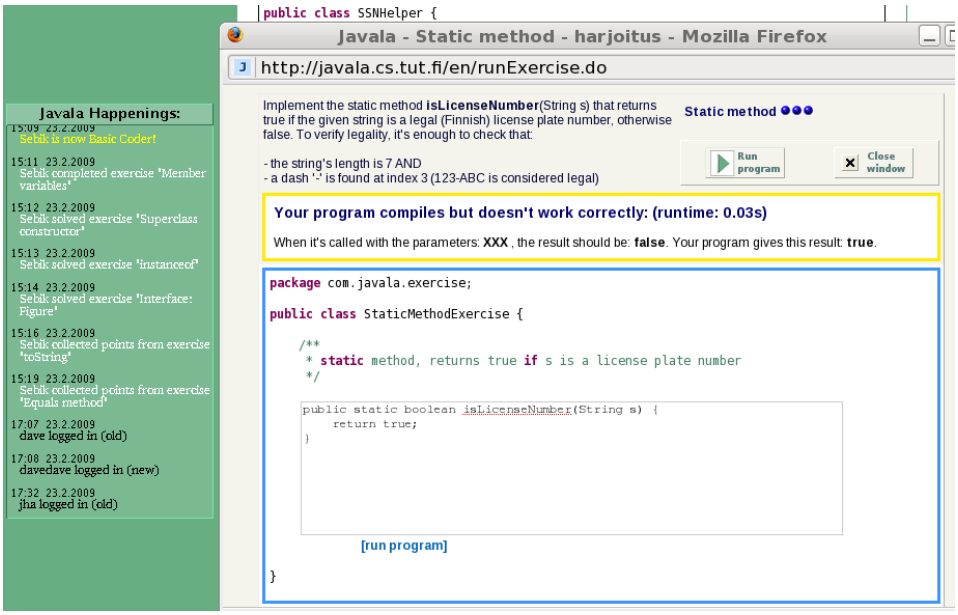

Figure 3.4: A screenshot of a Javala exercise.

ELP

ELP [136, 137] (Environment for Learning to Program) is a web environment for Java, C# and C example programs and exercises. The system supports fill in the gap exercises that can be solved and assessed directly online in the environment. The student fills in the gaps to complete the program and gets immediate feedback after submitting his solution. Feedback consists of the results of a static analysis that computes software metrics and a estimation of structural similarity to a model answer and a dynamic analysis that involves executing the student's solution with test data and comparing the expected to the observed behavior. Authors point out that ELP allows students "to produce working programs at the early stages of their course without the need to familiarize themselves with a complex program development environment". The client is implemented as a Java web Start application. Figure 3.5 shows a screenshot of an exercise in ELP.

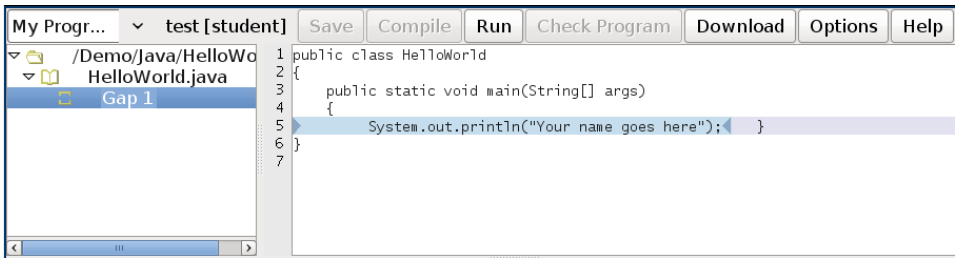

Figure 3.5: A screenshot of an ELP exercise.

CodeSaw

CodeSaw[21] is a commercial environment for running example programs and doing small fill in the gap programming exercises directly on the web. It supports many languages including Java, C++ and Python. Solutions are evaluated for correctness by running them with test data and comparing the results to the ecpected values. The programs and tests are run on the server and the CodeSaw client that manages the execution is a Java Web Start application. Figure 3.6 shows a screenshot of an exercise in codesaw.

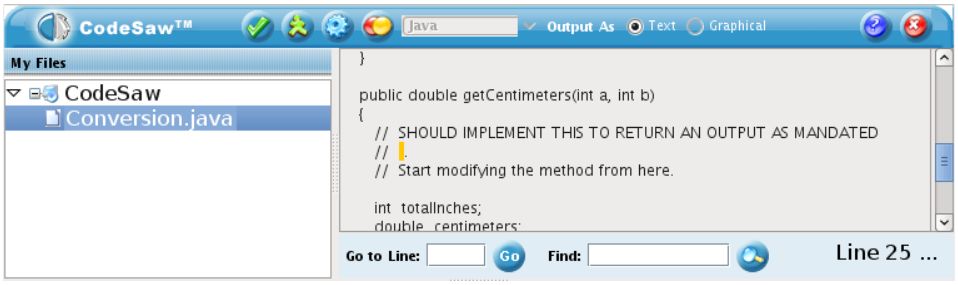

Figure 3.6: A screenshot of an exercise in CodeSaw.

WebToTeach and CodeLab

WebToTeach [6] was a web system for various types of exercises with instant feedback including writing a code fragment, writing data for a test suite or writing a complete source of a program directly in the browser by editing the answers in HTML text boxes. Later it grew into the commercial web system CodeLab[22] which provides similar features for a range of programming languages.

[21] http://www.codesaw.com
[22] http://www.turingscraft.com/

3.2.4 Tools for Python

There are several professional IDEs for Python that provide some visual debugging functionality (*e.g.* ERIC[23], PyDev[24], SPE[25], Wingware[26], Komodo[27]) such as controlled line-by-line execution and a table view of the values of variables. However, there are not many educational tools for Python. There are a few microworld programming environments (rur-ple[28], Guido van Robot[29], Turtlet[30]), a programming environment for exploring artificial intelligence and robotics (Pyro [17]) and media programming (JES, Jython Environment for Students [54]). Finally, we found a dead and unfinished project for building a visualization tool for Python programs (OpenExVis[31]).

3.2.5 Summary

First, all the described program visualization tools and non-commercial automatic assessment systems lack Python support, except for ViLLE which provides rudimentary support for visualizing Python execution by translating the Java code animation into Python. Python is an absolute requirement for us and therefore we would at least have to implement this support if we were to base our work on any of the existing systems. Second, none of the described automatic assessment tools provide an integrated visual debugging environment and none of the program visualization tools provide automatically assessed programming exercises, except for VIP which is designed for C++. Third, of the program visualization tools only Jeliot 3, ViLLE and VIP can be deployed over the web in a portable manner. To sum up, we might be able reach our goals by implementing Python support in Jeliot 3 or VIP. However, in addition to being aimed at Java and C++, neither of them support reverse stepping which is also one of our goals. In conclusion, as will be described in Chapter 4, instead of starting development on Jeliot 3 or VIP we chose the visualization framework developed in the research group where this thesis project was carried out because this had the advantage of enabling easy and direct contact with the developers of

[23] http://eric-ide.python-projects.org/
[24] http://pydev.sourceforge.net/
[25] http://pythonide.blogspot.com/
[26] http://www.wingware.com/
[27] http://www.activestate.com/Products/Komodo/
[28] http://sourceforge.net/projects/rur-ple/
[29] http://gvr.sourceforge.net/
[30] http://www2.lut.fi/ jukasuri/Kilppari/
[31] http://openexvis.sourceforge.net/

that library. Moreover, the author already had previous experience in developing visualizations with the library.

3.3 Design Choices

The goals described in Section 3.1 set the objective of improving the learning of programming on entry-level programming courses by aiding in the development of program comprehension, tracing and writing skills via the use of program visualization and automatic assessment. The purpose of visualization is to facilitate the forming of a viable mental model of program execution and state, and automatic assessment will enable us to provide enough practice and feedback to incrementally develop the desired skills and knowledge. According to our survey there is only one tool (VIP) that fits this description, however, this is built for C++ and we require Python language support. Moreover, it does not feature reverse stepping.

Next we will describe the different aspects of our tool. In terms of the Price taxonomy of software visualization [105] we have already defined the scope. Now we will define the content, interaction and the form to some extent. We will also describe the nature of the automatic assessment. Method will be discussed in Chapter 4 and effectiveness is discussed in Chapter 5.

3.3.1 Representation – Content and Form

The fundamental idea behind the visualization is that, when students are writing and running code to solve the programming assignments, we engage them into examining and inspecting different aspects of the program's behaviour through automatically generated graphical representations. As stated in Goal 1 the visualizations will always provide an accurate model of state and execution and therefore aid the student in forming viable mental models of programming constructs, and also help them to trace control flow and track the values of variables. To this end, control flow and the code must be depicted in such a way that we can immediately see the control jumps caused by branching and looping constructs. This can be achieved by appropriately highlighting the line being executed and the line where control was before that, that is, we show from where and to where the control transitioned. Conventional debuggers typically do not show the previous line of execution in this manner but only highlight the point where the execution has currently been suspended, however, as discussed in Section 2.3.3, this is common in educational program and algorithm visualization

(*e.g.* [76, 79]). In addition, to facilitate understanding scoping and recursion, the execution stack and the call sequence, must be visualized. The visualization should aim to give a basis for a viable mental model of execution scope, function invocations and recursion.

Furthermore, we need to address two differing needs of the data visualization: on the one hand, in short and simple programs the visualizations must be able to exactly model and illustrate the hidden interactions and dependencies in the data flow, and on the other hand, in more complex programs we should hide some complexity to ease conceptual understanding of algorithmic procedures. This is achieved by using simultaneous visualizations on two different levels of abstraction. We show both the states of variables in a table-like view where we also highlight the introduction of new variables and the changes in their values, and also a more conceptual view of the data in the form of canonical visualizations for basic data structures; arrays, lists, trees and graphs. This way the table view allows students to observe the exact manipulations and dependencies in the data and the data structure visualizations bring forth the mechanics of an algorithm. Finally, while we will in this project develop basic visualizations to address the described needs, the tool must be easily extendible in terms of the visualizations so that down the road we can try out different types of visualizations or provide more advanced or improved optional additional visualizations.

3.3.2 Interaction

Especially with beginning programmers the majority of time is spent on debugging erroneous programs. This can be very time-consuming and quickly lead to frustration. Therefore, it makes sense to streamline this process for novice programmers. This requirement is expressed in Goal 2 which calls for a reversible source code level visual debugger. This means that the tool must have a source code editor that supports running the program in steps line-by-line. This way, the visualizations described in the previous section allow students to observe a line-by-line animation of the progress of a program's execution. Indeed, the process of debugging inherently involves the application of a mental model of program execution to make predictions on variable values while observing the actual behaviour to discover possible discrepancies, and our visualizations are intended support this activity.

For temporal navigation there must be VCR-like controls: go to the beginning, step back, step forward and run to the end. The intent is that these controls enable and

engage students into exploring program behaviour in order to find programming errors and understand control and data flow better. They can instantly create animations of any program, which allows them to ask and answer questions about the exact execution and evaluation of different programming constructs. While professional IDEs often do have very powerful debuggers, the problem is that they tend to be difficult to operate and require better understanding of the debugging process and program execution than novices have, since these debuggers require breakpoints to be added to the program and do not allow reverse execution. After all, knowing where to insert a breakpoint in itself requires the programmer to have good enough understanding of control flow to be able to make a good educated guess of where a problem lies. By providing an easy interface for tracing code we expect that students will use that as a strategy for finding errors and understanding program behaviour.

In a related study Saraiya et al. [119] conducted experiments to determine which features are effective in an algorithm visualization. The applicability of the results to software visualization in general is debatable since, for one thing, the tests only included a single algorithm (heapsort), but they do warrant some discussion nonetheless. Saraiya et al. found that the most important features were having a direct and absolute control over the pace of the visualization and having a ready-made good example data set instead of having students select the input themselves. Interestingly though, the ability to back up the animation did not show significant improvement in learning. However, in the two experiments they carried out, in the first one the back button was only available in versions of the visualizations that had no given example data and in the second experiment the functionality was available only in versions that included example data. Therefore, no comparison was made between a version that had example data and a back button and another version without the functionality, which would have given more indication on the effectiveness of this specific feature as it appears that having good predesigned example data is a dominating feature in this case.

3.3.3 Automatic Assessment and Feedback

As stated in Goal 3, automatic assessment is used to enable delivering a large set of short assignments while still allowing students to get immediate feedback on the correctness of their solutions. In this project the assessment will be done based on the common unit test based method. This is a straightforward approach for instructors to implement new exercises since it does not require learning a separate description

language for describing the assessment criteria and they can make use of the wealth of existing exercises. It provides a low barrier to entry and is well-suited for the smaller programming exercises, such as, writing a single function that takes some input and processes it in some way to return some output.

Ideally, a student's workflow would be something of the following: 1) the student reads the assignment and starts experimenting with the programming concepts and constructs involved in the exercise possibly making use of existing code given as part of the assignment – this exploration is supported by the possibility of interactively stepping through the execution and the animation of program state, 2) next the student attempts to solve the exercise which he then tests by running it with test data while comparing its behaviour to the expected outcome, 3) when the student is satisfied that his solution meets the requirements he submits it for evaluation, 4) if the program is incorrect, the student goes on to debug the program based on the feedback given in the failed tests while making use of the reversible debugging functionality and the visualizations in this process until when finished he repeats step 3.

Chapter 4

Implementation

In this section we give an overview of the functionality of the tool. First we introduce the interface and visualizations from an external view and then we explain some of the technical details.

4.1 Functionality

Figure 4.1 shows the basic view of our tool known as *Jype* from now on.

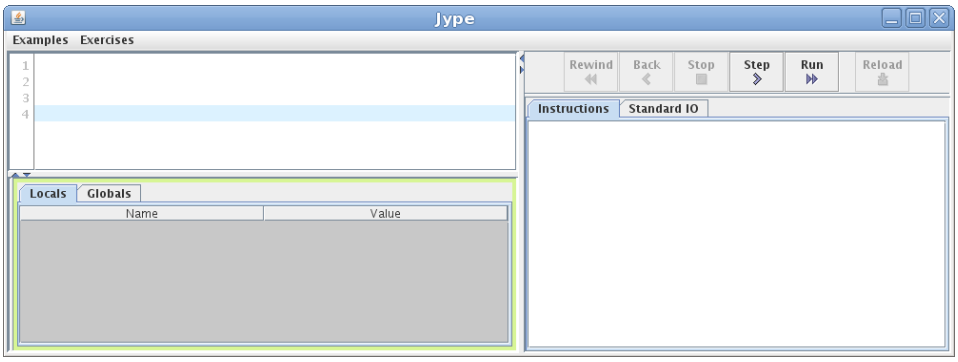

Figure 4.1: The basic view of Jype.

In the left upper corner is a text component that is used for editing and visualizing code. The line numbers are shown on the left and the current line of the caret is highlighted when editing. In the top right corner are the controls for Jype. In the left lower corner is the primary visualization area and next to it on the right are two tabs for text views. In this use mode, practice mode, topmost on the left are also menus for switching and loading content – examples and exercises.

4.1.1 Visualization

Figure 4.2 shows the basic control flow visualization in Jype when running a program step-by-step.

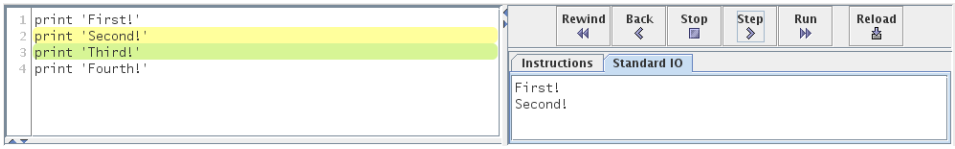

Figure 4.2: The basic visualization of control flow.

The currently executing line is highlighted in green and the previously executed line is shown in yellow. When an uncaught exception interrupts execution, the cause, that is, the line from which the exception originated is highlighted in red. On the right you can also see that the text written to the standard output stream is displayed in the 'Standard IO' tab.

Figures 4.3 and 4.4 show the basic visualization for data flow in Jype. The names and values of variables are shown in a table view with character representations.

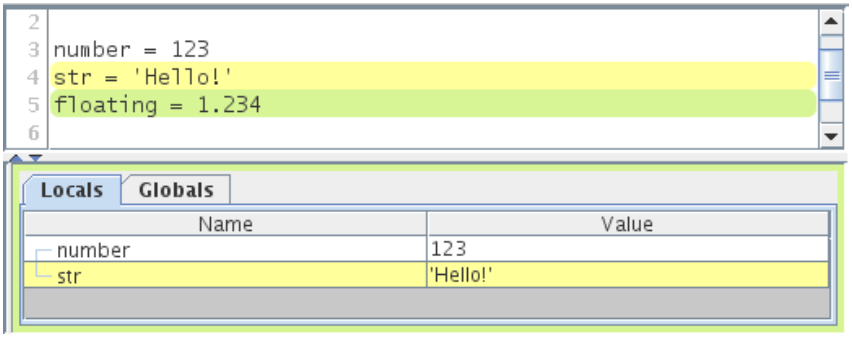

Figure 4.3: Newly introduced variables are highlighted.

When a new variable is introduced, that is, a name is bound to a value for the first time in that scope, the row of the table is highlighted in yellow matching the color in the code view. When an existing name is bound to a different value only the column with the value is colored. Tooltips show additional information about the values as seen in Figure 4.5. This information that shows the object nature of the values might be confusing to a student not yet familiar with objects so it is only available by request in the form of a tooltip. Figure 4.6 shows various types of data in the table view of variables.

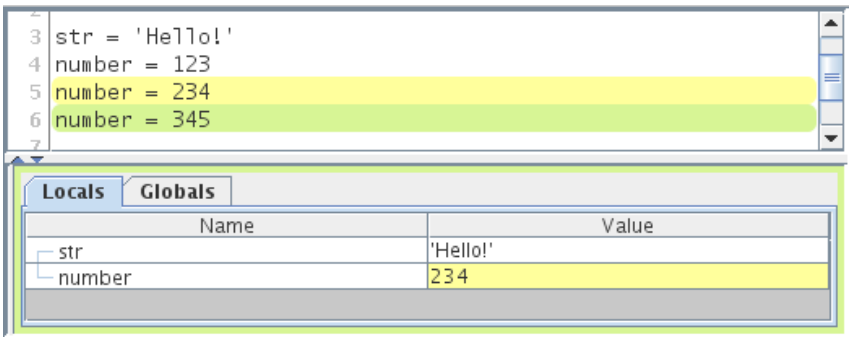

Figure 4.4: The values are highlighted when they change.

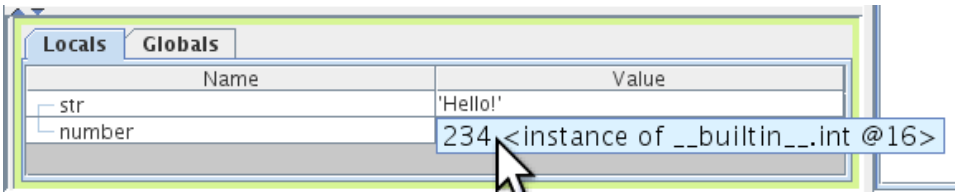

Figure 4.5: Tooltips show additional information about values.

The two tab panes give the names and values of variables in the local and global scopes. We chose to show the scopes this way instead of continuously showing both to preserve screen space which is often scarce when it comes to visualization software, and because one will rarely inspect the global scope in Python. Typically using global variables is discouraged because of the intricate coupling it introduces. In fact, by default you can only assign a value to a global name on the top module level where the local scope is equivalent to the global scope, unless a name is specifically declared global with the special 'global' keyword.

Finally, Figure 4.7 shows a class object and an instance of that class in the table view. For non-built-in classes and objects the attributes are displayed in a foldable tree view.

Next to the variable view is another visualization for the control and data flow of a program. Figure 4.8 shows the visualization of the execution stack of the program. The stack remains hidden when it consists of only the top module unless purposely dragged into view. In other words, when the student has not yet learned the concept of function invocations or execution scopes and the examples and exercises are written on the top module level, the stack will not be displayed and this way we avoid confusing the student with this additional aspect. For each new execution scope – which in Python can be a module, a class or a function – a box showing the

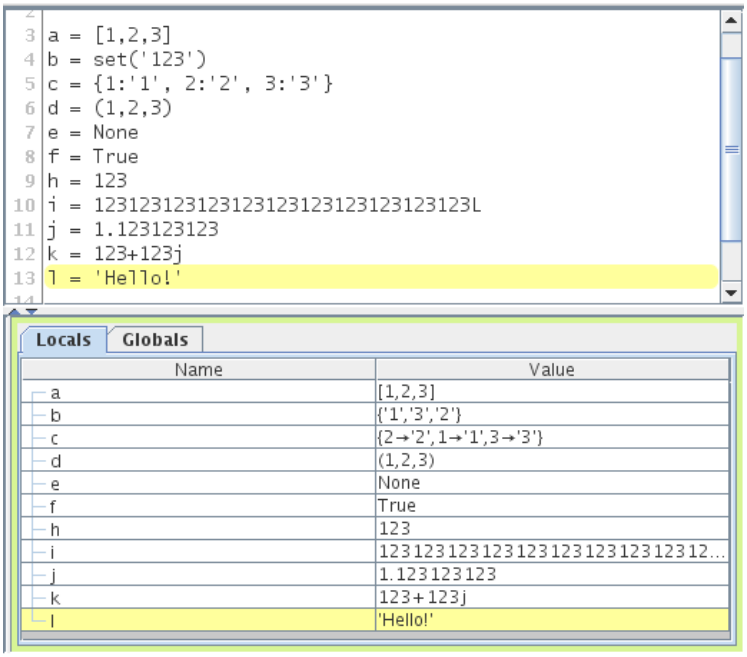

Figure 4.6: Representations for various types of data in the table view of variables.

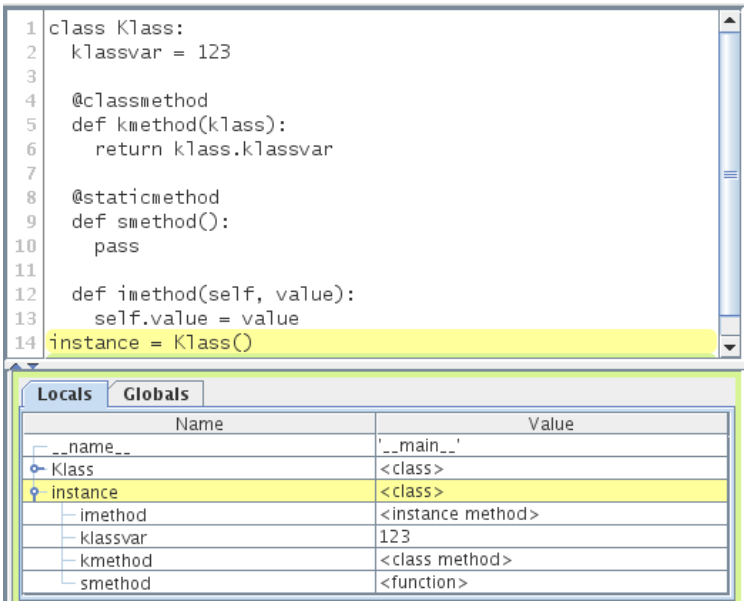

Figure 4.7: Class and object values in the table view of variables.

type and name of the frame is added to the stack. The colors of the boxes match those used in the code view. The call sequence can also be quickly observed via highlights in the code view that are invoked by hovering over the boxes as shown

in Figure 4.9. For functions we also see the list of arguments and explicit function returns are visualized as well as shown in Figure 4.10.

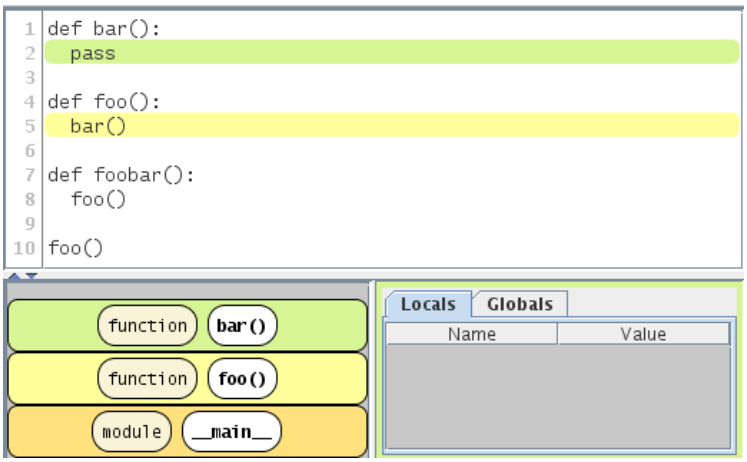

Figure 4.8: The execution stack visualization in Jype.

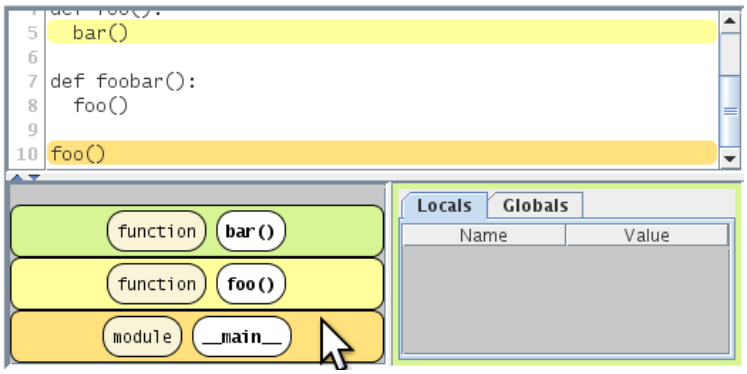

Figure 4.9: The respective lines are highlighted in the code view when the mouse pointer is hovered over the boxes representing stack frames.

In addition to the described basic visualizations Jype can be extended to provide alternative views to the data and the control flow. Currently, Jype uses a separate data structure visualization library to display a dynamic array visualization for Python's built-in list data type as shown in Figure 4.11. Also, because the Python implementation used in Jype is able to fully interact with Java classes and objects and the same visualization library also includes full java implementations of data structures with visualizations, these can also be used in Jype. Figure 4.12 shows an example with a binary search tree. Other examples of data structures implemented in the visualization library include a b-tree, a red-black-tree, graphs, lists and queues.

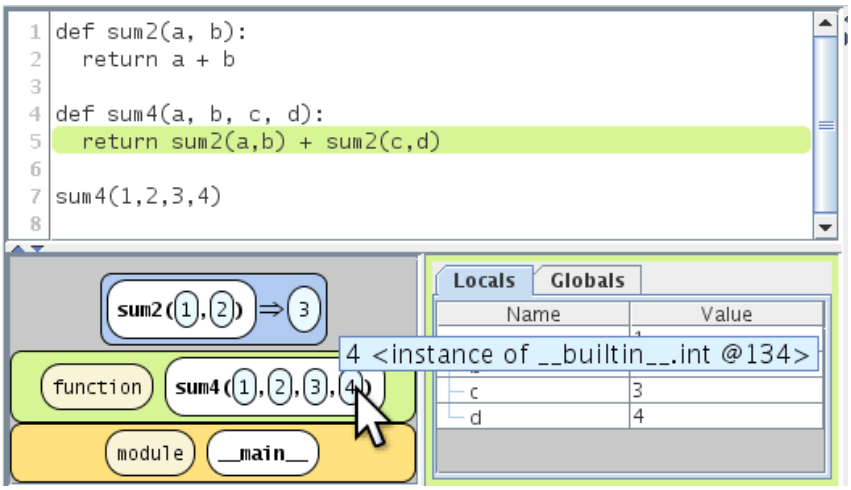

Figure 4.10: The blue box signifies the return from a function.

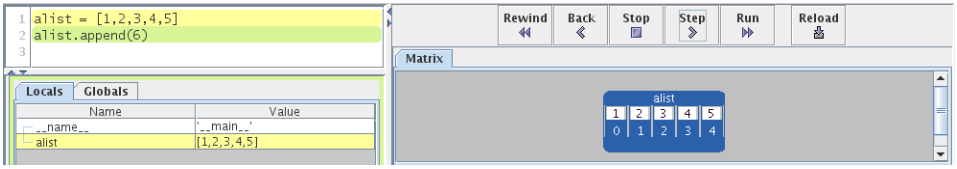

Figure 4.11: The additional visualization for a Python list in Jype.

4.1.2 Interaction

The execution of programs in Jype is controlled with basic VCR-like buttons as seen in the upper right corner in Figure 4.1: go to the beginning, back up one step, stop execution, execute one step and run the program. The user can fluently switch between editing and running code without any compilation steps – when the user modifies the code while its being executed/animated, the execution and animation are automatically stopped and cleared.

Another form of navigation supported by Jype is clicking on the boxes in the execution stack visualization. This moves the execution/animation to that state in the execution history, that is, rewinds the program animation to the step in that frame where the control was before entering the subsequent frames. This could be called 'stepping back to the caller'.

Something to note about the step-by-step execution is that even the standard input/output view is tied to the execution state, that is, like everything else printing to the streams is reversed when stepping back. Also, when the input stream is read, for example, with the built-in input() function this view provides a line editor

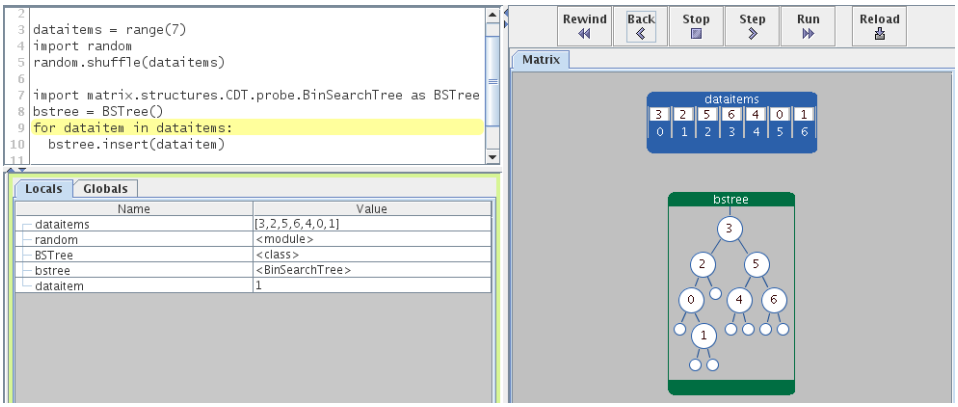

Figure 4.12: Using a binary search tree implementation from the included data structure visualization library in Jype.

where character data can be input to the program. However, when reversed and then stepped forward the input is not queried again. Instead the input received once already is used. This enables stepping back in the animation without having to write the input over and over again. If the execution/animation is stopped instead of reversing or rewinding then the input is again collected via the console view.

4.1.3 Content

Jype can host two types of content – examples and exercises. Examples are meant to provide a pedagogically effective example program for students to examine via the tracing and visualization capabilities of the tool. Exercises are intended to let students quickly and easily get into writing small programs with the help of instant feedback. In addition to code both content types may include some instructions in the tab next to the standard input/output console (see, for example, Figure 4.1). For each exercise submission a new tab pane is created in the lower right corner. It displays the feedback from the assessment as shown in Figure 4.13. For each test run on the student's code we see a comment describing what was tested, the result of the test and an additional comment describing the reason if the test failed.

4.2 Technical Details

Jype is implemented in Java using the standard Swing GUI library and is thus largely platform-independent. Furthermore, it can be run over the web as a Java Applet or a Java Web Start application, and as a stand-alone application as well.

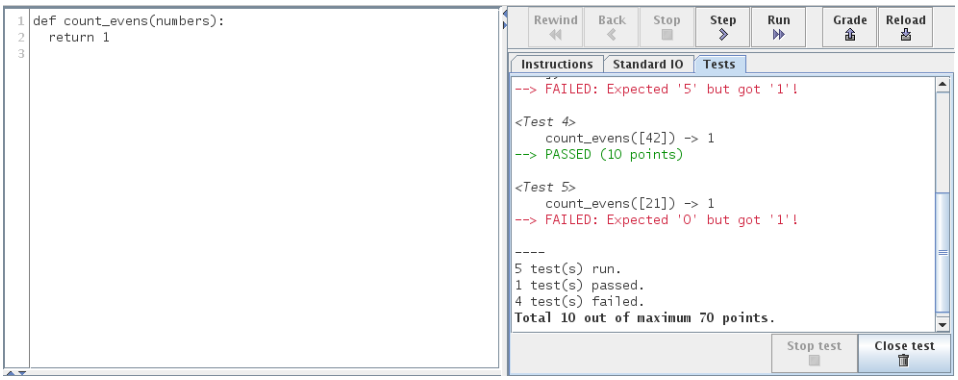

Figure 4.13: The feedback from the assessment is shown in a new tab pane.

Therefore, it provides a variety of possible options for deployment. Furthermore, we have integrated Jype into a course management system (TRAKLA2 [98]) for managing the submissions of exercises. The system allows us to create a course with Jype examples and exercises and individual accounts for each student whose progress – submissions and points – is then recorded and can be tracked by the instructor. Finally, the tool follows a modular design and is intended to be extendible especially in terms of the visualizations. Each of the views implement a common interface and it should be simple to develop additional or alternative visualizations .

4.2.1 Python Tracing

The execution of programs in Jype is accomplished via an embedded Java implementation of Python called Jython[1] that currently corresponds to CPython[2] version 2.5. The tracing and pacing of the execution for the debugging and visualization functionality is carried out via a tracing callback function[3]. This is the standard interface for source-level Python debuggers. The trace function is called with information about the current execution state every time an execution frame is entered or exited, an exception is raised or the interpreter is about to execute a new line of code.

An advantage of the approach of using the debugger interface is that hooking deeper into the internals of Jython to trace execution in more detail, *i.e.* customizing the Jython compiler and interpreter, would make keeping the system in sync with the development of Jython much harder. This could be considered an important

[1] http://www.jython.org

[2] The primary implementation of the Python programming language in C is generally referred to as CPython.

[3] http://docs.python.org/library/sys.html#sys.settrace

argument because students are meant to experiment with writing different types of programs and we should try not to restrict their expressiveness with regard to standard language constructs. This problem would obviously go away if an API for low-level tracing were added to Jython or there were sufficiently extensive developer resources for maintaining the tool. Furthermore, Jython is also a fast moving target since its currently quickly catching up with Python 2.7/3.0 and there has also been recent work to implement an actual Python bytecode virtual machine within Jython instead of translating Python programs into Java bytecode representations of Python programs as it now works. Tracing the execution via a customized version of the VM might be the way to go, in future when it is ready, in order to get finer tracing. Moreover, initially work was started with the latest release version of Jython. However, we had serious problems with this version, it was based on a fairly old version of Python (2.2) and most importantly was not developed anymore because the project had reworked the entire compiler infrastructure since then. Therefore, we moved on to using the development branch of Jython. At the end of this project the next release of Jython was still in the beta testing phase. Finally, initially we implemented loading Python modules via the Java classloader mechanism to enable importing modules in the applet and web start environments, however, this functionality was later also surprisingly added in Jython's latest development versions.

In addition to Jython, Jype makes use of the Matrix library [75] which is a data structure visualization and simulation framework. It is based on a data-driven approach where the data structures are implemented using special objects that correspond to basic primitives, such as, an integer or a boolean value, but whose state transitions are automatically stored in an associated *animator* object. The animator then provides an interface for stepping back and forward in the stored state sequence – undoing and redoing the stored operations – thus allowing us to produce an animation by repainting the visualizations when the sequence is navigated via the animator.

The Python code in Jype is executed in a separate thread that is suspended in the callback function when the execution is paused. After each execution step the list of registered observers are called with the current Python execution state information. These observers – the various visualizer components – store the state information for the visualizations in the Matrix animator managed objects. When stepping back, the debugger undoes operations via the animator and finally requests a repaint of the visualizers. They then go on to load their state information from the animator-managed objects, thus giving an appearance that we have reversed the execution

while actually we are simply traversing the execution history stored as an animation. When the execution is stepped forward, states are loaded from the animator until we have reached the current state of the Python execution which is then continued to gather information for the new step. The debugger fully transparently switches between pacing the actual execution and controlling states via the animator. This is a similar approach to that of JIVE that also stores a history of the execution to provide an illusion of reverse stepping [47]. A weakness of this approach is that anything that should be viewable at any time must be stored in the animator. This means that we cannot access and store visualization data in a lazy fashion, for example, by retrieving the attributes of objects only when their tree view is expanded, because that information is not generally available since we might or might not be in that specific state of the actual Python execution.

4.2.2 Defining Exercises and Examples

The automatic assessment and feedback of programming exercises in Jype is based on including unit tests made with Python's standard 'unittest' module with the assignment. Defining an exercise for Jype is extremely simple. Creating an exercise starts with a main file 'exercisename.py'. This file contains the boilerplate code that is initially given to the student. Also, the 'docstring' of this Python source code file (module) is shown as the instructions for the exercise. The code after the '#BEGIN' instruction is the code given to the student. If the exercise needs something to be set up, for example, defining some classes or importing supporting exercise modules – this can be done by putting the needed code before the '#BEGIN' instruction. That code is not shown to the student and is always executed before student's own code when running the program. Examples can be defined in the same way.

The unit tests for an exercise are defined in a separate file whose name prefixed with 'test_'. The only requirement is that this module defines a 'suite' that contains the tests. The docstrings of the test functions are shown to the student as test descriptions if they are defined and the optional messages of assertions are shown if a test fails. Within Jype the student's code is available as if it was defined in the file that describes the exercise. These imports are intercepted and the tests are run on the code the student has actually written. A function decorator named 'score' can be used to place more value on specific tests (default score is 10).

Figures 4.14, 4.15 and 4.16 show an extremely simple example of an exercise. Appendix A gives another example of an exercise and its tests. An obvious improvement

for the tests would be to randomize them – the tests can of course make use of any available standard Python libraries such as 'random'. Figure 4.13 shows an example of the feedback produced by the tests in Appendix A.

```
'''Complete the function toggle so that it returns
the boolean value false if it is called with true
and vice versa.'''

#BEGIN
def toggle(bool):
  pass
```

Figure 4.14: The definition of the instructions and initial code for an exercise 'toggle'.

```
import unittest
from test_support import score
import toggle as student

class ToggleTestCase(unittest.TestCase):

  def test_true(self):
    '''Calling toggle with value True:'''
    self.assertEqual(False, student.toggle(True),
                     'Function toggle must return False when called with True!')

  @score(20)
  def test_false(self):
    '''Calling toggle with value False:'''
    self.assertEqual(True, student.toggle(False),
                     'Function toggle must return True when called with False!')

suite = unittest.TestLoader().loadTestsFromTestCase(ToggleTestCase)
```

Figure 4.15: The tests for the exercise 'toggle'.

```
<Test 1>
    Calling toggle with value False:
--> PASSED (20 points)

<Test 2>
    Calling toggle with value True:
--> FAILED: Function toggle must return False when called with True!

----
2 test(s) run.
1 test(s) passed.
1 test(s) failed.
Total 20 out of maximum 30 points.
```

Figure 4.16: The feedback from a single test run using the tests in Figure 4.15.

The overall solution resembles the way how exercises are defined in Javala [83], that is, instead of, for example, some new XML description format, the exercises are defined in a simple, low-overhead way directly with Python. The idea is that an

exercise and its tests can be designed, run and tested even independently from the tool.

Chapter 5

Evaluation

In this chapter we discuss the implemented tool, Jype, with regard to our goals and related work.

In the time frame of a master's thesis project it was not possible to design and implement a new tool and still carry out a proper student evaluation of its effectiveness. Instead, as an initial pointer to future integration to an introductory programming course and possible further development, we conducted a qualitative expert evaluation supported by input from 7 researchers in the field of programming education tools. The evaluation was divided into six sections that each have a specific area of focus. The evaluation form contained a set of additional questions for each section to guide the focus of the evaluation. The form is available in Appendix B. The reviewers were also supplied with an extended abstract and a video tutorial of the tool. They could, of course, try out the tool in practice as well. Next, drawing on the feedback and comments received from the experts, we discuss the possible advantages and drawbacks of Jype.

5.1 Visualization in Jype

Most reviewers agreed that the visualizations provided by Jype are clear and intuitive, and that the overall layout of the application is well organized. One reviewer did note, however, that the tooltips in the variable view go beyond novice understanding since they bring out the object nature of the values. This might be true but, on the other hand, it was a conscious decision to only make this information available in the tooltips and the student can do well without looking at them until they are

equipped to process this information. In addition, a few reviewers brought up the obvious shortcoming of the current visualizations which is that the data *flow* is not actually visualized in the manner as in some state of the art algorithm and program visualization tools. In other words, you cannot actually see a smooth animation of the transition when a value is bound to a name. Instead, Jype only shows snapshots of a program's execution – the state of the program at a given step – and the transitions between states are immediate. This might cause the student to not understand where a value that is bound to a name actually came from when this transition is not readily visible. With regard to the variable view two reviewers also wondered why the global scope was hidden behind a tab control instead of being always visible. As explained in Chapter 4 this was done to save screen space and because we did not see the global scope being inspected often if ever.

Another feature that leaves the current version of Jype at a notable disadvantage with regard to some program visualization tools is Jype's lack of a visualization for the evaluation of expressions and some reviewers did mention this as well. Instead, Jype only traces execution line-by-line in a similar fashion as conventional source-level debuggers. As discussed in Chapter 4 a benefit of the current tracing implementation in Jype is that it is less prone to break with new versions of the embedded implementation of Python.

With these failings of the visualizations in mind, a few reviewers regarded Jype's integration with the Matrix visualization framework with its sophisticated data structures as the key contribution of the tool. Indeed, this allows Jype to be used for creating algorithm visualizations with very little effort. With the exception of the Matrix visualizations a reviewer summarized that "from the program animation point of view, visualization [in Jype] can be considered quite simple".

Finally, a few reviewers missed object visualizations beyond the simple tree view that shows the attributes of objects. This was out of scope in this work, however, we can envision creating a separate visualizer component that would give a more abstract depiction of objects in addition to the existing tree representation in the general table view. In terms of implementation this would simply be another view into the execution and could be accomplished via the existing common visualizer interface. As far as new views are concerned, one reviewer also suggested looking into adding textual and aural explanations of the execution.

5.2 Interaction in Jype

All reviewers agreed that the interface of Jype is self-explanatory and that it is very easy to transition from writing to executing/animating code and vice versa. The ability to step back in the execution was also considered a useful feature at least in theory – some doubted if it will be of use in practice.

Many reviewers suggested keyboard shortcuts and breakpoints to be added and these are valid proposals that had indeed already been planned as probable additions when the tool is developed further. Another such feature is the support for editing and tracing code in multiple tab panes. A more interesting suggestion was adding functionality for running the program up to a line. In practice, this would merely be a one-time breakpoint, however, there is much potential to make this more intuitive than breakpoints with an intelligent user interface. We can imagine a student benefiting from the ability to immediately step the execution to the line where they have just modified the code – without the overhead and complexity of 1) adding a breakpoint, 2) running the program to that breakpoint and 3) removing the breakpoint.

Finally, two reviewers were of the opinion that Jype absolutely required a 'play' control which would start running the program step-by-step with an adjustable delay between the transitions. In fact, this functionality was implemented and the control was then subsequently disabled because we see no practical use for it. It is always much simpler and intuitive to just step through the execution at your own pace. The only legitimate possible use scenario we could come up with was on lectures when the teacher is not able to access the control while simultaneously explaining what is happening in the execution. However, currently Jype is not primarily designed for this type of use, though, maybe the control should be optionally enabled. Another way the functionality could be used is to quickly step through some lines of code but this, on the other hand, is simply a poor replacement for a proper way of directly stepping the execution to a specific line in the way as discussed above.

5.3 Discussion of Goals

Overall, most reviewers found Jype to be applicable for the learning scenarios it was designed for but noted that there are many other systems with similar goals and partially similar functionality, although very few currently provide support for Python. Referring to the visualizations, the reversible debugging functionality and the programming exercises with automatic feedback one reviewer summarized:

> "There are many related web-based applications. However, Jype is probably the only one containing the three features given above."

However, one system was found to exist that provides a very similar feature set (VIP). Finally, commenting on the concept as a whole, two reviewers thought that integrating visual debugging and automatically assessed programming exercises, which is indeed one of the more unique features of Jype, was an interesting idea that should be examined further.

In Section 3.1 we set 5 primary goals for our tool. In brief these were: 1) facilitate program comprehension and 2) aid in debugging with visualizations and reverse navigation of a program's execution, 3) provide programming exercises with automatic feedback, 4) ease of use and 5) web deployment with a low barrier to entry. Goals 3, 4 and 5 can be readily regarded as met. This is also evident from the expert evaluations. However, goals 1 and 2 deal with higher level aspirations than mere functionality and ultimately require experiments with the tool's end users, *i.e.* students, to evaluate the effectiveness of the provided visualizations and interaction with regard to actual learning. This is an obvious continuation to the research started in this work and was also stressed in many of the evaluations that expressed the functionality to be adequate for the designed use and stated that more than anything else we now need experiences from real use. Indeed, as a reviewer succinctly put it: "the tool does what is expected". Along the same lines, another reviewer warned about the possibility of more information making for less clarity and said that the tool should be formally evaluated before adding more complexity.

One reviewer did, however, offer an opposite view. He felt existing tools "Jeliot and BlueJ to be sufficient for teaching introductory programming" and did not a see a need for another competing tool. Overall, he believed Jeliot 3 to simply be a much better tool for learning CS concepts than Jype with its current visualizations – so why even bother with developing it? Indeed, he suggested moving the focus of the tool's goals more clearly towards CS2 and the teaching of algorithms and data structures. On the other hand, as he was also quick to mention, in this area jGRASP is a pioneering system that provides a comprehensive IDE and advanced visualizations for Java data structures. The fact that the two tools mentioned are for Java and do not currently support Python can be regarded as a somewhat of a minor detail. However, BlueJ is a standalone IDE especially aimed at providing class visualizations and an easy interface for interacting with objects, which is far from the intended focus of Jype that is first and foremost on tracing and writing code and understanding program state in terms of control and data flow. In fact, using

the objects-first teaching approach with BlueJ exclusively has been reported to leave students with a vague concept of control flow [55, 106]. Jeliot 3, on the other hand, which can also be used as a plugin in BlueJ, does provide advanced visualizations of the progress of execution but still does not engage the student in any way, whereas in Jype students are to write and debug solutions to programming exercises with the support of automatic feedback. All in all, we must conclude that he simply did not agree with the premiss that there is a need and use for the type of tool Jype is.

Table 5.1 gives a summary of the functionality in Jype in relation to other tools brought up in the discussion. The overall goal for Jype was an integrated tool – with a low barrier to entry – for writing and debugging Python programming exercises with the help of supporting automatically generated visualizations. While there is always room for improvement and ultimately experiments with students will determine the tool's actual usefulness, this goal has been achieved.

	Jype	VIP	ViLLE	Jeliot 3	JIVE	jGRASP	BlueJ
Web deployment	X	X	X	X			
Python language support	X		(X)[1]				
Programming exercises with automatic feedback	X	X					
Visual debugging	X	X	(X)[2]	X	X	X	X
Reverse stepping	X		X	(X)[3]	X		
Data structure visualizations	X					XXX[4]	
Visualization of expression evaluation		X		X			
Smooth animation of data flow				X			
Object interaction						X	X

Table 5.1: Jype's features compared to a selection of related tools

[1] The limited supported set of Java can to an extent be automatically translated into Python with builtin substitution rules (see Section 3.2.1).

[2] Java code can be edited and animated within the confines of the limited support for language features (see Section 3.2.1).

[3] There is a separate history view that provides screenshots of the visualizations of past states.

[4] jGRASP provides comparatively more advanced visualizations of data structures with specific visualizations for standard Java library collection data types and also more general structural views.

Chapter 6

Conclusions

From our review of research on teaching introductory programming we determined the need for an educational tool that specifically targets students' apparent fragile knowledge of elementary programming which manifests as difficulties in tracing and writing even simple programs. We concluded that a tool that tightly integrates programming tasks with visualizations of program execution and lets students practice writing code and easily transition to visually tracing it in order to locate programming errors would help to address this problem. In addition, recent research into educational visualization of programming concepts indicates a need for including an active component in visualization tools in order to properly engage students into examining visualizations and thus enable them to actually learn from them. Therefore, the tool would also have to let students get feedback on their progress in the programming tasks.

In our review of existing supporting tools for teaching introductory programming, we found one tool for C++ that integrates programming exercises and program visualization to provide an integrated programming exercise and visual debugging environment (VIP). We found very few tools specifically designed for Python programming education.

To address the described needs on our future introductory programming courses taught in Python, we implemented a web-based (Java Applet / Web Start) easy-to-use tool, named as Jype, that provides an environment for effortlessly visualizing the line-by-line execution of Python programs, and for solving programming exercises with the support of immediate automatic feedback and an integrated debugger. Moreover, the debugger allows the student to step back in the visualization of the execution as

if executing in reverse. This lets students to backtrack if they get lost when tracing a program, for example, when attempting to locate an error in their solution. In this regard Jype is a rather unique tool. We also integrated Jype into the TRAKLA2 course management system which can be used to record and track students' points and submissions. Finally, Jype supports the full Python (2.5) language as implemented in the Jython library and also integrates the Matrix visualization framework, which allows the effortless creation of algorithm visualizations with sophisticated depictions of data structures. This extends the scope of Jype beyond CS1.

An initial qualitative evaluation supported by reviews from experts in the field of programming education tools seems to confirm that in terms of functionality we have met our goals – Jype is a tool with a low barrier to entry that lets effortlessly transition between writing programs and observing program behaviour with the help of automatically generated visualizations. While the effectiveness of the tool in improving learning has to be formally evaluated, we believe that when it will be used to provide students with a large set of gradually more difficult programming tasks that let them efficiently practice a number of different mechanics and programming idioms, this will help in building students' confidence in programming, aid them in getting a good grasp of basic syntax and semantics of the programming language, and ultimately improve students' understanding of program execution. After this, students are better equipped to move on to learning program design and higher level problem solving skills.

6.1 Future Work

First, Jype's effectiveness in terms of its intended purpose, that is, in improving the learning of programming, must be evaluated with students. While this probably will not result in a complete rejection of the tool, it will most likely spring up many ideas for refinement of its visualizations, interaction and the concept overall. Especially, the effects of the tight integration of automatic assessment with a visual debugger should be examined.

Some planned minor enhancements could be made to the tool immediately as discussed in Chapter 5. This includes adding breakpoints and support for editing and tracing multiple code files in different tab panes. Furthermore, we could provide functionality for annotating example programs and the boilerplate code in exercises in other ways than simply using Python comments. For example, Jype could recognize special instructions within the code, discard them from the editor view, and show in some

more interactive and visually pleasing fashion. A more fundamental change that would require changing the internals of the tool is adding the capability of tracing expression evaluations. Similarly, proper implementation of smooth animation would require extensive development on the animator component provided by the Matrix visualization framework.

Jype integrates many areas of programming tools – code and data visualization, code editing, debugging, automatic assessment, and data structure visualizations – and it is not quite state of the art in any of these subcategories. However, now that the infrastructure for controlled execution and run-time data extraction is in place, Jype can act as a platform for further studies on Python visualization, automatic assessment, and novice programming behavior. Indeed, due to its integrated nature and deployment on the web, the tool could easily be used to record a log of programming activities to study students' patterns of code writing and debugging. As mentioned in Chapter 5, we could also experiment with adding textual and aural explanations of execution. Other ideas that spring to mind are, for example, implementing a scheme to carry out peer reviews via the tool instead of or in addition to plain automatic assessment, or even online collaboration and co-operative programming. Furthermore, we could develop support for different types of exercises and assessment, such as, Web-CAT-style test-driven exercises, fill-in-the-gap type assignments, or pop-up prediction questions on the execution.

The tool could also, to an extent, be used to serve the needs of CS0 with microworld assignments that include an abstract visualization which in the GUI would take a similar role as the data structure visualizations currently. This would only require a simple Python or Java implementation of the microworld, a simple visualizer that gives an abstract view of the elements in its environment, and perhaps some supporting Python code for easy authoring of unit tests for this world. All this should be fairly straightforward and even easy to implement within Jype. It might even be possible to adapt an existing microworld written in Java or Python into Jype.

Alternatively, Jype's focus could be moved towards CS2 with more advanced IDE controls and visual navigation aids for handling larger files, such as, search and replace, code folding, code outline, integrated versioning, and code completion. With more code it would also be increasingly important to include automatic feedback on code style and other metrics as well.

In fact, we could have functionality on three different tiers. Each mode of operation would comprise a different set of controls and visualization components in order of

increasing complexity and detail of information: the CS0 microworld, CS1 elementary programming and the CS2 IDE with a feature set set approaching the feel of a typical stand-alone programming environment.

In terms of the deployment on programming courses there are also many possible directions for improvement. We could build exercise management features directly into Jype, such as, authoring of exercises and importing/exporting them. Furthermore, the tool could be integrated to other course and learning management systems besides TRAKLA2, even a CSE-specific LMS that could support semi-automatic assessment of coding exercises with instructors commenting and annotating code to give feedback on the submitted programs. Jype could also be developed for other use scenarios, such as, demonstrating program behavior on lectures by adding functionality for storing and annotating animations. Furthermore, when we have confirmed that the use of the tool positively correlates with learning results, we could also develop Jype towards use in grading, in which case we must implement preventive measures for and/or detection of plagiarism.

Finally, we could look into adapting Jype for other programming languages, such as, Ruby with JRuby and Javascript with Rhino. However, the benefits of using Jype as a starting point would probably be quite low since a major part of the work has gone into interacting with Jython and extracting and interpreting Python trace data. The GUI components could be reused, however, those are purposely quite simple, and, on the other hand, Sun is moving off from Swing and another GUI solution could in this regard be justifiable in future projects.

All in all, there are quite a few possible future directions for this work.

References

[1] T. Ahoniemi and T. Reinikainen. ALOHA – A Grading Tool for Semi-automatic Assessment of Mass Programming Courses. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 139–140. ACM Press, New York, NY, USA, 2006.

[2] A. Ahtiainen, S. Surakka, and M. Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 141–142. ACM Press, New York, NY, USA, 2006.

[3] K. Ala-Mutka and H.M. Järvinen. Assessment Process for Programming Assignments. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*, pages 181–185. IEEE Computer Society, Washington, DC, USA, 2004.

[4] K.M. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102, 2005.

[5] J.R. Anderson and B.J. Reiser. The LISP tutor: it approaches the effectiveness of a human tutor. *BYTE*, 10(4):159–175, 1985.

[6] D. Arnow and O. Barshay. WebToTeach: an interactive focused programming exercise system. In *Frontiers in Education Conference, 1999. FIE'99. 29th Annual*, volume 1, 1999.

[7] B.S. Baker. Parameterized Pattern Matching: Algorithms and Applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

[8] B.S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[9] B.S. Baker, L. Wills, P. Newcomb, and E. Chikofsky. On Finding Duplication and Near-Duplication in Large Software Systems. In *Second Working Conference on Reverse Engineering*, pages 86–95. IEEE Computer Society, Washington, DC, USA, 1995.

[10] R.S. Baker, M. Boilen, M.T. Goodrich, R. Tamassia, and B.A. Stibel. Testers and visualizers for teaching data structures. *SIGCSE Bulletin*, 31(1):261–265, 1999.

[11] M. Ben-Ari. Constructivism in computer science education. *SIGCSE Bulletin*, 30(1):257–261, 1998.

[12] M. Ben-Ari. Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[13] R. Ben-Bassat Levy, M. Ben-Ari, and P.A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

[14] J. Bennedsen and M.E. Caspersen. Failure rates in introductory programming. *SIGCSE Bulletin*, 39(2):32–36, 2007.

[15] H.L. Berghel and D.L. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65–76, 1984.

[16] D. Binkley. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society, Washington, DC, USA, 2007.

[17] D. Blank, D. Kumar, L. Meeden, and H. Yanco. Pyro: A python-based versatile programming environment for teaching robotics. *Journal on Educational Resources in Computing (JERIC)*, 3(4), 2003.

[18] D.G. Bobrow, R.P. Gabriel, and J.L. White. Clos in context: the shape of the design space. *Object-oriented programming: the CLOS perspective*, pages 29–61, 1993.

[19] V. Bonifaci, C. Demetrescu, I. Finocchi, G.F. Italiano, and L. Laura. Portraying Algorithms with Leonardo Web. *Lecture notes in Computer Science*, 3807:73, 2005.

[20] C.M. Boroni, T.J. Eneboe, F.W. Goosey, J.A. Ross, and R.J. Ross. Dancing with DynaLab: endearing the science of computing to students. In *Proceedings of the*

twenty-seventh SIGCSE technical symposium on Computer science education, pages 135–139. ACM, New York, NY, USA, 1996.

[21] S. Bridgeman, M.T. Goodrich, S.G. Kobourov, and R. Tamassia. PILOT: An Interactive Tool for Learning and Grading. *SIGCSE Bulletin*, 32(1):139–143, 2000.

[22] D. Buck and D.J. Stucki. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 16–20. ACM, New York, NY, USA, 2001.

[23] B. Cheang, A. Kurnia, A. Lim, and W.C. Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[24] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. *Proceedings of the 2003 ACM symposium on Software visualization*, 2003.

[25] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000.

[26] J.H. Cross, D. Hendrix, and D.A. Umphress. jGRASP: an integrated development environment with visualizations for teaching java in CS1, CS2, and beyond. *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages 1466–1467, 2004.

[27] J.H. Cross, T.D. Hendrix, and L.A. Barowski. Integrating Multiple Approaches for Interacting with Dynamic Data Structure Visualizations. *Proceedings of the Fifth Program Visualization Workshop*, pages 3–10, 2008.

[28] J.H. Cross, T.D. Hendrix, L.A. Barowski, and J. Jain. Dynamic Object Viewers for Java. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)-Volume 02*, pages 374–375, 2006.

[29] J.K. Czyz and B. Jayaraman. Declarative and Visual Debugging in Eclipse. *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 31–35, 2007.

[30] N.B. Dale. Most difficult topics in CS1: results of an online survey of educators. *SIGCSE Bulletin*, 38(2):49–53, 2006.

[31] C. Daly and J. Horgan. A Technique for Detecting Plagiarism in Computer Code. *The Computer Journal*, 48(6):662–666, 2005.

[32] C. Daly and J.M. Horgan. An Automated Learning System for Java Programming. *IEEE Transactions on Education*, 47(1):10–17, 2004.

[33] C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing Algorithms over the Web with the Publication-Driven Approach. *Lecture Notes in Computer Science*, 1982:147, 2001.

[34] F. Détienne. *Software Design – cognitive Aspects*. Springer, 2002.

[35] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[36] J.L. Donaldson, A. Lancaster, and P.H. Sposato. A plagiarism detection system. *SIGCSE Bulletin*, 13(1):21–25, 1981.

[37] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 2005.

[38] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human–Computer Studies*, 51(2):265–277, 1999.

[39] S. Edwards. Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA*, volume 3, 2003.

[40] S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.

[41] S.G. Eick, J.L. Steffen, and E.E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[42] S. Fitzgerald. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[43] S. Fitzgerald, B. Simon, and L. Thomas. Strategies that students use to trace code: an analysis based in grounded theory. *Proceedings of the 2005 international workshop on Computing education research*, pages 69–80, 2005.

[44] A.E. Fleury. Parameter passing: the rules the students construct. *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 283–286, 1991.

[45] A.E. Fleury. Programming in Java: student-constructed rules. *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 197–201, 2000.

[46] P. Gestwicki and B. Jayaraman. Interactive Visualization of Java Programs. *Human-Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 226–235, 2002.

[47] P. Gestwicki and B. Jayaraman. Methodology and Architecture of JIVE. *Proceedings of the 2005 ACM symposium on Software visualization*, pages 95–104, 2005.

[48] P.V. Gestwicki. Interactive Visualization of Object-Oriented Programs. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 48–49, 2004.

[49] P.V. Gestwicki and B. Jayaraman. JIVE: java interactive visualization environment. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 226–228, 2004.

[50] H.Z. Girgis, B. Jayaraman, and P.V. Gestwicki. Visualizing Errors in Object-Oriented Programs. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 156–157, 2005.

[51] K.E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 170–177. ACM, New York, NY, USA, 2003.

[52] S. Grier. A tool that detects plagiarism in Pascal programs. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pages 15–20. ACM Press, New York, NY, USA, 1981.

[53] S. Grissom, M.F. McNally, and T. Naps. Algorithm visualization in CS education: comparing levels of student engagement. *Proceedings of the 2003 ACM symposium on Software visualization*, pages 87–94, 2003.

[54] M. Guzdial. A media computation course for non-majors. *SIGCSE Bulletin*, 35(3):104–108, 2003.

[55] H. Hegna and A. Groven. Stumbling thru' with Objects First: Some Observations from a Study of Objects First with BlueJ in a non-CS Context. In *ECOOP05, 9th Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts*, Glasgow, UK, 2005.

[56] T.D. Hendrix. jGRASP: a lightweight IDE with dynamic object viewers for CS1 and CS2. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 356–356, 2006.

[57] T.D. Hendrix, J.H. Cross, and L.A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. *SIGCSE Bulletin*, 36(1):387–391, 2004.

[58] T.D. Hendrix, D.A. Umphress, and L.A. Barowski. Exploring Accessibility and Visibility Relationships in Java. *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 103–108, 2008.

[59] R.R. Henry, K.M. Whaley, and B. Forstall. The University of Washington illustrating compiler. *ACM SIGPLAN Notices*, 25(6):223–233, 1990.

[60] C.A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 2005.

[61] J. Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, 1960.

[62] 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.

[63] D. Jackson and M. Usher. Grading student programs using ASSYST. *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339, 1997.

[64] J. Jain, J.H. Cross, T.D. Hendrix, and L.A. Barowski. Experimental Evaluation of Animated-verifying Object Viewers for Java. *Proceedings of the 2006 ACM symposium on Software visualization*, pages 27–36, 2006.

[65] M. Joy, N. Griffiths, and R. Boyatt. The boss online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 2005.

[66] V. Karavirta and A. Korhonen. Automatic tutoring question generation during algorithm simulation. *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 95–100, 2006.

[67] V. Karavirta, A. Korhonen, and L. Malmi. Taxonomy of Algorithm Animation Languages. *Proceedings of the 2006 ACM symposium on Software visualization*, pages 77–85, 2006.

[68] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke. MatrixPro-a tool for demonstrating data structures and algorithms ex tempore. *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, pages 892–893, 2004.

[69] I.R. Katz and J.R. Anderson. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction*, 3(4):351–399, 1987.

[70] P. Kinnunen and L. Malmi. Why students drop out CS1 course? *Proceedings of the 2006 international workshop on Computing education research*, pages 97–108, 2006.

[71] P. Kinnunen and L. Malmi. CS Minors CS1 course? *Proceedings of the 2008 international workshop on Computing education research*, 2008.

[72] M. Kölling. Greenfoot: a highly graphical ide for learning object-oriented programming. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 327–327. ACM, New York, NY, USA, 2008.

[73] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ System and its Pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[74] A. Korhonen, L. Malmi, P. Myllyselkä, and P. Scheinin. Does it Make a Difference if Students Exercise on the Web or in the Classroom? In *Proceedings of The 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology*

in Computer Science Education, ITiCSE'02, pages 121–124. ACM Press, New York, NY, USA, 2002.

[75] A. Korhonen, L. Malmi, P. Silvasti, V. Karavirta, J. Lönnberg, J. Nikander, K. Stålnacke, and P. Ihantola. Matrix – a framework for interactive software visualization. Research Report TKO-B 154/04, Laboratory of Information Processing Science, Department of Computer Science and Engineering, Helsinki University of Technology, Finland, 2004.

[76] A. Korhonen, E. Sutinen, and J. Tarhio. Understanding Algorithms by Means of Visualized Path Testing. In *Software Visualization: International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001: Revised Papers*, page 256. Springer, 2002.

[77] J.L. Korn and A.W. Appel. Traversal-based Visualization of Data Structures. *Information Visualization, 1998. Proceedings. IEEE Symposium on*, pages 11–18, 1998.

[78] M. Krebs, T. Lauer, T. Ottmann, and S. Trahasch. Student-built algorithm visualizations for assessment: flexible generation, feedback and grading. *SIGCSE Bulletin*, 37(3):281–285, 2005.

[79] M.J. Laakso, E. Kaila, T. Rajala, and T. Salakoski. Define and Visualize Your First Programming Language. In *Advanced Learning Technologies, 2008. ICALT'08. Eighth IEEE International Conference on*, pages 324–326, 2008.

[80] M.J. Laakso, L. Malmi, A. Korhonen, T. Rajala, and E. Kaila. Using Roles of Variables to Enhance Novice's Debugging Work. *Setting Knowledge Free: The Journal of Issues in Informing Science and Information Technology Volume 5, 2008*, 5, 2008.

[81] S.P. Lahtinen, E. Sutinen, and J. Tarhio. Automated Animation of Algorithms with Eliot. *Journal of Visual Languages and Computing*, 9(3):337–349, 1998.

[82] T. Lancaster and F. Culwin. A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education*, 14(2):101–112, 2004.

[83] T. Lehtonen. Javala – Addictive E-Learning of the Java Programming Language. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 41–48. University of Joensuu, November 2005.

[84] R. Lister. Teaching Java first: experiments with a pigs-early pedagogy. In *Proceedings of the sixth conference on Australasian computing education-Volume 30*, pages 177–183. Australian Computer Society, Darlinghurst, Australia, 2004.

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

[86] K.P. Lohr and A. Vratislavsky. Jan – Java Animation for Program Understanding. *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*, pages 67–75, 2003.

[87] J.I. Maletic, A. Marcus, and M.L. Collard. A task oriented view of software visualization. *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 32–40, 2002.

[88] L. Malmi, V. Karavirta, A. Korhonen, and J. Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 2005.

[89] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

[90] M. McCracken, T. Wilusz, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y.B.D. Kolikant, C. Laxer, L. Thomas, and I. Utting. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[91] I. Milne and G. Rowe. Difficulties in Learning and Teaching Programming – Views of Students and Tutors. *Education and Information Technologies*, 7(1):55–66, 2002.

[92] A. Mitrovic and S. Ohlsson. Evaluation of a Constraint-Based Tutor for a Database Language. *International Journal of Artificial Intelligence in Education*, 10(3-4):238–256, 1999.

[93] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. *Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, 2004.

[94] N. Myller. Automatic Generation of Prediction Questions during Program Visualization. *Electronic Notes in Theoretical Computer Science*, 178:43–49, 2007.

[95] T.L. Naps. JHAVÉ – Addressing the Need to Support Algorithm Visualization with Tools for Active Engagement. *IEEE Computer Graphics and Applications*, 25(6):49–55, 2005.

[96] T.L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, et al. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, 2003.

[97] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[98] J. Nikander. Managing automatically assessed exercises in TRAKLA2. Master's thesis, Helsinki University of Technology, 2005.

[99] E. Odekirk-Hash and J.L. Zachary. Automated feedback on programs means students need less help from teachers. *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 55–59, 2001.

[100] R. Oechsle and T. Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). *Lecture Notes In Computer Science; Vol. 2269*, pages 176–190, 2001.

[101] K.J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bulletin*, 8(4):30–41, 1976.

[102] N. Parlante. Nifty reflections. *SIGCSE Bulletin*, 39(2):25–26, 2007.

[103] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bulletin*, 39(4):204–223, 2007.

[104] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[105] B.A. Price, R. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[106] N. Ragonis and M. Ben-Ari. On understanding the statics and dynamics of object-oriented programs. *SIGCSE Bulletin*, 37(1):226–230, 2005.

[107] K.A. Reek. The TRY system - or - how to avoid testing student programs. *SIGCSE Bulletin*, 21(1):112–116, 1989.

[108] A. Robins, J. Rountree, and N. Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.

[109] G.C. Roman and K.C. Cox. A Taxonomy of Program Visualization Systems. *IEEE Computer*, 26:11–24, 1993.

[110] G. Rößling and S. Hartte. Webtasks: online programming exercises made easy. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 363–363. ACM, New York, NY, USA, 2008.

[111] G. Rößling and G. Haussge. Towards tool-independent interaction support. *Proceedings of the Third Program Visualization Workshop*, pages 110–117, 2004.

[112] G. Rößling and S. Schneider. An Integrated and "Engaging" Package for Tree Animations. *Electronic Notes in Theoretical Computer Science*, 178:69–78, 2007.

[113] G. Rößling, M. Schüer, and B. Freisleben. The ANIMAL algorithm animation tool. *SIGCSE Bulletin*, 32(3):37–40, 2000.

[114] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. *SIGCSE Bulletin*, 33(3):133–136, 2001.

[115] J. Sajaniemi. From Procedures to Objects: What Have We (Not) Done? In *PPIG07, 19th Annual Psychology of Programming Workshop*, Joensuu, Finland, 2007.

[116] J. Sajaniemi and C. Hu. Teaching Programming: Going beyond "Objects First". In *PPIG06, 18th Annual Psychology of Programming Workshop*, Brighton, UK, 2006.

[117] D. Sanders and B. Dorn. Jeroo: a tool for introducing object-oriented programming. *SIGCSE Bulletin*, 35(1):201–204, 2003.

[118] K. Sanders and L. Thomas. Checklists for grading object-oriented CS1 programs: concepts and misconceptions. *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 166–170, 2007.

[119] P. Saraiya, C.A. Shaffer, D.S. McCrickard, and C. North. Effective features of algorithm visualizations. *SIGCSE Bulletin*, 36(1):382–386, 2004.

[120] O. Seppälä. Program state visualization tool for teaching CS1. *Proceedings of the Third Program Visualization Workshop*, pages 59–63, 2004.

[121] S.C. Shaffer. Ludwig: an online programming tutoring and assessment system. *SIGCSE Bulletin*, 37(2):56–60, 2005.

[122] J. Sheard, M. Dick, S. Markham, I. Macdonald, and M. Walsh. Cheating and plagiarism: perceptions and practices of first year IT students. *SIGCSE Bulletin*, 34(3):183–187, 2002.

[123] B. Shneiderman, S.K. Card, J.D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[124] B. Simon, R. Lister, and S. Fincher. Multi-Institutional Computer Science Education Research: A Review of Recent Studies of Novice Understanding. *Frontiers in Education Conference, 36th Annual*, pages 12–17, 2006.

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

[126] J. Sorva and L. Malmi. An object testing tool for CS1, 2005.

[127] J.C. Spohrer and E. Soloway. Analyzing the High Frequency Bugs in Novice Programs. *Empirical Studies of Programmers*, 1986.

[128] J.C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986.

[129] J.T. Stasko. Tango: A Framework and System for Algorithm Animation. *Computer*, 23(9):27–39, 1990.

[130] M.A. Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proceedings of the 13th International Workshop on*

Program Comprehension, pages 181–191. IEEE Computer Society, Washington, DC, USA, 2005.

[131] N. Sumner, D. Banu, and H. Dershem. JSAVE: Simple and Automated Algorithm Visualization Using the Java Collection Framework. *Tenth Annual Consortium for Computing Sciences in Colleges, October*, 2003.

[132] E. Sutinen, J. Tarhio, and T. Teräsvirta. Easy Algorithm Animation on the Web. *Multimedia Tools and Applications*, 19(2):179–194, 2003.

[133] E.R. Sykes and F. Franek. A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java. *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education, June 30-July 2, 2003, Rhodes, Greece*, pages 78–83, 2003.

[134] R. Tamassia, M.T. Goodrich, L. Vismara, M. Handy, R. Cohen, B. Hudson, R.S. Baker, N. Gelfand, G. Shubina, M. Boilen, et al. An Overview of JDSL 2.0, the Data Structures Library in Java, 2005.

[135] J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, et al. Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4(1):143–162, 2005.

[136] N. Truong, P. Bancroft, and P. Roe. A web based environment for learning to program. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 255–264. Australian Computer Society, Darlinghurst, Australia, 2003.

[137] N. Truong, P. Bancroft, and P. Roe. Learning to program through the web. *SIGCSE Bulletin*, 37(3):9–13, 2005.

[138] D.W. Valentine. Cs educational research: a meta-analysis of sigcse technical symposium proceedings. *SIGCSE Bulletin*, 36(1):255–259, 2004.

[139] A.T. Virtanen, E. Lahtinen, and H.M. Järvinen. VIP, A Visual Interpreter for Learning Introductory Programming with C++. *Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 125–130, 2005.

[140] L. Voinea, A. Telea, and J.J. van Wijk. CVSscan: visualization of code evolution. *Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, 2005.

[141] E. von Glasersfeld. A constructivist approach to teaching. *Constructivism in education*, 3:15, 1995.

[142] J. Whalley, C. Prasad, and P.K.A. Kumar. Decoding doodles: novice programmers and their annotations. In *ACE '07: Proceedings of the ninth Australasian conference on Computing education*, pages 171–178, Ballarat, Victoria, Australia, 2007. Australian Computer Society, Darlinghurst, Australia.

[143] M.J. Wise. Yap3: improved detection of similarities in computer program and other texts. *SIGCSE Bulletin*, 28(1):130–134, 1996.

[144] D. Woit and D. Mason. Effectiveness of Online Assessment. *SIGCSE Bulletin*, 35(1):137–141, 2003.

Appendix A

An Exercise Definition in Jype

In the following we give an example of how a simple exercise 'count_evens' could be defined in Jype.

A.1 count_evens.py

This is the primary source file of an exercise 'count_evens'. The docstring of this file defines the instructions given to the student. Everything after the special comment '#BEGIN' is initially shown to the student in the code editor as a starting point for solving the exercise.

```
'''Return the number of even integers in the given list.'''

#BEGIN
def count_evens(numbers):
  pass
```

A.2 test_count_evens.py

This source file contains the tests for the exercise 'count_evens'. The file must define a variable suite that provides suite of tests for exercising the functionality of the student's code. The 'score' function decorator can be used to assign specific scores to individual tests.

```
import unittest
from test_support import score, JypeHelperTestCase
import count_evens as student

class CountEvensTestCase(JypeHelperTestCase):

  def test_1245(self):
    self.assert_count_evens(3, [1,2,4,5,6])

  def test_one_uneven(self):
    self.assert_count_evens(0, [21])

  def test_one_even(self):
    self.assert_count_evens(1, [42])

  @score(20)
  def test_large(self):
    self.assert_count_evens(5, [234,67452,88762,987,12,68,2343])

  @score(20)
  def test__(self):
    self.assert_count_evens(0, [])

  def assert_count_evens(self, expected, *args):
    self.assert_return_value(expected, student.count_evens, *args)

suite = unittest.TestLoader().loadTestsFromTestCase(CountEvensTestCase)
```

A.3 test_support.JypeHelperTestCase

This is a supporting class used in the tests for the exercise 'count_evens'. It provides a method that can be used to test the return value of a function when given the expected value, the function object and the arguments for the function. It sets the docstring for the test which is shown as a test description when the test is run and carries out an assert operation.

```
class JypeHelperTestCase(unittest.TestCase):
  def assert_return_value(self, expected, func, *args):
    if (self._testMethodDoc == None):
        self._testMethodDoc = ''
    self._testMethodDoc += func.func_name + '('
    first = True
    for arg in args:
      if (first):
          first = False
      else:
        self._testMethodDoc += ','
      self._testMethodDoc += str(arg)
    self._testMethodDoc += ') -> '
    result = func(*args)
    self._testMethodDoc += str(result)
    self.assertEqual(expected, result, "Expected '" + str(expected) + /
                     "' but got '" + str(result) + "'!")
```

Appendix B

Jype Evaluation

B.1 Extended Abstract

First programming courses often have high drop-out rates. In a recent study Kinnunen and Malmi reported a rate of 26 percent at their institution [1] and in general the rate at many institutions is estimated to be at 20-40 percent. Multi-national studies have also found that even those that pass have great difficulty in implementing and understanding even simple programs [2,3]. An ITiCSE 2004 working group led by Lister came to the conclusion that many students are hindered by their inability to trace and understand code, and by their overall fragile knowledge of basic programming constructs [4]. Furthermore, Kinnunen and Malmi reported in their research on the drop-outs that finding run-time errors was the most difficult programming-related issue, and that students spent frustratingly much time on tracing even relatively simple errors to their causes [1,5]. A related study confirms that locating a bug is the actual challenge in the debugging process [6].

The implications are that the hidden aspects of data and control flow must be made explicitly visible to the novices on an appropriate level of abstraction to let them properly learn to trace program state, to find errors effectively and to prevent fragile knowledge from evolving into misconceptions about basic programming constructs. Ultimately, gaining program comprehension and writing skills requires practice – reading and writing lots of different types of programs. However, giving feedback on a large set of programs written by students or explaining programs by tracing them on lectures is not possible on large courses. Instead to solidify students' skills in basic programming we must provide students with a large set of varying types of small example programs and programming exercises that give automatic feedback

and are designed to challenge their understanding to prevent misconceptions.

In order to combat the problem novice programmers are facing, we developed a tool, whose purpose is to gradually build students' knowledge and confidence in programming and to ensure they become well-versed in elementary programming before moving on to issues of program design and architecture on subsequent assignments and courses. The goals for the tool were:

1. Facilitate the development of an accurate mental model of program state and execution through consistent automatically generated visualizations.
2. Aid in tracking down the causes of programming errors and possible underlying misconceptions with reversible visual source code level debugging functionality.
3. Provide automatically assessed programming assignments to enable and support the learning of programming, in the sense of actually writing code, by practice and repetition.
4. In achieving the goals 1-3 add as little overhead as possible to the actual process of writing program code.
5. Minimize the barrier to entry and facilitate wide adoption by implementing the system as an easy-to-use web application, which also allows it to be easily updated and distributed, and to fully support distance learning.

In brief, the goal was to develop a tool to aid in teaching Python programming and specifically in developing students' program tracing skills. For this purpose we developed a program visualization and visual debugging tool for delivering automatically assessed Python programming exercises over the Web. It is intended to be used by students to write, debug and observe small programs – to let them expand and refine their comprehension of control and data flow by tracing programs. The current prototype delivers some simple examples of possible use scenarios and content, that is, short automatically assessed programming exercises and example programs for students to observe and learn from. As of now, these are primarily aimed at demonstrating the functionality of the tool rather than for actual learning material.

The tool is implemented in Java and can be deployed as a Java Applet, Web Start or a stand-alone application. Python code is executed and traced via Jython, an embedded Python Interpreter implemented in Java. The automatic feedback and assessment of

programming exercises is based on including unit tests with the assignment. Exercises can be deployed on the TRAKLA2 course management system for recording and tracking points and submissions.

1. P. Kinnunen and L. Malmi. CS Minors CS1 course? *Proceedings of the 2008 international workshop on Computing education research*, 2008.

2. B. Simon, R. Lister, and S. Fincher. Multi-Institutional Computer Science Education Research: A Review of Recent Studies of Novice Understanding. *Frontiers in Education Conference*, 36th Annual, pages 12–17, 2006.

3. J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, et al. Students designing software: a multinational, multi-institutional study. *Informatics in Education*, 4(1):143–162, 2005.

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

5. P. Kinnunen and L. Malmi. Why students drop out CS1 course? *Proceedings of the 2006 international workshop on Computing education research*, pages 97–108, 2006.

6. S. Fitzgerald. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

B.2 Evaluation Form

The evaluation consists of 6 separate sections that each contain a set of questions intended to guide the focus of the evaluation but the reviewer is not restricted to address these exactly with direct answers question by question.

B.2.1 The Rationale of the Tool

- What is the problem the tool adresses and how does it address it?
- What are the main functionalities/features provided by the tool?

- In which way does the tool differ from other tools solving the same or similar problem? What are the key contributions of the tool?

B.2.2 Visualization

- Do you find the visualizations provided for the code and data to be intuitive and correct? How would you improve them? What visualizations should be used instead or in addition to these?
- How well do the visualizations help in discerning control and data flow? What additional information should be visualized?
- How do you find the use of colors and screen real estate? How would you improve the visual layout?

B.2.3 Interaction

- How do you find the interface for executing/animating and writing programs? Are the controls intuitive and easily learned? Are there usability issues?
- How useful do you find the reversible execution feature? Do you think it helps in understanding the flow of execution and/or locating programming errors?
- What functionality is missing? What would you change or add?

B.2.4 Technical Quality

- If you tested the system, did you encounter any bugs or unexpected behaviour?
- Any comments or suggestions related to the technical aspects of the tool?

B.2.5 Goals

- How reasonable do you find the goals of this work (see the extended abstract for information about the goals and their motivation)? How well have they been met? Which ones should perhaps be re-evaluated?
- What are your suggestions with regard to further development guided by the goals?

B.2.6 Applicability

- How applicable do you find the tool to be for the described learning scenarios (see the extended abstract for details)?
- How beneficial do you find the integrated debugging and automatic assessment functionalities?
- Can you come up with any other possible use cases, either from the student's or the teacher's point of view?
- How do you find the tool's potential on further research or improving learning outcomes?
- If you were teaching a programming course would you be interested in using the tool on the course? What modifications or improvements would have to be made to be able to successfully integrate it into your curriculum?
- How would you continue the development of this tool? What aspects of the tool need most improvement? What do you find to be the strengths of the tool?