The Fundamental Design Issues of Jeliot 3

Niko Myller

20.2.2004

University of Joensuu
Department of Computer Science
Master’s Thesis

Abstract

Programming and algorithms are hard subjects to teach and learn. Especially novices seem to have problems to grasp the basic concepts of programming and algorithms. Software visualization has tried to provide help for teaching and learning these subjects. The Jeliot family is a group of program and algorithm visualization tools to help novices to form the new concepts of programming and algorithms. In this thesis, a new version of Jeliot, called Jeliot 3, and its design are introduced. The new version is meant especially for novice students who are learning programming in Java language. It visualizes the data and the control flow of the programs as well as the object-oriented concepts such as constructors and objects. In the design, we have used a modular approach that connects an interpreter and a visualization engine to each other. Our design allows the utilization of two existing systems, Jeliot 2000 as a visualization engine and DynamicJava as an interpreter, to create the basis for this new system. To connect these two systems, we have created a new intermediate language. This new language also allows a formation of different kinds of visualizations with relative ease making the system extensible. In the design of the visualization, we have used the results from studies conducted in the fields of educational psychology and software visualization, and come into the conclusion that the program visualization for novices should be as complete, continuous and consistent as possible. Jeliot 3 extends the visualization capabilities of previous members of the Jeliot family with object-oriented concepts.

ACM-Classification (ACM Computing Classification System, 1998 version): K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer Science Education*; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems – *Animations*; I.6.8 [**Simulation and Modeling**]: Types of Simulation – *Animation, Visual*

Keywords: Software Visualization, Program Animation, Jeliot, Novice Programmers

Preface

The study process of two and half years for my Master's degree is now almost finished. It has been a enjoyable but also exhausting time. During this process I have been able to meet several new people and there are several people that I would like to thank because without their support and help I would not have been able to reach my goal.

Firstly, I would like to thank Professor Erkki Sutinen, my supervisor, who has been supporting me with all the discussions during the whole process. His office and home has always been open for me to visit him for discussions and those moments have helped me to carry on with my work. I am also very grateful to my other supervisor, Professor Mordechai "Moti" Ben-Ari, who has been endlessly supporting and guiding me in the work with Jeliot 3. It has been very helpful and it has produced fruitful results.

Secondly, I am thankful unto the Department of Computer Science at University of Joensuu to be able to work there during my studies, unto staff of the department for cooperation, and especially unto the research group of Education Technology for creating the supportive atmosphere that is open to all kinds of ideas and discussions. I want to especially mention Andrés Moreno with whom I have developed Jeliot 3 and shared the joys and the sorrows during the development.

Thirdly, I wish to show my gratitude to my parents, their spouses and all my siblings for their support. My parents have given me good guidance during my growth and introduced into my life the right values that I can cherish, cultivate and be proud of.

Finally, I want to give my dearest thanks to my significant other, Sini, to whom I dedicate this work. Without her support, encouragement and also suppression when needed, I would have lost my confidence and motivation and been unable to accomplish my goal. I hope I can make up somehow for all the extra hours that I spent with my studies and not with you during the last two and half years.

Contents

1 Introduction

Programming and algorithms are hard subjects to study and teach. Lecturers have introduced many ways to assist their own lecturing and to help students to acquire new knowledge in these fields. The learning aids have varied from paper, pen and scissors to the static images on the slides. However, it appears most natural to illustrate the algorithm's or program's execution over time with an animation. Thus, for the last couple of decades the software visualization has been an active field of study (Baecker, 1981; Hundhausen et al., 2002). The research has concentrated on two overlapping fields: algorithm visualization and program visualization.

When considering the possibility to produce a visualization of an algorithm or a program, it may appear that the visualization is a superior way to illustrate their behavior. Especially, when students are learning algorithms or programming, this kind of a tool could occur to them as an excellent learning resource. This was probably the purpose of the developers designing and implementing the first software visualization tools. However, several empirical experiments have shown that an animation of the running algorithm only helps students to learn if it somehow cognitively engages them and is especially targeted for the particular user population (e.g. for novices) (Hundhausen et al., 2002; Petre, 1995; Petre and Green, 1993).

The results achieved in the previous studies have driven the researchers to reformulate their research questions. It has become clear that the media itself does not strongly affect the learning outcomes. The effect of visualization is in the organization of the content, in the way the subject is taught (i.e. are the students passive observers or active learners), how it is designed and how it suits the particular learner population (Smith and Webb, 2000; Korhonen, 2003; Grissom et al., 2003; Naps et al., 2003a). This is also in line with more general research done by educational psychologists in the area of multimedia learning (Mayer, 2001; Kozma, 2003).

In Jeliot, the idea is to involve the students to construct their own programs and at the same time engage them to examine a visual representation of their program's execution and to develop their program further. During this process they acquire a mental model of the computation that helps them to understand the constructs of programming. Furthermore, the model can be used to acquire new knowledge and the vocabulary used to discuss programs and programming concepts. Thus, the students are engaged with the

tool and are learning by doing and by constructing new knowledge in a constructivist way (Ben-Ari, 2001a).

1.1 Background

The development of the Jeliot family started almost ten years ago when the first system Eliot (Lahtinen et al., 1998) was implemented to help the production of algorithm animations. The name Eliot came from the Finnish word *Eliöt* or in longer form *Elävät oLIOT* meaning living organisms or objects but an anecdote tells that it could be an acronym from the words *"Err... LIving ObjecTs"* as well.

After Eliot two other systems have been developed, namely Jeliot I (Sutinen et al., 2003) and Jeliot2000 (Ben-Bassat Levy et al., 2003). All these systems have been implemented into different environments and a new version has been developed either to extend the possibilities for visualization or to support different user populations.

The development process of Jeliot has been research-oriented, meaning that all the versions have had their own research problems rising from the previous versions' design and empirical evaluations. In the first version, Eliot, the main goal was to ease the production of the algorithm animations. Jeliot I can be used on the Internet making the use of Jeliot distance independent. Jeliot 2000 was especially designed for novice learners, whereas Jeliot 3 is a generalization of the work done with Jeliot 2000.

This has led to the stage when the software has become product-like both usable and stable. During the development and evaluation cycle of Jeliot, it has been learned that there is no one best formula for all learning needs, but there should be several items in the learning environment from which the learner can select the ones she needs (Ben-Ari et al., 2002a). This means that we should give students the possibility to use different kinds of systems with various orientations leading to a stage where an extensible and modular system is needed as a basis for this development.

1.2 Research Questions and Structure of the Thesis

The main goal of this thesis is *to explain the structure of Jeliot 3 and to justify the design decisions made*. In order to understand all of the decisions presented in the thesis

the original requirements for the new version of Jeliot must be stated. The requirements are shown in the list below with short explanations. However, the rationale for most of these requirements will be described in more detail in the chapters 2 and 3.

- *The system must be easy to use*. This is a crucial feature for a software meant for novices. If the system is not easy to use, novices, in particular, will refuse to use the system.

- *The visualizations produced by the system should be consistent with the graphical notation in all cases.* As stated by Petre (1995), novices lack the understanding of how to read graphical displays and they do not understand well the secondary notation of the visual layout. This means that if the graphical notation given by the system is inconsistent, novices have a greater chance of acquiring an incorrect mental model of computation. This can affect radically the learning of programming.

- *The visualizations produced by the system should be complete and continuous.* This requirement was also one of the key features of Jeliot 2000 and it was found to be very important. For novices, nothing can appear from an unexpected source, because confusion hinders the learning.

- *The system should support the visualization of as large a subset of programs written in Java language as possible.* Although the system is intended for novice users, we want to support the possibility of staying with Jeliot longer than the first couple of weeks. The object-oriented programming is getting much attention and object-oriented languages such as Java are used as the first language in teaching programming. If programming is taught with an objects first approach, it means that even inheritance and polymorphisms can be discussed within the first courses of programming (Barnes and Kölling, 2003). However, it seems that these concepts are not easily grasped by novice programmers (Holland et al., 1997). Furthermore, at the moment there are only a few visualization systems that visualize object oriented concepts such as objects and inheritance. All these issues should be addressed in the Jeliot 3.

- *The system should be extensible internally and externally.* Extensibility has been a problem of the previous versions of Jeliot. Adding new features to the visualization has not been easy, thus with this new version the extensibility was one of

the issues. The only exception is Jeliot I having support for the visualization of a new data type but it lacked other features that would have been hard to add (see section 3.2).

The thesis is organized as follows; Chapters 2 and 3 review related work done in the field and the previous versions of Jeliot family. The rest of the thesis can be divided into chapters according to the requirements discussed above: modularity and extensibility (Chapters 4, 5 and 6) and user interface and visualization (Chapter 7). In Chapters 8 and 9, I give some future perspectives and the conclusions from my work. All these chapters together answer the questions of *why, how and for whom this new version of Jeliot was designed and developed and how it can be further developed*.

2 Literature Review

The field of software visualization has been actively researched during the last decade. In this chapter two relevant subfields of software visualization are defined. In the Section 2.2 a literature review is carried out and the most relevant work to this research is presented.

2.1 Definitions

There are no established conventions in the literature on how to divide the research done in software visualization into subfields. Thus I define here terms that I use in the following chapters. First of all, I define the software visualization and after that, two subdisciplines: algorithm visualization and program visualization. Figure 1 shows a Venn diagram of the relationships between these fields.

As defined by Price et al. (1993), *software visualization* combines all the visualizations that have been made to help or to teach the process of software engineering (e.g. design or debugging). Software visualization utilizes typography, graphics, animation, cinematography or any other visual means with the technology for human-computer interaction to ease human understanding, learning and effective use of computer software (Price et al., 1993). The definition given leaves the field very open and interdisciplinary. Thus it is clear that subfields have been formed. Here I only define two relevant subdisciplines but others also exist.

An algorithm is an abstract conceptualization of a program or a part of it. Thus *algorithm visualization* (or *animation*) stands for a high-level characterization of a piece of software with a visualization (Price et al., 1993). These visual descriptions are not related to the source code of any program, but are on a higher level where only the most relevant aspects of the program are shown and the details are left out. This can be achieved, for instance, by using systems that allow the user to manually construct either a *static* visualization or a *dynamic* animation from the ready-made visual objects. Moreover, there are also a few automated systems that use the source code or the pseudo-code of the program to form the visualization of the underlying algorithm. However, the difference between the algorithm visualization and the program visualization can be very small and the fields even overlap.

Whereas algorithm visualization has an abstract view on the piece of software, the program visualization concentrates on the concrete aspects of the program, such as its code and data. Thus *program visualization* can be defined as a mapping, or a transformation, of a program to its graphical representation (Roman and Cox, 1993). This means that in program visualization, the software is defined in textual form (e.g. source code or machine code), and graphics are used to illustrate some aspects of the program or its execution (Myers, 1986). These representations can be either *static*, for example snap shots of a data structure's content during the execution, or *dynamic*, highlighting of the source code. It can also be related either to the *code* or the *data* of the program as in the previous examples.

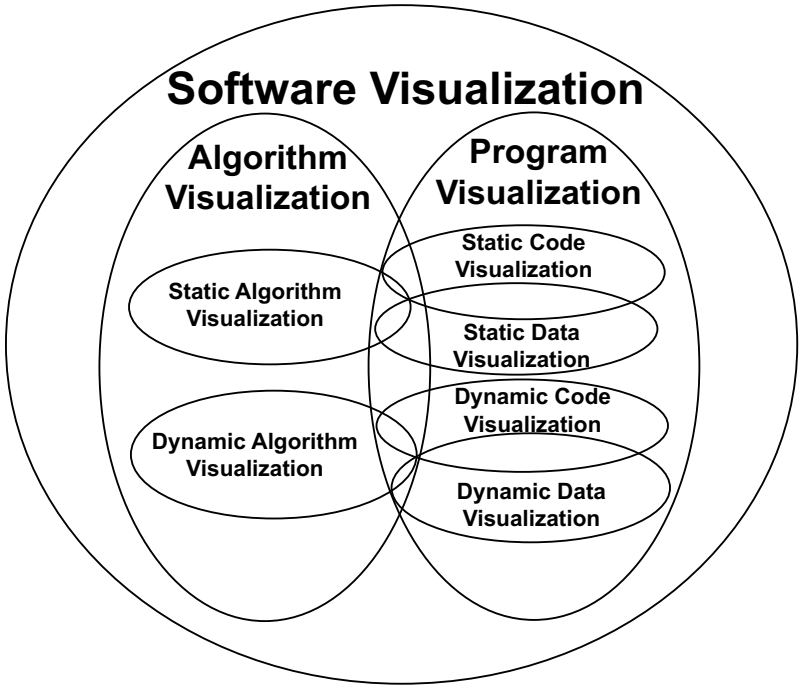

Figure 1: Venn diagram for the terms defined in this section. The size of each area is not relevant but the intersections between the areas are (Price et al., 1993).

In this thesis, the members of the Jeliot family are regarded as systems that have properties of both sets, algorithm and program visualization. This depends on how the systems are used. Eliot and Jeliot I are more related to algorithm visualization, whereas Jeliot 2000 and Jeliot 3 are more related to program visualization.

2.2 Related Work

In this section, I review work that is related to Jeliot 3. My aim is not to give a thorough analysis of all the research related to my topic, but rather to introduce a few interesting systems or ideas that have influenced the development of Jeliot 3. Other research related to my topic is introduced separately in each chapter the research is connected to.

An idea of using an XML-based language to connect a debugger and a visualization engine together was introduced by Stratton (2001, 2003). The language would describe the program's source code as well as the data flow, so that the program could be debugged and visualized without knowing what the debugger does. The XML-language makes it possible to some extent to form a programming-language-independent program visualization tool. The proposed language is explained in Section 6.1.

The research also proposes a model for new program visualization systems. The model is introduced in the Section 4.1. Both of these propositions affected the design and development of Jeliot 3 by introducing the idea of separation between the visualization target and the engine. For our purpose, the proposition to use XML-based language was too complex to implement, because of the requirements of XML for well-formedness and validity.

Another appealing idea came from Ben-Ari et al. (2002b). They propose to use a separate system for visualization of object-oriented programs. In this system, the key features are the static visualization of class structure and the dynamic visualization of object creations and object method calls. In this way, the students are introduced to object-oriented programming concepts in the first few weeks of the course. In Jeliot 3's object-oriented features we try to do the same, but we do not separate the visualizations into a different tool or view; instead, they go in hand with the visualizations of other programming concepts. Moreover, another important issue is discussed in this paper concerning the views of the program. As stated by Petre (1995), users on different levels need different kinds of views of the same matter. This led us to think about how we could form different views without developing a whole new tool. This is an important point that should be further developed.

One of the first and most renowned systems developed to teach introductory object oriented programming is BlueJ (Kölling et al., 2003; BlueJ, 2003). The key feature of

the system is the static visualization of the class structure as a UML diagram. Furthermore, it allows the learner to interact with the objects by creating them, calling their methods and inspecting their state with easy to use menus and dialogs. However, it does not provide any dynamic visualizations of the program, which is the purpose of our system. It would also be possible to introduce a similar kind of view into Jeliot 3.

There are two versions of the same system, namely Javavis (Oechsle and Schmitt, 2002) and Jvisual (Birkheim, 2002), developed on top of the Java Debugging Interface (JDI) to obtain information about the runtime behavior of the program. They visualize the state of the program and its changes during execution. These systems are not meant for real novices, because the visualization they produce expects that students are familiar with UML and the basics of programming. However, this kind of system could be very useful for advanced courses in programming.

There are also several pedagogical environments that help the novice students to overcome problems of compiling and debugging the software. DrJava (Allen et al., 2002; Stoler, 2002) is one of them. It is not a software visualization tool, however, it is mentioned here due to the usage of DynamicJava, a Java source interpreter. It is elegantly employing the features of DynamicJava to help students to interact with self-written classes by creating objects and executing separate statements on them easily. The read-evaluate-print loop is introduced into Java teaching, which means that students do not have to write complete programs before they can test the programs but they can evaluate each line of code separately. Moreover, because of the integrated interpreter, students do not have to worry about compiling the program with standard Java compiler that could introduce different kinds of problems. This leaves students more time to struggle with the difficulties of programming. This kind of scripting would also be possible with slight modifications in Jeliot 3. Nevertheless, we want to emphasize the writing of programs instead of partial scripts and it also makes more sense to visualize a whole program than partial scripts. In this way the transition from Jeliot 3 to using standard Java compiler will be also smaller.

3 History of the Jeliot family

The development of the Jeliot family began more than ten years ago and different kinds of research projects have been developing the concept further. There are three previous systems in the Jeliot family: Eliot, Jeliot I and Jeliot 2000. This chapter narrates the history of the Jeliot family, introduces the structures of the different systems and explains the empirical research done with them. The section concludes with the implications from the previous research and design of Jeliot family to the design of Jeliot 3. For an overview of the Jeliot family and the empirical evaluations see the article by Ben-Ari et al. (2002a). However, that article does not contain detailed information about the functional structures of the systems that is found in this chapter.

3.1 Eliot

The development of the first version of Jeliot family began when Erkki Sutinen and Jorma Tarhio were producing algorithms' animations to teach string matching (Sutinen and Tarhio, 1993). They found that it took almost 100 hours to create a single animation with the tools available. To ease up the development of the visualizations, they decided to build an animation library that could be used to animate programs written in C-language. In this process, a library of *self-animating data types* was created. A data type is self-animating when it visualizes its run-time behavior without extra code or annotations added to the program. The animation of the data type is then a natural result of the usage of the data type. Moreover, the appearance of the data type's visualization is modifiable both before and, in some cases, during the animation. This is close to the *interesting events* approach in which each of the events is annotated in the code (Demetrescu et al., 2002). With self-animating data types, no annotation is needed and the operations of the data types are always considered as interesting events.

Eliot (Sutinen et al., 1997; Lahtinen et al., 1998), the first version of Jeliot family was a user interface to write programs, use the library of the self-animating data types and modify their appearance through dialogs making the animation generation *semi-automatic*. In semi-automatic program visualization most of the work is done by visualization system, but the user can fine tune the visualization to better suit her mental model (Lahtinen et al., 1998).

3.1.1 Theater Metaphor

The design of Eliot uses an important metaphor, the *theater metaphor*. It was introduced first in this form in Halsa++ (Lahtinen et al., 1994) and refined for Eliot (Sutinen et al., 1997). This metaphor has been used in all the implementations of Jeliot in different forms. When introducing each of the systems, I will explain the differences between the theater metaphor in Eliot and in the new version. In this introduction, I have used the definitions from Sutinen et al. (1997).

In the theater metaphor, the program code is considered to be the *script of the play* that consists of several *roles*. The role in this sense means the data object in the visualized algorithm, for example a variable, an array or a data structure. Each role knows how the operations of that role are performed. It is also possible to leave any of the variables or other roles out from the visualization as some of the roles can be left out from a play.

For each selected role, there should always be one *actor* on the *stage*. In this context, actors are graphical objects which have attributes whose values affect their appearance and performance, just as each actor in a theater has his or her personal qualities that affect the performance. A case in point is a situation where integer variable x can be visualized with either a rectangular box that represents the value as a scaling rectangular or a circle with a number inside. Moreover, the box or circle can be placed in any position on the screen and its color and size can be modified. Several actors can be associated to one role on separate stages. A stage refers to a window on which the animation is performed.

The performance of the actors is guided by a *director*. In Eliot, the director has a twofold role. On the one hand, the user is thought to be the director when she defines the attributes of an actor and so affects the performance of the actor. On the other hand, there is a class that controls the performance of all the actors that can be thought of as a director of the play. This permits the possibility for many simultaneous productions of the play by different directors and actors on *multiple stages*, as one algorithm can be visualized on many windows in various ways at the same time.

3.1.2 User Interface and Visualization

The user interface in Eliot consist of multiple windows. It can be divided into two separate parts, namely *code and appearance editor view* (Figure 2), that is actually the user interface of Eliot, and *animation view* (Figure 3), that is shown when the compiled animated algorithm is run.

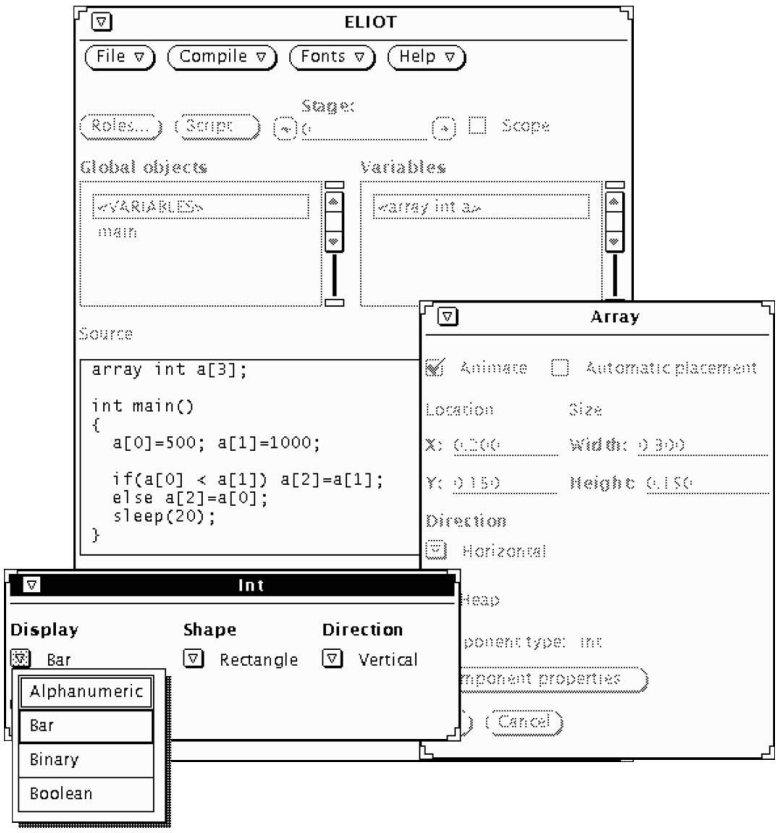

Figure 2: The user interface of Eliot's code and appearance editors (Sutinen et al., 1997).

Figure 2 shows three windows for editing the code and the appearance of the variables. There is a menu bar with four menus in the main window titled "ELIOT". Below that, there is the selector for variables, that are to be shown in the animation. The source code editor is at the bottom of the main window. There are also two other windows open. One entitled "Array" for an array variable and its appearance parameters, and another entitled "Int" for the array's component type that is integer.

The animation windows that show the actual visualization of the code are illustrated in Figure 3. In those animation windows, the array variable of the source code is animated

in two different ways. In the first stage entitled “ELIOT: Stage #0”, the array is shown as scaled vertical bars. There is also the control panel for the animation in the upper left corner of the stage in a separate window entitled “Polka Control Panel”. In the second stage entitled “Eliot: Stage #1”, the array is animated as a vertical column of bounding boxes with a number in the center. There is another control panel at the bottom of both windows for scrolling the window, zooming in and out and refreshing the display.

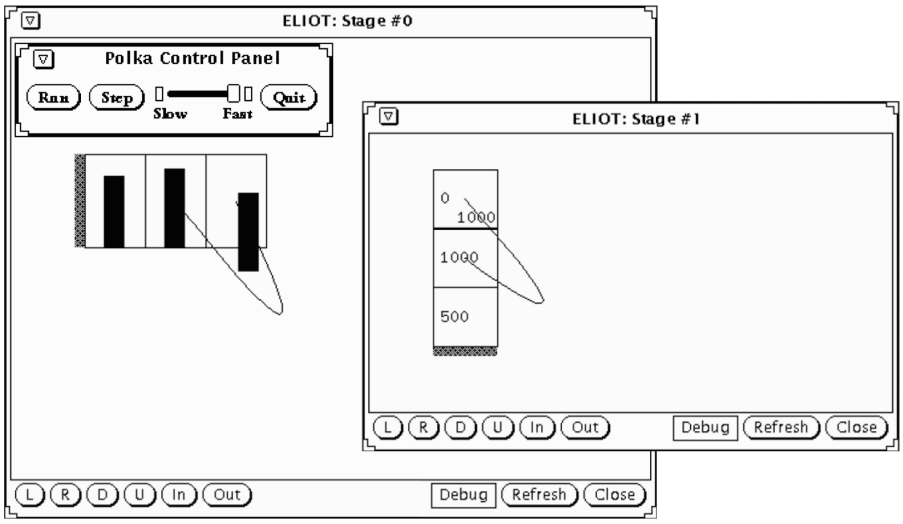

Figure 3: The user interface of the animation view with two stages (Sutinen et al., 1997).

Eliot can visualize most of the C-language’s data types (e.g. integers, floats and characters). Moreover, it can also visualize one and two dimensional arrays and tree structures. These visualization can take various forms. Visualization is concentrated only on the data types and their operations, mainly on the comparison operations. The control flow is not visualized, but after a comparison, it is stated whether or not the comparison was successful. The code is not visualized at all. The active code line is not highlighted and the code is not even shown if the animation is not run through the Eliot environment, as it is possible to run the animations separately from the executable files.

3.1.3 Design and Implementation

Eliot runs in X window system and is implemented in C++. The self-animation of the data types is based on overloading of operators in C++ and is implemented by using

Polka algorithm animation library (Stasko and Kraemer, 1993). Eliot consists of three separate packages, namely the user interface, the Eliot-C compiler and the class library containing the self-animating data types (Figure 4). The separation seems to be quite strict and well-defined, making the interfaces simple.

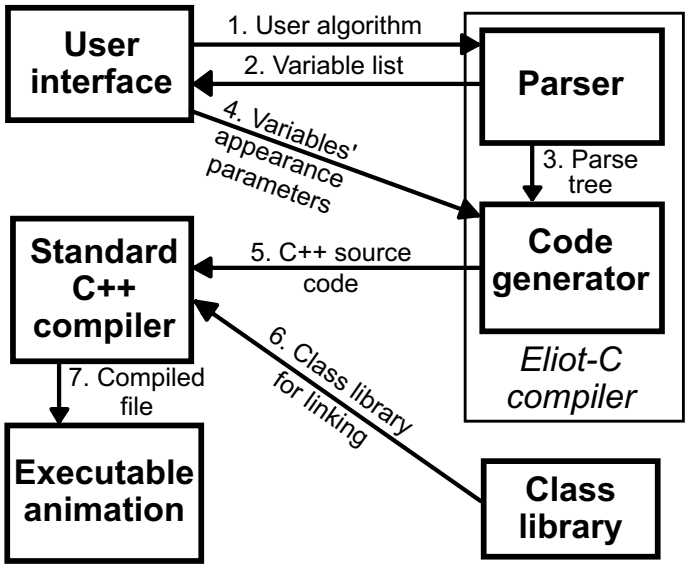

Figure 4: The functional structure of Eliot (Lahtinen et al., 1998).

Figure 4 illustrates the structure of Eliot system, showing also the process of normal usage. The user interacts with the system and writes an algorithm to be visualized. When the algorithm written in Eliot-C language, dialect of C-language, is finished, it is sent to be compiled into an executable animation program (1). First, it is processed by the Eliot-C parser, which consists of both a lexical analyzer and a parser. They were made with the Unix tools Lex and Yacc that are automatic lexical analyzer and parser generation tools respectively. During parsing, a list of the variables is extracted and shown to the user, and she can decide the appearance of the variables (2). The code generator produces C++ source code according to the parse tree and the appearance parameters (3 and 4). In this stage, the variables selected to be visualized are changed to role instances. The resulting source code is sent to the standard C++ compiler (5). The source code is compiled and the class library is used for the self-animation data types and Polka animation system (6 and 7). After this procedure a ready-to-run animation is formed and can be run with or without the system.

Figures 5 and 6 illustrate how the visualization is implemented in the Eliot in two different situations, namely in variable declaration and in assignment. The cases are

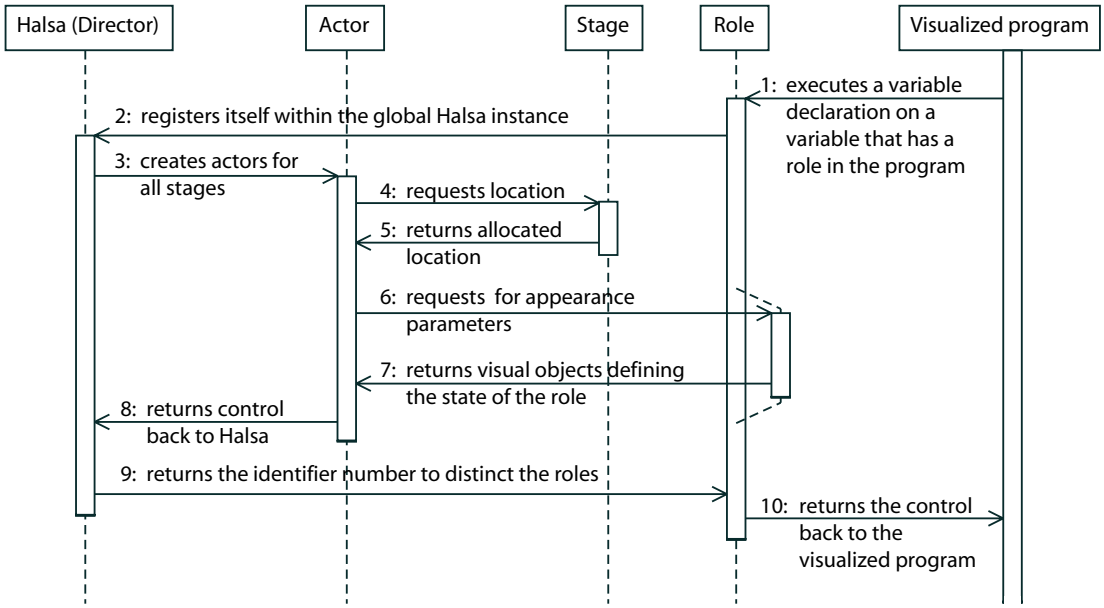

Figure 5: The sequence diagram of Eliot's animation when new variable that has a role in the visualization is declared in the user program (Sutinen et al., 1997).

shown as sequence diagrams. In Eliot, director class is called Halsa which came from the previous system made for visualization of string algorithms. The other classes are taken directly from the theater metaphor.

In Figure 5, the procedure of variable declaration is explained. In the program, each operation that should be visualized calls the corresponding role class (1). This is done through operator overloading. The role informs the director (Halsa) (2). Halsa object creates the proper actor on the stage, which allocates the actor at the correct place (3, 4 and 5). The role instance gives the corresponding appearance parameters to the actor (6 and 7). Then, the actor is ready to appear on the stage and the user program's execution can continue (8, 9 and 10).

The process of visualizing an assignment is illustrated in the Figure 6. The procedure is very similar to the variable declarations process. The only difference is that the animation is shown before appearance parameters are given, so that the assigned value changes to the actors type of visualization after the assignment (4 and 5).

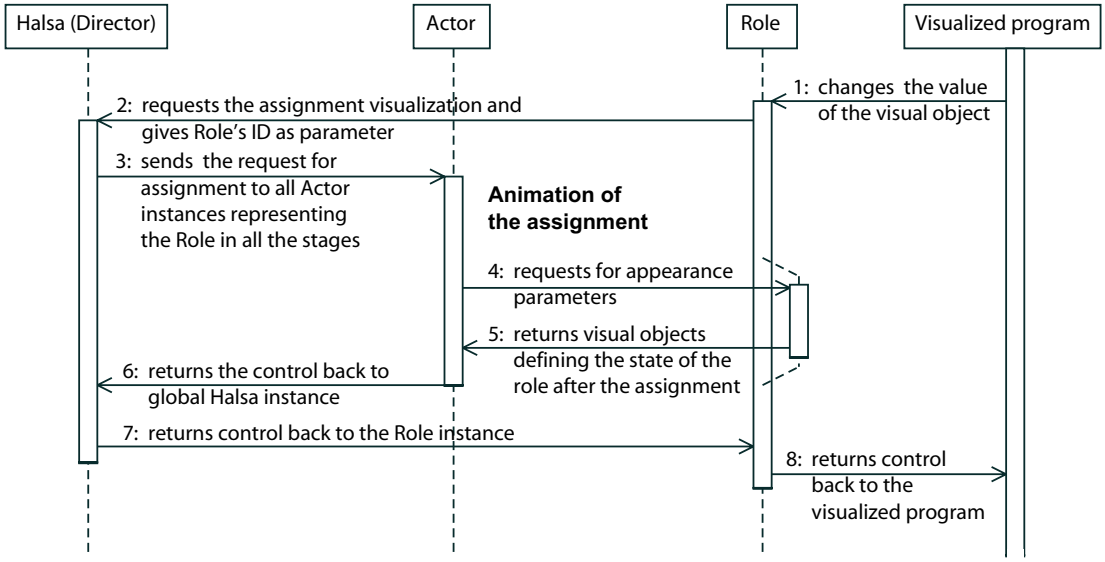

Figure 6: The sequence diagram of Eliot's animation when a value is assigned to a variable with a role (Sutinen et al., 1997).

3.1.4 Empirical Evaluation

An empirical evaluation was carried out in a data structures laboratory course with 8 students of which 6 used Eliot (Sutinen et al., 1997; Markkanen et al., 1998). In the course the students implemented different algorithm and data structures, studied their complexities and prepared posters. The evaluation collected only qualitative data. The data was collected in various forms and it consisted of questionnaires, video taping and study diaries, etc.

The results of this study reported by Sutinen et al. (1997) showed that the students were not generally motivated for the course in the beginning, but they became motivated during the course. Nevertheless, a precise reason for the motivation boost could not be found, because in addition to using Eliot the students were also exposed to a different kind of teaching style than they were used to. These reasons together could have had a positive effect on the motivation. It was also claimed that students obtained a deeper understanding of the algorithms they were studying.

Markkanen et al. (1998) carried out an analysis of the code and documentation produced by the students in the same course. The analysis showed that the quality of the code and the documentation were better when students used Eliot during the development process.

3.2 Jeliot I

The evaluation of Eliot's use in classroom settings showed that this kind of a tool for visualization can help students in various ways, especially in motivation. However, Eliot worked only in the X Windows environment and to port the software to other environments was laborious. This led to the development of a new version, Jeliot I (Java-Eliot) (Sutinen et al., 2003; Haajanen et al., 1997). Jeliot I can be used on the Internet making the usage of it location independent. Although the design of Jeliot I is different from the design of Eliot, the same concepts of self-animating data types and semi-automatic animation are used. It is a general purpose visualizer accepting many kinds of programs written in Java, but only visualizing the supported aspects of the programs; it concentrates mainly on the data flow of programs. Jeliot I can also be used in other contexts. Meisalo et al. (1998), for example, used it to teach Mendelian inheritance in Biology. Jeliot I is still in use having a few hundred hits every month (see AAPS-project, 1997).

3.2.1 Theater Metaphor

Jeliot I was built on the same concepts as Eliot. The theater metaphor was also used in a quite similar way. However, one interesting change was made; in *improvisation*, the user can modify the appearance of the actors and the changes are shown immediately on the stage just as actors can improvise in theater. Moreover, few new members were introduced to the theater metaphor. The stages are administered by *stage managers* and the director has its own helper called *director's assistant*. Stage manager handles the allocation of actors on the stage and also keeps tracks of the roles that are currently played on the stage. Director's assistant helps the coordination and synchronization of the play by collecting all the messages from the stage managers and informs the director only after messages from all the stages have arrived.

3.2.2 User Interface and Visualization

In Jeliot I, the code editing and visualization are separated. The code editor view is shown in Figure 7. The code editing possibilities are limited. User can only clear everything, use the default template and load some examples. The code cannot be saved

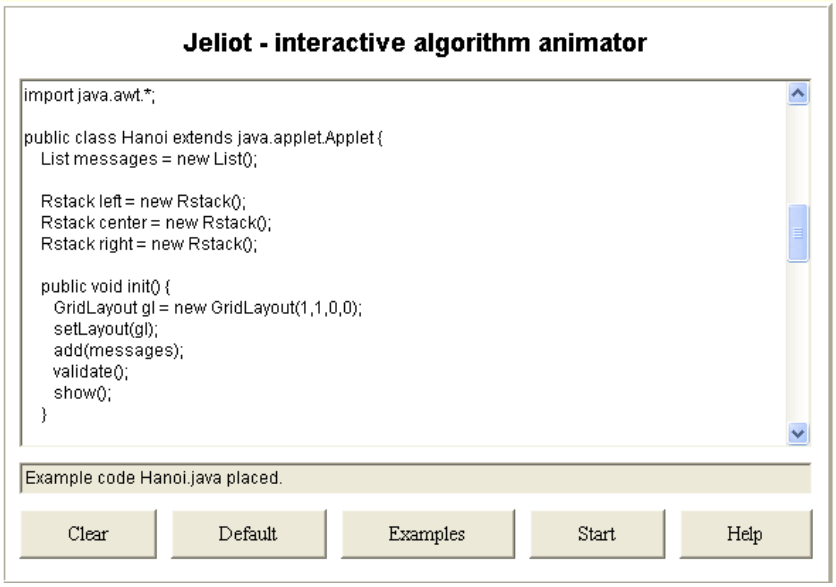

Figure 7: The editor window of Jeliot I.

because Java applets do not have any access to the hard drive of the user's computer. The user can also open a browser window for accessing the help files. After finishing the coding, the user can push start button to begin the visualization of the code.

If any errors occur during the compilation, new browser window is opened and the error message and possible cause are shown in it. Otherwise, three windows are opened to control the reparation of the visualization and its playing. Three windows, namely "Director", "Stage Manager 1" and "Stage 1", are shown in Figure 8.

On the left side of Figure 8 is the director window, the main control panel of the visualization. The user can play, stop, rewind, play step-by-step and change the speed of the animation through it. Furthermore, the user can set the properties of the actors on the stages or open new stages. The source code is also shown on the bottom of the window.

The stage window is shown in the lower right corner in Figure 8. It shows the actual animation. There may appear several stages at once and the animations in them are shown simultaneously. The stage window also contains buttons to set up the stage, to see the help files and to close the stage. When the user pushes the "setup" button the manager of the stage is shown. When the stage is initialized for the first time, the stage manager is automatically shown.

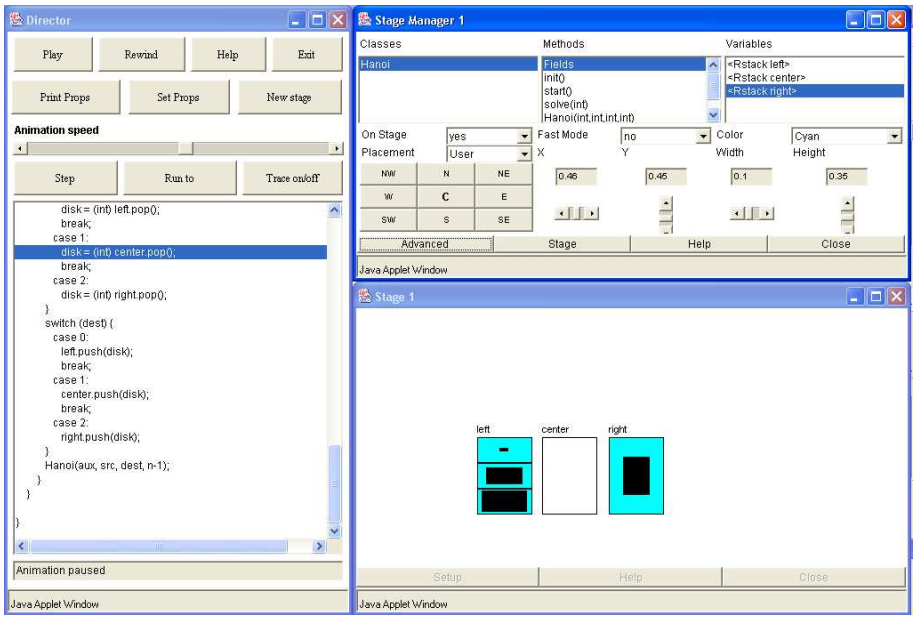

Figure 8: The user interface for the animation system of Jeliot I.

The stage manager window is used to select the visualized variables and to control their visual appearance. It is shown in the upper right corner in Figure 8. The user can select the visualized variables from each class and method with three selectors in the upper part of the window. The lower part of the window contains widgets to locate each variable on the stage. The user can also change the appearance of the actor of each variable by pushing the “Advanced” button. This opens a new window that contains several widgets to set the appearance parameter, for example, size, shape and color.

Jeliot I visualizes most of the primitive data types of Java. It can also visualize one and two dimensional arrays and queue and stack data structures. They can have several different kinds of visualizations. New visualized data types can be added by coding the corresponding visualization of data type and its operations. The visualization of the program is concentrated only on the data types and their operations, mainly on the comparison operations. The control flow is not visualized, but after a comparison, it is stated whether or not the comparison was successful. The code is visualized in the director window by highlighting the whole line of code at the current location of the execution. Visualization can be shown in several windows and different variables can be visualized in each of the stage. Furthermore, the appearance of the actors can be changed at the any stage of the animation, and the modification will be shown immediately.

3.2.3 Design and Implementation

Jeliot I was developed in Java. It is a client-server application. It runs as a client applet on the web page visualizing Java programs and as an application on the server. This means that the arduous work is done on the server and the client is just a user interface to write the source code of the program and to view and to control the animation.

The design of Eliot was adapted to the client-server environment in Jeliot I. Figure 9 illustrates the functional structure of Jeliot I. Jeliot I consists of five different components, namely *main window*, *server*, *EJava compiler*, *class library* and *animator*.

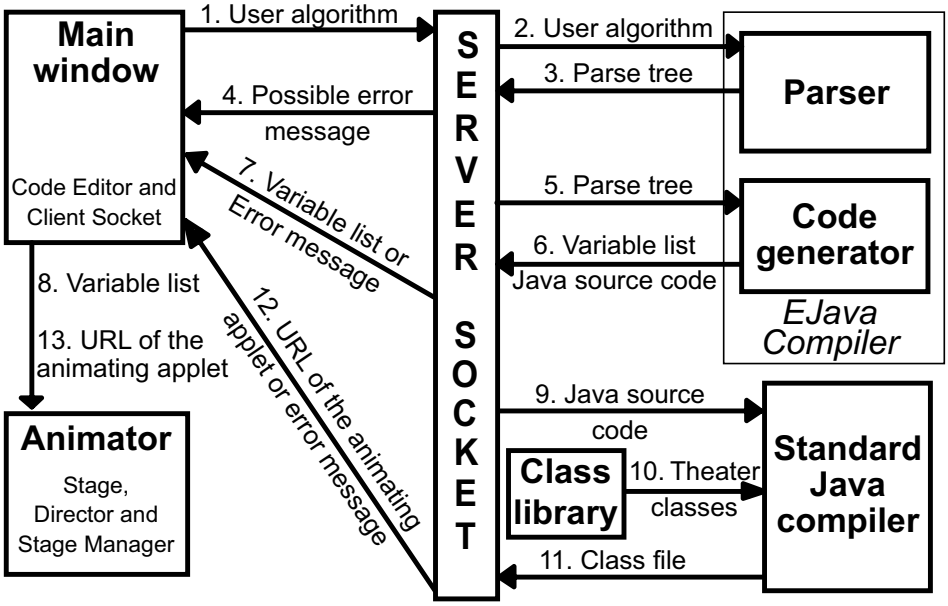

Figure 9: The functional structure of Jeliot I (Sutinen et al., 2003).

When the Jeliot I applet is loaded on the web page, the user can interact with the code editor in the main window and either write an own source code with EJava language or load an ready made source code. Jeliot I uses a dialect of Java called *EJava*. The differences between EJava and Java are small. For instance, arrays must always be declared using notation `int [] array;` and notation `int array[];` can not be used. When the animation process is started the code is sent to the server to be compiled to the corresponding visualization code (1).

EJava compiler consist of a lexical analyzer, a parser and a code generator. The lexical analyzer and the parser were made automatically with JLex (Berk and Ananian, 2003) and CUP (Hudson et al., 1999) respectively (2). The parser returns the resulting parse

tree or an error message to the server if any errors occurred (3 and possibly 4). The parse tree is then given to the code generator (5). During the compilation, the code generator changes all the visualized types in the source code to their corresponding self-animating data types. For example, variable declaration `int x;` is changed to `Rint x;` and an assignment statement `x = 3;` is changed to `x.assign(3);`. This is done in order to generate the corresponding visualization of the variable and its operations. The code generator returns a list of the variables that can be visualized and the modified source code of the program or an error message if an error occurred (6, 7, 8). After that the modified source code is compiled into byte code by standard Java compiler (9). The class libraries of the role classes (e.g. `Rint` class) and the classes helping the animation generation are used in the compilation (10). Finally, the visualization is sent to the client-side and the animator shows the visualization interface where the user can select the variables, changes their appearance and then play the animation (11, 12 and 13).

The process of the visualization is introduced in Figures 10 and 11 as sequence diagrams. The diagrams show the different stages of the visualization in Jeliot I in connection with the different classes used.

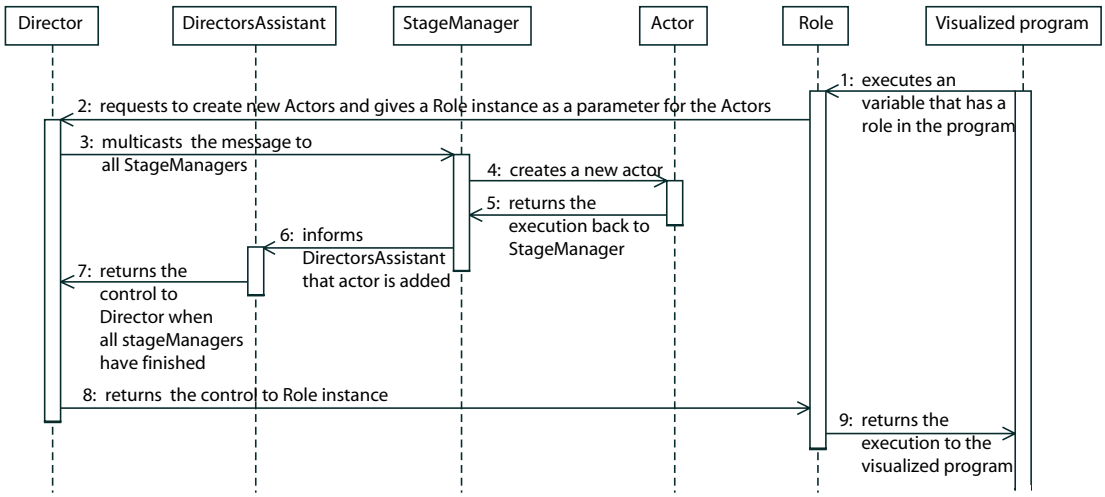

Figure 10: The sequence diagram of Jeliot I's animation when a new variable that has a role in the visualization is declared in the user program (AAPS-project, 1997).

In Figure 10 the sequence diagram of the animation process of a variable declaration is shown. The procedure is similar to Eliot's corresponding process. When the visualization is run and a variable with a role (e.g. `Rint x;`) is declared, the role requests the director to create new corresponding actors to the stages. The message is multicasted

to all stage managers who create a new actor with the user-defined visual appearance if the role is animated on the stage. When all stage managers have finished the process the control will be returned to the executed program according to the diagram.

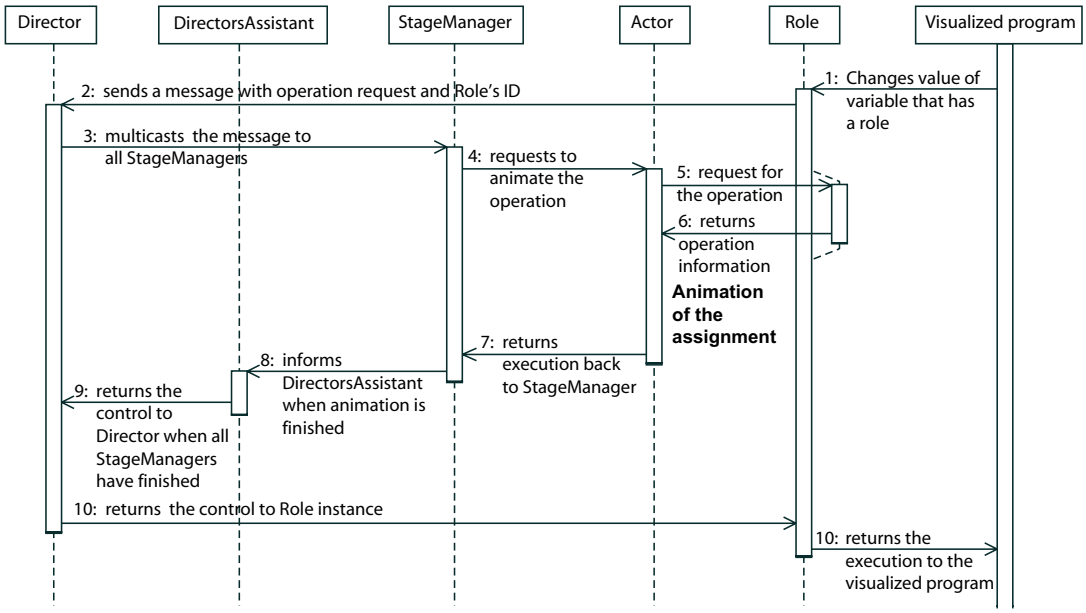

Figure 11: The sequence diagram of Jeliot I's animation when a value is assigned to a variable with a role (AAPS-project, 1997).

The sequence diagram of the procedure to animate an assignment statement is illustrated in Figure 11. When an assignment statement (e.g. `x.assign(3);`) is executed in the visualized program, the role class's `assign` -method sends the director a request to visualize the assignment. The director multicasts the message to all stage managers. If the corresponding actor is found from the stage, the stage manager asks it to visualize the operation. The actor instance uses the reference to the role class to find out which operation is to be visualized and shows the animation. The actor instance uses a graphical primitives library Jeliot Animation API (JAPI) that replaced POLKA animation library of Eliot because there was no Java implementation of it. When the animation is finished, the execution control is returned to the visualized program according to the diagram.

3.2.4 Empirical Evaluation

Jeliot I was evaluated in different kinds of situations (Lattu et al., 2003, 2000). The data collected in these experiments was mainly qualitative. In the first study, Jeliot I was evaluated in the contexts of university and high school programming courses. Lattu et al. (2000) analyzed the different utilizations of Jeliot I in lecture, classroom and distance learning courses. The strengths and weaknesses of its usage were considered from the perspective of the students and educators. A number of students were interviewed on the courses and some lectures and assignment sessions were observed. The university students did not use Jeliot I frequently but only a few times during the whole course. Thus, the evaluation mainly consists of the observations done in the high school course.

The results of the evaluation can be summarized by explaining the different utilization strategies, eases and difficulties encountered when Jeliot I is used as well as by pointing out directions for further development of Jeliot. There were five different utilizations of Jeliot identified: *to support the presentation of new concepts*, *to explain an example program to students*, *to tutor the students during independent work*, *to do programming exercises* and *to be used independently outside classroom*. One interesting usage pattern was found in all of these categories; the animation was primarily played in the continuous mode even though the step-by-step mode seems more intuitive as it gives more time to the user to process the graphics display and control the animation. Teachers used this playing mode as well and to gain enough time for explanation they slowed down the speed of the animation. This is understandable because then they would not need to handle the animation control but they could concentrate on the explanation. None of the features in the system (e.g. playing modes or visualization) could be identified as very helpful for learning, and only positive overall comments of the usability, motivation and suitability for novices were received in interviews.

The main aim of the study was to identify the problems of Jeliot I to aid the further development. The problems could be classified into four different categories: *problems in programming*, *user interface issues*, *visualization problems* and *computer related problems*. In programming, the problems were related to insufficient error messages and the differences between Java and EJava. User interface was found too complex and hard to use by novices. The user interface should contain less windows and common icons should be used to ease the usage. The common utilization patterns should be

analyzed further to make those usage sequences easier to perform. The visualization itself introduced one crucial difficulty; the granularity of the visualization was too coarse for novices and they would have needed much finer visualization, for example, for loop or conditional statements. This indicates the need of the programs' control flow visualization.

In the second study, the visualization of programs in general was studied in the assignment sessions of an introductory programming course at university (Lattu et al., 2003). This study also included findings about the use of Jeliot I in the assignment sessions. Nevertheless, the primary findings are connected to the general use of visualizations when a student explains a program to other students. In the study, the researcher observed about 60 hours of assignment sessions and collected notes about the usages of the program explanation strategies. The study was explorative in nature and the collected data was analyzed mainly qualitatively, but some of the findings were validated with quantitative methods.

In the study, it was found that the visualization of programs was an important part of the communication in the assignment sessions. Some kinds of visualizations were used on average more than four times in each assignment session. The explaining strategies and visualization methods of the programs varied greatly. This means that demands for a general purpose visualization tool are high, especially in a sense of flexibility.

Students visualized the program code, program structures (e.g. methods), variables, objects and object structures. During the spontaneous visualizations the blackboard and overhead projector were used because of their availability, flexibility and ease of use. Technical problems and complex user interface prevented novices from using Jeliot I or any other computer based visualization tool. The speed of the animation generation was also found very important.

When considering the code visualization, it should be possible to browse the code view up and down but moving between each method and class should also be easy. Moreover, the program code should be visualized with indentations, colored texts and other text style effects. Even writing and drawing remarks on the source code should be possible.

When visualizing variables and data structures, there should be as many different kinds of visualizations as possible to suit the different preferences of the students. This was

also found by Ford (1993) when researching the visualizations made by programmers. The classes and objects should be visualized either in the separate view of their own or in the same view with variables. The relationships between classes and between objects should be visualized.

Both of these evaluations gave new directions to the further development and the propositions of the authors are summarized below in the list with explanations.

- *Language support* was a problem and even though the differences between Java and EJava were small, they introduced considerably much trouble. Furthermore, the absence of input and output statements restricted the university students from using Jeliot I. This indicates that the rewriting of the programs for visualization tools should not be needed.
- *Error messages* were not informative enough in the current version as especially novices need support for syntactic mistakes.
- *Syntax highlighting* could help novices when learning the language. This could reduce the number of syntactic mistakes during the coding of the program.
- Especially the teachers but also students requested more *control over the visualization*. Especially when preparing visualizations for the lectures, the fine tuning of the visualizations is important.
- Novices need *finer granularity of the visualization* making the visualization of the program more complete. Jeliot I concentrates mainly on data visualization even though the novice students also required visualizations of the control flow and even object structures.
- *Source code annotations* (e.g. tooltip comments) and *explanations during the visualization* would support the going through of the source code and promote the visualization of the programs.
- *Simpler user interface* was needed. As seen in Figures 7 and 8, the user interface of Jeliot I is too complex with several windows and a large number of widgets in all of them. This hindered especially the novices' usage of the tool.
- Students used to introduce the programs in many different ways. This indicates that *different explanation strategies should be supported* by visualization tools.

- *Tutoring examples* that introduce the different visualizations of the data structures to the students should be available. This could familiarize the user with different visualization and different data structures but it could also stimulate the problem solving process.

3.3 Jeliot 2000

From the research done with Jeliot I, it was concluded that the current version of Jeliot I needed to be improved. However, the needed modification to the user interface and to the visualization of the programs needed different kind of an approach to the whole design. Thus, a new version was developed from the beginning, which allowed radical changes to the design. A new version, Jeliot 2000 (Ben-Bassat Levy et al., 2003), was designed especially for novice students whot did not have any programming knowledge before-hand. It was based on the technical expertize and research results gained from Eliot and Jeliot I. Jeliot 2000 was designed and developed at Weizmann Institute of Science by a visiting student Pekka Uronen from University of Helsinki under the supervision of Mordechai Ben-Ari.

Jeliot 2000 is a stand-alone Java application, which has a simplified user interface with VCR-like buttons. The animation generation is made fully automatic and animation shows both control and data flow of the program (see Figure 12). The new design is based on the idea that the visualization of the program is actually a consequence or a side-effect of the interpretation of the program. This means that the program is no longer first compiled to a annotated source code and then to a program, but it is directly interpreted with a Java interpreter. This idea led into an innovative design that enables the visualization of the data and control flow of the program. However, Jeliot 2000 supports only a relatively small subset of Java language and does not support object-oriented programming.

3.3.1 Theater Metaphor

The self-animating data types were not enough to visualize the control flow of the programs because they only visualized the operations of the data type. This also led to the revision of the theater metaphor in Jeliot 2000.

Jeliot 2000 supports only a *single stage* where the actors perform. This means that the *roles are no longer needed* because performance of *each actor is already connected to a certain role*. For example, a variable actor always has a role of a certain variable. The utilization of actors was also elaborated so that they do not just represent values, variables or data structures but also program structures such as methods, method calls, operators, expressions or even explaining messages. All the actors have similar properties and actions, however, each actor type has also specialized features making the actor structure hierarchical.

Another important new idea was that some of the *actors can contain other actors*. This means that when an actor contains another actor or actors it takes care of the placement and painting of the actors it contains. For instance, when a binary expression is encountered during the execution of the program, a new expression actor is created and the operand values and operator actors are shown in the corresponding expression actor. This expression actor then places the other actors on the stage.

A special case of the actor container is the theater that is not an actor but can contain other actors. Actually, it is the lowest level of the actor containers. The stage manager of the previous versions is now a theater manager managing the whole theater. It allocates the space for the actors on the theater.

3.3.2 User Interface and Visualization

The user interface of Jeliot 2000 consists of a single window shown in Figure 12. The user can interact with the code editor on the left pane of the window and code the program with it. The code editor contains editing commands that can be found in the menu bar in the upper left corner of the window. The user can begin the visualization of the program by pushing the “Compile” button in the control panel in the lower left corner of the window. When the program is ready to be visualized, the curtains of the theater on the right pane of the window are opened. The user can control the visualization with the VCR-like buttons in the control panel. User can play, pause, play step-by-step or rewind the visualization of the program. If any input is requested from the user during the program execution a box with a text field is shown to the user in the theater. The output console in the lower right corner of the window collects all the output that is produced during the program execution.

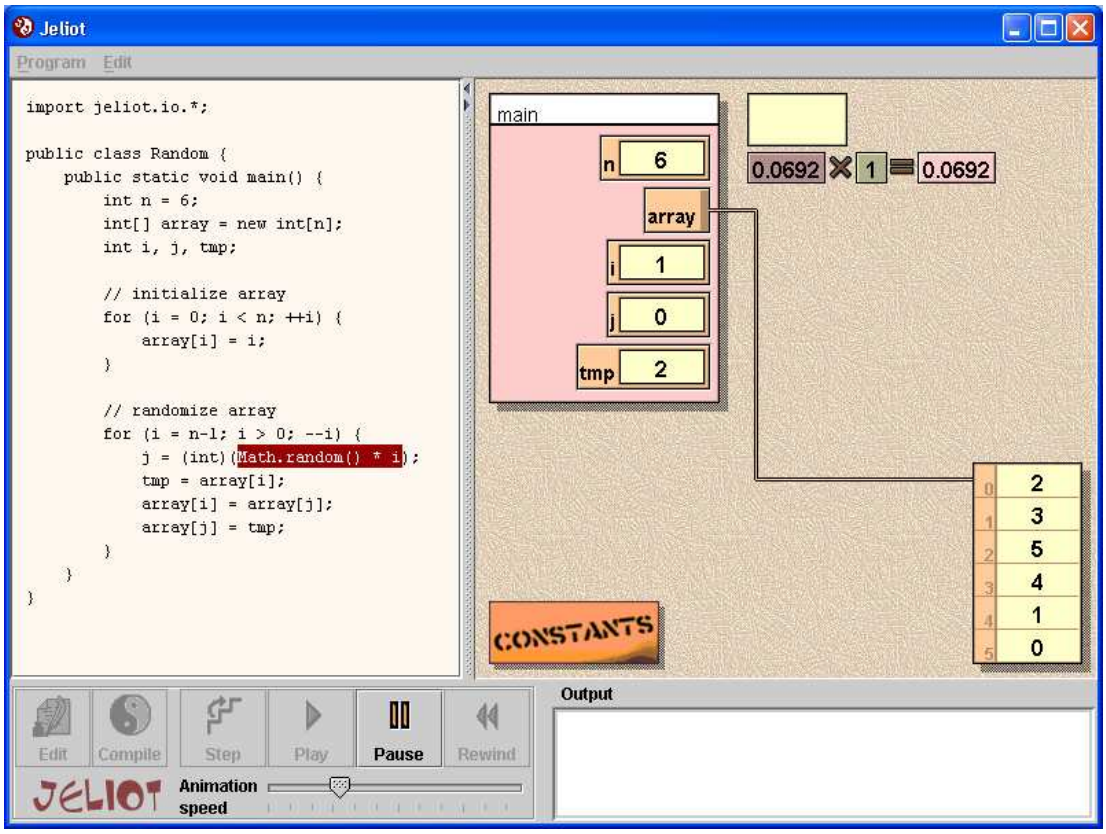

Figure 12: The user interface of Jeliot 2000.

The visualization is done on the level of programming language constructs (e.g. variables, values, methods and expressions). This means that each construct has their own visualization. New visualizations can be added by coding the corresponding visualization and its operations, but this can also require modifications to the interpreter. Thus, it is not easy and requires knowledge of the whole system. The input and output visualizations are done using this extendability. Each component has only one form of visualization and all the primitive values are visualized uniformly.

The visualization has two main principles in Jeliot 2000; the visualization is *continuous* and *complete* (Ben-Bassat Levy et al., 2003). Visualization is continuous in the sense that all the operations visualized must have relation to the previous operation. All these relations have to be visible so that the viewer does not need to guess what happened in the middle. For example, the visualization of the method call has to show the evaluation of the actual parameters and the method invocation and leave them visible until the method frame is shown and the values of the actual parameters are assigned to the formal parameters. The visualization of the program is complete when all the

features of the program are visualized. For instance, literal constants cannot appear from anywhere but they have to have a specified place to for appearance (e.g. constant box) and the meaning of the expression in the conditional statement have to be stated.

The code is visualized in code viewer in the left pane of the window. The code visualization is done in the level of statements and expressions. For example, if there is a assignment expression `x = x + 1;` and the right hand side expression is evaluated, only `x + 1` part of the expression is highlighted.

3.3.3 Design and Implementations

The design of the system is shown in Figure 13. The structure was simplified compared to the previous versions. The system consists of four different components: user interface, parser, interpreter and animation engine. The user interface was used to produce the appearance of the program. It laid out all the components on the window and handled the events of the components.

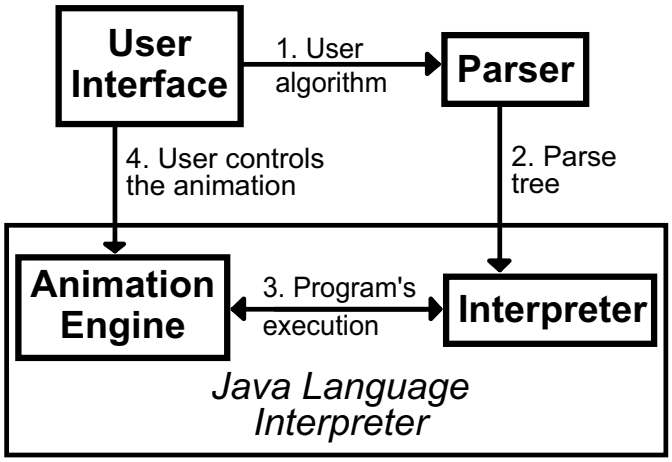

Figure 13: The functional structure of Jeliot 2000.

When the user decides to visualize an algorithm (1), the code is sent to a parser that first lexically analyzes and then parses the program into an abstract syntax tree according the rules of the Java language grammar (Gosling et al., 2000). The lexical analyzer was generated by JLex (Berk and Ananian, 2003) and the parser by CUP (Hudson et al., 1999). The possible error messages are shown by the user interface. When the user starts playing the animation, the parse tree is analyzed by the interpreter for the correct naming and typing (2). If there are no errors, the evaluation of the parse tree

is started. Every node of the parse tree is evaluated and each node is programmed to evaluate itself and produce a visualization of the evaluation at the same time (3). This way it is possible to visualize the program by interpreting it. This also means that if the visualizations are changed, it can imply changes to the interpreter. The user can control the visualization of the program with VCR-like buttons in the user interface (4).

With this kind of a design, a Java language interpreter was needed. However, there was not any stable open source Java language interpreters available at the time. Thus, there was a need to make one for Jeliot 2000. This led to a situation where a partial Java language interpreter was implemented. Thus the interpreter only understands a small subset of Java language and restricts the use of Jeliot 2000 into that subset of the language. This also means that extending Jeliot 2000 requires further development of the interpreter.

Figure 14 explains in the form of a sequence diagram what happens during the program execution when a variable declaration node is evaluated (e.g. `int x;`). First, one of the variable declarators is evaluated (1). In the example case, there would be only one declarator but if the statement was of the form `int x, y, z;` the variable declaration would contain three variable declarators. The variable declarator requests the director to highlight the corresponding code area (2). This can also result in pausing the visualization if the step-by-step playing mode is on.

When the animation continues (3), the variable declarator asks the director to create a new variable in the theater and gives the information of the variable as a parameter (4). The variable is created by the actor factory that handles the creation of all the actors (5, 6 and 7). In this way, the certain parameters like font of all the actors can be controlled in one class. When the variable actor is created (8), the current method stage is requested to locate this actor inside of it (9 and 10). The location is given to the variable that then produces the appearing animation for the variable declaration (11 and 12). The animation is shown with the help of the animation engine (13). During the animation, the visual appearance and possibly location of the variable actor is changed according the animation. After each change is made to the appearance the actors are painted again and when this procedure is done multiple times over time it produces an animation. When the animation is over (14), the variable actor is bound to the stage so that it will handle the placement and painting of the variable actor (15 and 16). Finally, the variable actor is returned to the variable declarator to be connected with

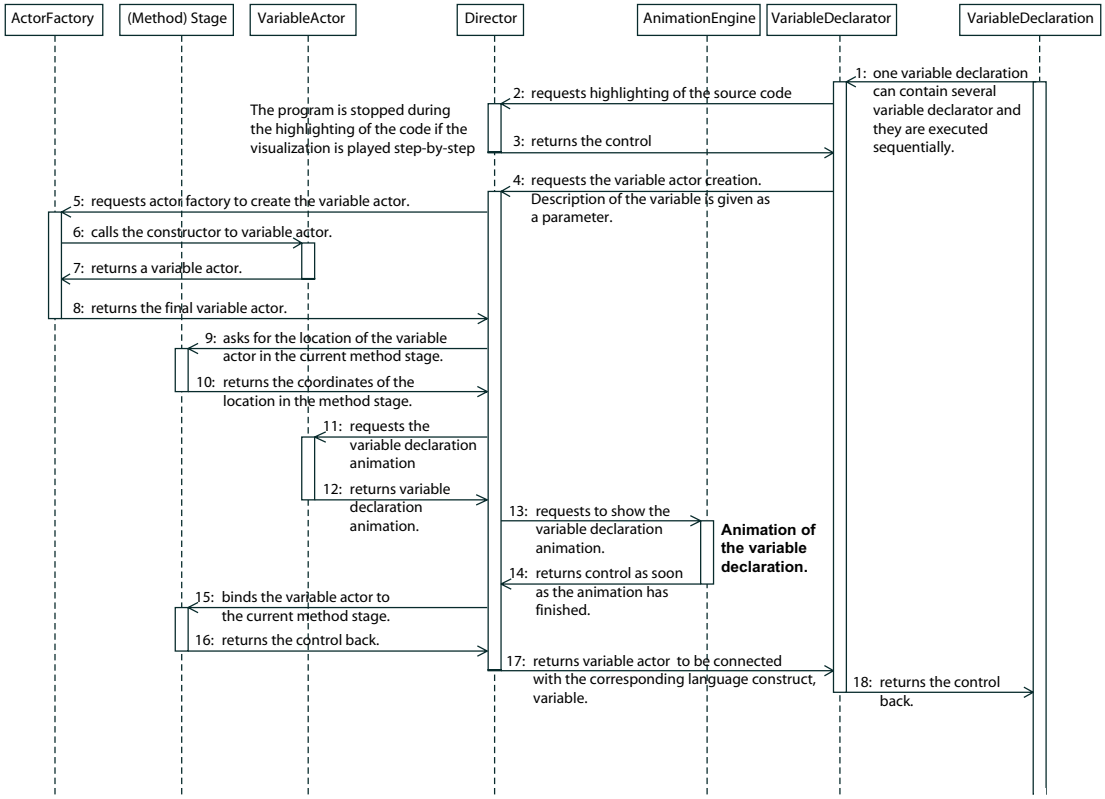

Figure 14: The sequence diagram of Jeliot 2000's animation when a new variable is declared.

the corresponding language construct, the variable (17). In this way, next time the same variable is accessed and the corresponding variable actor can be found. Then the interpretation of the program is continued (18).

Figure 15 illustrates the visualization procedure of the assignment expression. When an assignment expression is executed, the right hand side of the expression is evaluated first. It can be any expression suitable for assignment. For example, in the assignment expression `x = 1;` the right hand side value would be a literal value 1. The value actor is stored for the assignment. After that, the left hand side of the assignment is executed, meaning that the variable actor for the assigned is stored.

The actual assignment visualization process that is shown in Figure 15 begins after these preparations. The assignment expression requests the director to highlight the corresponding code area (1). This can also result in pausing the visualization if the step-by-step playing mode is on. When the animation process continues (2), the assignment visualization is requested from the director (3). The variable and value actors

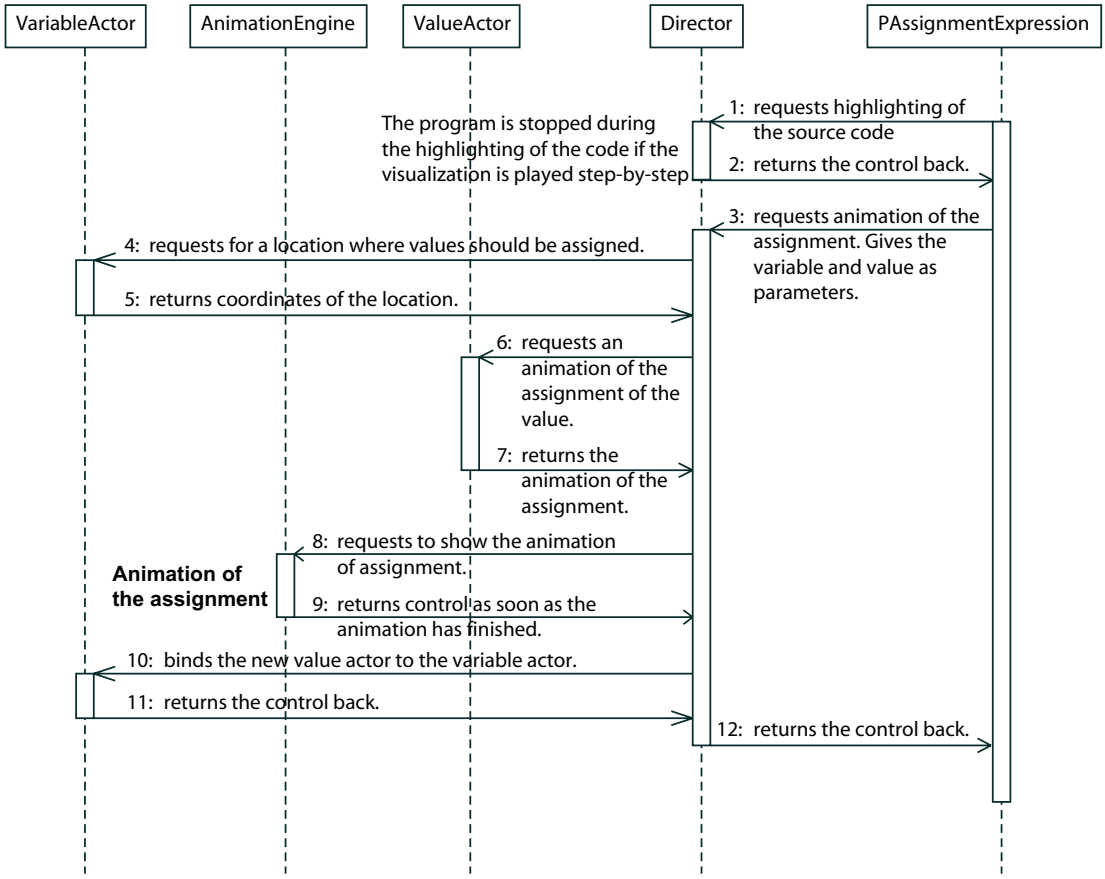

Figure 15: The sequence diagram of Jeliot 2000's animation when a value is assigned to a variable.

are given as parameters so that the right value actor is assigned into the right variable. The director requests the location for the value in the variable actor (4 and 5). With the received location, the value actor can generate the animation that makes the value actor fly from the current location to the given location in the variable actor (6 and 7). Then the animation is shown as explained in the previous case (8 and 9). Finally, the value actor is bound to the the variable actor and control is returned back to the assignment expression (10, 11 and 12)

3.3.4 Empirical Evaluation

Jeliot 2000 was evaluated in classroom settings with 10th grade high school students who were taking their first course in algorithms and programming (Ben-Bassat Levy et al., 2003). Students were actually learning Pascal but a preliminary test showed

that the use of Java syntax was not a problem for learning even though students were learning programming with Pascal. Two classes were observed. Both classes received normal instruction, but in addition to this, the treatment group had one hour of instruction using Jeliot 2000, whereas the control group had one hour of instruction without Jeliot 2000. The control group had better grades from the previous courses of Mathematics and turned out to be consistently better than the other class in pretests and post-tests.

The students were introduced to the basics of programming (e.g. input and output, assignments, conditional statements and loops). Before and after each of these subjects were, taught students were given a test and the results were analyzed both quantitatively and qualitatively. In addition to this, qualitative data was collected by interviewing the students and recording the verbalizations of their problem solving procedures.

Although the control group was consistently better than treatment group using Jeliot 2000, the improvement of the treatment group using Jeliot 2000 was relatively better than the improvement of the other class. However, this was not the main finding of the evaluation. The results indicated that Jeliot 2000 helped in establishing a vocabulary of verbal and visual terms of programming concepts, such as loops and conditional statements. The availability of this vocabulary facilitated the discussion of programming concepts and improved understanding.

Especially the mediocre student seemed to benefit the most from the visualization while the weaker students felt the visualization was still too difficult to grasp. The stronger students refused to use the tool because they thought they did not need it. This proves that different kinds of user populations need specialized tools for their use.

3.4 Implications to the Design of Jeliot 3

Eliot and Jeliot I share a similar kind of architecture. Both of them used an approach where the user code was translated to the closest programming language (i.e. Eliot-C was transformed to C++ and EJava to Java) and then compiled with a standard compiler. This made the framework stable and the visualization of programs semi-automatic but restricted the abstraction level of the visualization by only visualizing the variables of the programs. Especially, the empirical evaluations with Jeliot I gave many insights into what users want from the visualization system (see the proposition

list in the Section 3.2.4). Those issues were also taken into consideration as much as possible in the design of Jeliot 3.

Jeliot 2000 took a different approach to program visualization and used an interpreter as the model and the visualization only was a side effect of the program interpretation. The design allowed more complete visualization of the program's data and control flow as requested by the novices. However, the design of Jeliot 2000 was considered impractical for direct further development. There were two problems in the design of Jeliot 2000. Firstly, the interpreter and the visualization engine were strongly coupled, which meant that modifications to the visualization of the programs in Jeliot 2000 would lead to modifications in the interpreter. Secondly, the used interpreter was hand-crafted and its further development would have taken much effort. This led us to find new possibilities in the form of ready-made Java interpreters. Nevertheless, the visualization engine of Jeliot 2000 was modular and it could be separated from the interpreter with reasonable modifications. The visualizations were also found to be effective in classroom settings with novices (Ben-Bassat Levy et al., 2003). The new system was also meant for novice programmers and thus, we decided to use a similar kind of approach in Jeliot 3. This means that Jeliot 3 should consist of an interpreter and a visualization engine. Visualization should be complete and continuous as in Jeliot 2000 and the user interface should be easy to use. However, the design of the systems should be more modular.

To sum up, our aim from the historical perspective is to form a system which is modular like Eliot and Jeliot but which has the usability, design and visualization capabilities similar to Jeliot 2000. Moreover, the system should be able to implement new kinds of features that where found important for novices in the empirical evaluations, such as visualization of object-oriented constructs and reasonable error messages.

4 Modularity

Modularity was risen as one of the key issues in the design of Jeliot 3. Jeliot should be internally and externally extensible without large modifications. Because several software visualization systems have been developed from different perspectives and needs, a number of models for software visualization systems have also been proposed. In this chapter I will introduce some proposals other researchers have made for the models of software visualization system and outline our solution in connection with the previous work and recommendations.

4.1 Models for Modularity in Software Visualization

A model for a software visualization system was introduced by Price et al. (1993), who used it as a basis for their taxonomy. The model identifies four different parties: the programmer of the visualized program, the software visualization system developer, the visualizer designing the visualization of the program and the user of the visualization system. The model is based on the basic model of any software in which user gives input to the program which after computation produces output that is evaluated by the user. The model for software visualization system is illustrated in Figure 16.

In the model, the different parties give their parts of the input data: The software visualization system developer has programmed the features that the system can visualize, the programmer produces the program to be visualized, the visualizer gives the specifications for the particular visualization and the user commands the visualization system. These parties can be all different people but in some contexts all the parties can be represented by a single user.

The visualization system visualizes the program as its output that is then viewed and interpreted by the user. This model is abstract and does not give much basis for the modularization of the software visualization system. However, when looking at the input data we can see that different components are needed to handle the separate input data. The user should be able to command the program through user interface, visualization specifications should be managed by the visualization engine, and the program should be analyzed by some part of the software visualization system that also knows which aspects of the program are visualized.

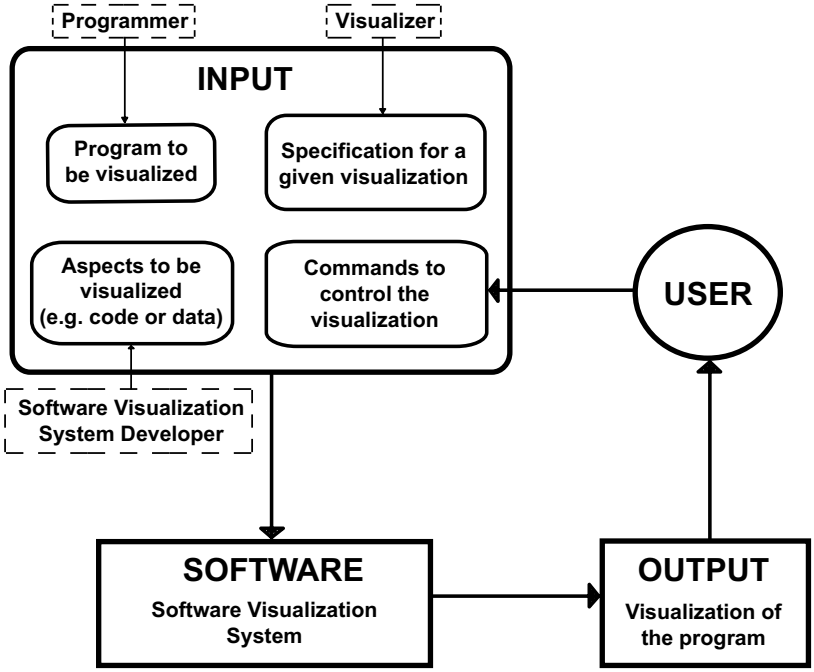

Figure 16: Model of Price et al. (1993) for software visualization system.

Another kind of model was taken as a basis for their taxonomy by Roman and Cox (1992, 1993), presented in Figure 17. Similar participants can be identified as in the model of Price et al. (1993): the programmer is the same in both models, the animator corresponds to both the visualizer and the software visualization system developer and the viewer is the same as the user.

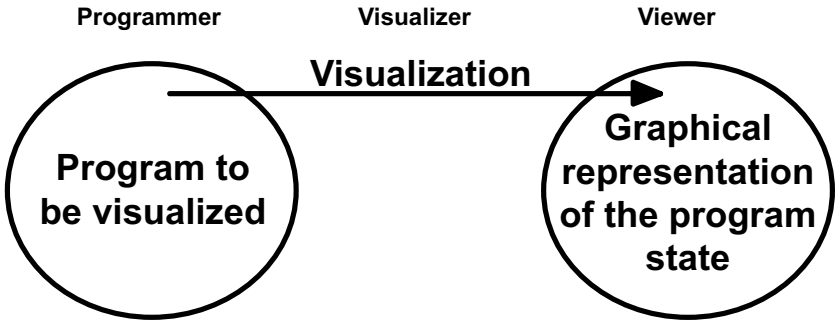

Figure 17: Model of Roman and Cox (1993) for software visualization system.

In the model of Roman and Cox (1992, 1993), the program and its graphical presentation are separated, and a mapping in between them is proposed. This mapping is described as a direct relation between the program state and its visualization, meaning that a strong decoupling between all the forms of the program (e.g. its code and its

execution) and its visualization should be made. The visualizations can be specified in many ways and they can represent different levels of abstraction. Also their presentation techniques can vary. This model gives an interesting starting point for the architecture of the software visualization system separating the visualization from the program execution. In Eliot, Jeliot I and Jeliot 2000, the execution of the program and its visualization are not separated from each other. However, in Eliot and Jeliot I the connection is looser than in Jeliot 2000.

Roman et al. (1992) used the model of Roman and Cox (1993) in Pavane program visualization system. In Pavane, the model was further developed into several separations between different mapping spaces that are connected with declared rules. The model of the Pavane's visualization is shown in Figure 18.

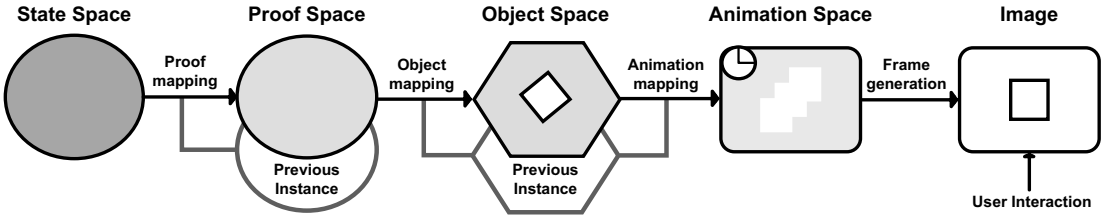

Figure 18: Model for visualization in the Pavane program visualization system (Roman et al., 1992).

In the model, the mapping between the state of the program and its visualization are separated into four different mappings. First, the program *state space* is mapped into *proof space*. Proof space is an abstraction of the program state that combines the relevant information from the current program state and the history of the computation connecting both the data and the control flows of the program.

The state of proof space is mapped together with the previous state of the object space into the new *object space*. This is a space of abstract three-dimensional geometric representation of the current proof space instance. The objects in the space are presented as tuples that describe an abstract three-dimensional world of objects. It is abstract in a sense that there is no one-to-one mapping between the graphical objects and these abstract objects, but one abstract object can be represented by several graphical objects and vice versa.

Next mapping between object space, previous object space and *animation space* changes the abstract objects into three-dimensional objects that can be rendered to produce

the image that the user sees. This mapping also needs to take care of the timing of the animation so the objects are actually four-dimensional, where their fourth dimension is time.

Finally, there is a one-to-one mapping between animation space and *final images*. Each of the animation objects are rendered in the frame generation mapping producing an animation of the object. Each frame of the animation is a snapshot of the four-dimensional world of the animation objects. In this way, Pavane system produces an animation of the program. The design of a declarative program visualization gives multiple possibilities to visualize the same program from several perspectives without changing the program but the mappings. However, the design of this system is complicated and even though this kind of visualization can be automatized to some extent, it cannot be done fully automatically, which is a problem to a system meant for novice users. Leonardo uses a similar kind of approach to declare visualization with a logic-based language (Crescenzi et al., 2000).

A framework to produce program visualization systems and a system implementing the framework, Viz, was proposed by Domingue et al. (1992). The authors claim that the framework concentrates on the *programming language visualization* that overlaps program and algorithm visualization, concentrating more on the program visualization. Its main concern is to visualize the language constructs of as wide a variety of target language programs as possible. This is similar to Jeliot in which one of the goals is the same. The framework is based on the visualization of the history data from the program execution. The architecture of Viz system is illustrated in Figure 19.

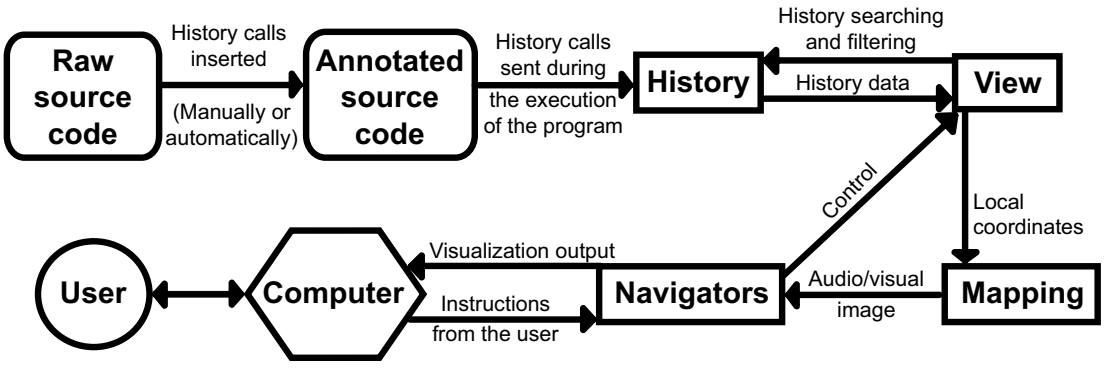

Figure 19: Model for visualization in the Viz programming language visualization system (Domingue et al., 1992).

In the model, the *source code* of the visualized program must be *annotated* to produce *history* of the program execution. The *view module* inspects the history of the program execution when the visualization is run. The view sends the visualized history data to the *mapping module* that decides the appearance of the data object, whereas the view module is only concerned about the location of the data object in the display. The navigator module transforms and presents the data objects on the screen. The user can also interact with the navigators and control the view module by zooming, panning or moving forward or backward in time in the program execution. The model is similar to model of Roman and Cox (1993) in a way that they both propose the separation of the program and its visualization. However, the model of Domingue et al. (1992) does the mapping between the program and its visualization in a much lower level than what was proposed in the model of Roman and Cox (1993) and especially what was implemented in Pavane (Roman et al., 1992).

The ideas of Roman and Cox (1993) and Domingue et al. (1992) were combined in the model of Stratton (2001, 2003) in which the debugger and visualization engine communicate with each other to form a visualization tool. The model proposal is shown in Figure 20.

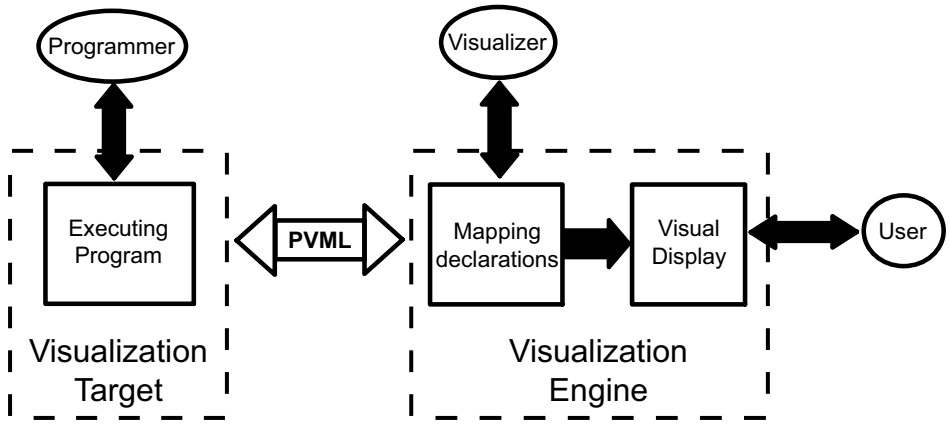

Figure 20: Model for program visualization system by Stratton (2001).

This model makes the explicit separation between visualization target executing the program and the visualization engine that handles the program state mapping and graphics displaying. This separation is made with a language called program visualization meta language (PVML) that is made for both controlling the visualization target from the visualization engine and transferring the program state information extracted during the execution of the program.

The visualization target is supposed to be a debugger that runs a program and extracts the program state information from it in PVML. The PVML stream is then sent to the visualization engine, and the mapping to the visual objects is made in a similar fashion as in the model of Roman et al. (1992).

In the reference implementation of the PVML, Stratton (2003) uses XML-based language to connect GDB and JDB debuggers to the visual debugger GUI on another machine. With a programming language independent meta language it is made possible to visualize several programming languages with a single visualization engine. The idea of programming language independent meta language is similar to the history data in the model of Domingue et al. (1992).

There are a few problems in the design of this approach. One is the granularity of the information. The debuggers give out much information that needs to be filtered, and the level of abstraction on which the debuggers are able to present the execution is insufficient, for example, for expression evaluation visualization. Another problem relates to the usage of XML. The XML generation has to be stopped at the input data requests during the program execution because of the well-formedness requirement of XML. This can generate extra communication between the target and the engine.

A general model for modularity in visualization systems was discussed in the ITiCSE 1996 working group on visualization (Bergin et al., 1996). The proposal uses the model-view-controller (MVC) design pattern illustrated in Figure 21.

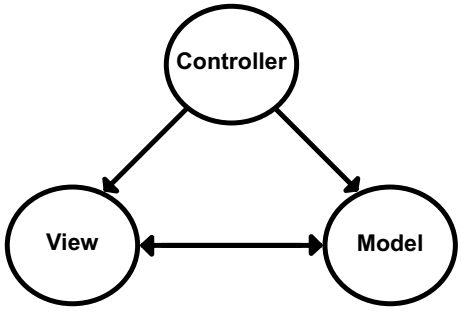

Figure 21: Model-view-controller design pattern (Bergin et al., 1996).

In program visualization context, the model package in MVC could be, for example, the history data of a computation, an observer of the virtual machine through a debugging interface or an interpreter. The view could be anything from static images describing data structures to animations of data flow in the virtual machine or anima-

tions of the data structures state changes according to the model. The model informs the view when it has changed, and the view updates itself according to the changes in the model. The controller is normally the user interface. It lets the user modify the model and the view by textual or graphical interface and through direct or indirect manipulation. Normally, systems only have one model but they can have several views and controllers.

The MVC design pattern is used successfully in the design of Javavis program visualization system (Oechsle and Schmitt, 2002). Javavis connects to a Java virtual machine and updates the model of the computation state through an interface defined for debugging by the virtual machine, Java debugging interface (JDI). The design of Javavis system is depicted in Figure 22.

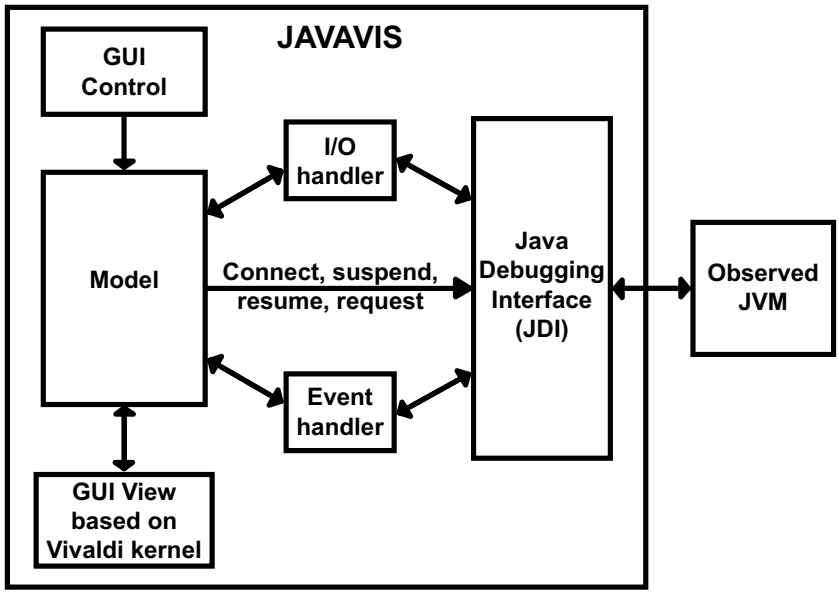

Figure 22: The architecture of JAVAVIS (Oechsle and Schmitt, 2002).

In terms of MVC design pattern, there is one model of computation in Javavis. This model keeps track of the current state of the program that is evaluated by the virtual machine. In this design, there are two controllers for the model: the graphical user interface that can control both the model and the view and the virtual machine that transmits the information of the program execution to the model. There is one view that renders the model on the screen. The view is based on a general purpose graphical library called Vivaldi that is extended with the library that can be used to produce UML-based diagrams and animations.

Also another model was introduced by Bergin et al. (1996) in the form of a data flow diagram shown in Figure 23. This model is similar to model of Roman et al. (1992) where the procedure of the program animation is just mappings from one space to another. In the naming, there are also similarities to model of Domingue et al. (1992). This model also suggests the use of history data of the algorithm that can be visualized after the execution by mapping, animating and rendering the filtered history data. It seems that in this way, the algorithm execution and its visualization can be separated. However, the interface used to connect these two is not proposed or defined in any way. This kind of definition could ease the development of the different visualization systems as also proposed by Stratton (2001).

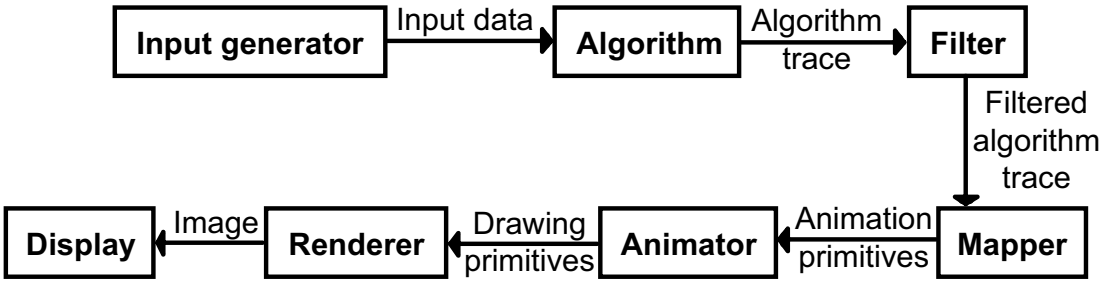

Figure 23: Data flow of an algorithm visualization system (Bergin et al., 1996).

To sum up the different models, the visualization systems should separate the execution of the program to be visualized and its visualization. This can be done in many different levels, but the more abstract the level, the more flexible the visualizations are. For example, in Javavis the connection is made on the level of programmable API, and adding a new visualization of the model can be more complicated and can require changes in the model. The model proposed by Stratton (2001) separates the systems with an abstract language that should be able to produce different kinds of visualizations from one program execution defined by that language. Same should apply to Viz system. Pavane takes a different kind of approach with several separations between the executed program and its visualization.

4.2 Model for Modularity in Jeliot 3

As one of the requirements for Jeliot 3 was the extensibility, the modularity of the system is in the key role. As stated before in Section 3.4, the previous versions of Jeliot

raised some propositions that should be considered in the design of the new system. The use of an interpreter in Jeliot 2000 was found a good way to make the program visualization complete. The analysis in Section 4.1 showed that most of the propositions for visualization system models separated the execution of the program to be visualized and the visualization engine that visualized the execution. Both of these issues were taken into consideration when designing the model for modularity. Proposition of Stratton (2001, 2003) for the program visualization meta-language was important in the sense that it gave us a starting point when thinking the interface between the interpreter and the visualization engine.

The structure of Jeliot 3 is shown in Figure 24. The user interacts with the user interface and creates the source code of the program (1). The source code is sent to the interpreter and the intermediate code is extracted during the execution of the code (2 and 3). The intermediate code is interpreted and the directions are given to the visualization engine (4 and 5). The user can control the animation by playing, pausing, rewinding or playing step-by-step the animation (6). Furthermore, the user can give input data, for example, an integer or a string, to the program executed by the interpreter (6, 7 and 8).

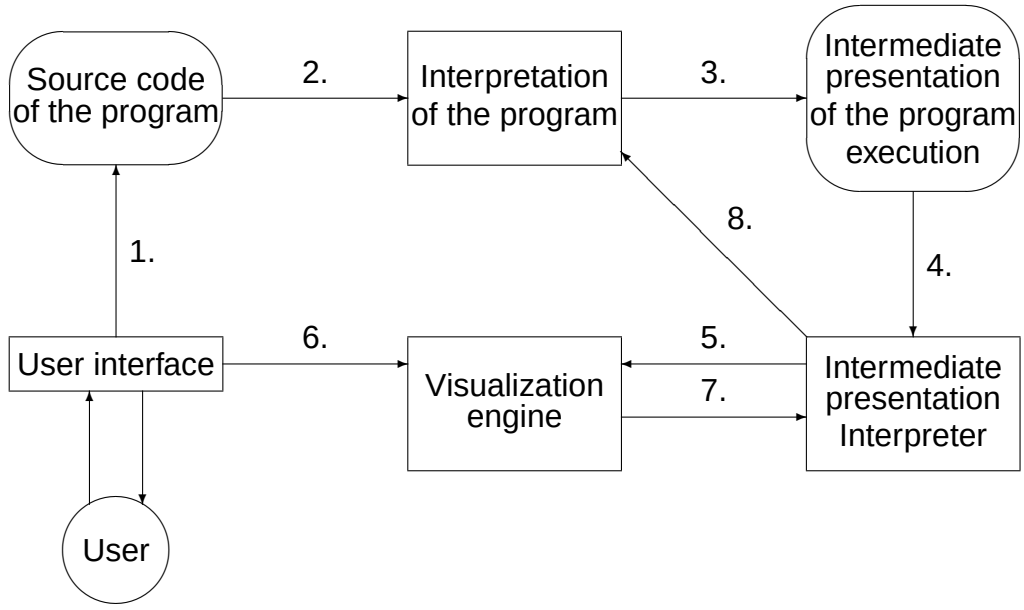

Figure 24: The functional structure of Jeliot 3.

This model can be thought as a hybrid of the previously proposed models of Bergin et al. (1996), Domingue et al. (1992) and Stratton (2001). The similarities between our model and these models are discussed below.

Our design implements the MVC design pattern when the user interface is considered the *controller*, program code, its interpreter and the intermediate presentation of the execution are the *model*, and the intermediate presentation interpreter and visualization engine make up the *view* (cf. Figure 21).

On the other hand, we have a similar design with Domingue et al. (1992) in which the program code is run and history data is collected from it. Our system uses the interpreter to run the code and extracts history data in the form of the intermediate presentation. This history data is then visualized with a procedure that differs in our design but in any case has a similar kind of an approach. The main difference is that our system is designed to work only on-line meaning that visualization is done during the interpretation, whereas Viz system seems to visualize the program history data after the whole program is executed making it more post-mortem.

The model of Stratton (2001) is the most similar when the program code and its interpreter are considered the visualization target, the intermediate presentation the program visualization meta-language, intermediate presentation interpreter the mapping declarations and the visualization engine the visual display. The difference is that instead of using a debugger we use an interpreter as the visualization target.

4.3 The Consequenses of the Model

A design for modularity, such as the one described above, makes use of two separate systems that are connected with interface of intermediate code. The interpreter can be a ready-made one if the intermediate code can be extracted during the interpretation with only little modifications to the interpreter. Similarly, the visualization engine can be of any kind, as long as the intermediate code interpreter can add calls to the visualization commands and the requirements for the visualization are met. The possible reuse of these systems is discussed further in Section 5.

The design also means that an intermediate language is needed. The possibilities for the intermediate presentation are evaluated and the designed intermediate language in-

troduced in Section 6. There are certain requirements for the intermediate presentation: It should be usable for several different kinds of visualizations meaning high enough abstraction level and it should not be complicated in order for it to be interpreted with relatively low effort compared to the high level programming languages. Otherwise, there are no good reasons to make this kind of an intermediate presentation.

5 Reuse of Existing Systems

The separation of the program execution and its visualization in the design made it possible to use existing systems to implement Jeliot 3. In this chapter, I introduce the selected systems and explain the reasons why these systems were chosen over any other systems.

5.1 Java Language Interpreter

When looking for Java source code interpreters, we found three existing and maintained open-source systems: *Iava* (Richter, 2000b), *BeanShell* (Niemeyer, 2003) and *DynamicJava* (Koala Project, 2002). In this section I introduce briefly these systems and evaluate our decision to select DynamicJava which is presented more closely in Section 5.1.2.

5.1.1 Comparison of the Systems

The similarities and differences between BeanShell and DynamicJava are analyzed by Hightower (2000). I have used this article as the basis for the analysis of these two systems. Although both of these systems have been developed further, the structure and purpose of these systems have stayed the same. I have also gathered the information from the websites of all three systems as well as from other articles written about them if any is found. The summary of the findings are shown in Table 1.

Iava is a scripting language that is a subset of the Java programming language (Richter, 2000a). It also contains a scripting language interpreter that can be integrated into other Java applications. This means that Iava can be used seamlessly to merge the compiled and the scripted parts of the program. Iava is an interpreter that only accepts those parts of Java language that can be processed by using Java and its reflection API. Iava is also capable of declaring new methods and variables and interpreting all statements. The interpreter does not accept any kinds of class or interface declarations. This means that Iava interprets only a small subset of Java language, it is not possible to create new classes with it and the syntax is Java like but not Java.

Table 1: Comparison between the different Java source interpreters.

	Iava	**BeanShell**	**DynamicJava**
Level of Java compatibility			
Statements and expressions	+	+	+
Own method declarations	similar	similar	+
Own class declarations	-	similar	+
Precompiled classes	+	+	+
Support for Java API	+	+	+
Other features			
Scripting features	a few	many	a few
Support and development	inactive	active	only if Java language specifications change
Known uses	Reported in Richter (2000a)	BlueJ	DrJava
Documentation	two articles in Richter (2000b)	user guide, map to the source code and JavaDoc	JavaDoc
Size	c. 140KB	c. 210 KB	c. 475 KB

BeanShell is a Java source interpreter that executes the standard Java statements and expressions and its own scripting commands. It does not support the class declarations in same way as Java but it supports object scripting language features in a similar way as JavaScript. This means that classes can be declared but with different syntax than in Java. However, Niemeyer (2003) claims that the new version of BeanShell (version 2.0 beta 1) is fully compatible with Java and can handle class declarations in same way as in Java. This version of BeanShell was released after the decision of the used interpreter was made so in the rest of the comparison the information about the old version is used.

BeanShell can interact with precompiled classes and construct objects from them. Method calls are also similar to Java language specifications. In addition, some of the reserved words have new meanings (e.g. *this* and *super* refer to different scopes

of the program). In this way, BeanShell interprets a subset of Java language and adds several scripting features, such as UNIX-like shell commands, top-level scripting and loosely typed variables. BlueJ uses BeanShell to implement expression evaluation (BlueJ, 2003).

DynamicJava is a Java source interpreter written in Java. At the moment, DynamicJava is almost fully compliant with Java language specifications (Gosling et al., 2000). It supports even multithreading and inner-classes, although these are not needed by the novice programmers. In addition to this, it contains some scripting features such as statements and methods can be defined outside the classes on top-level. Thus, DynamicJava interprets a superset of Java language adding some scripting features to the language. DrJava (Stoler, 2002) uses DynamicJava to interpret programs in its programming environment.

Each of the systems have their own strengths and weaknesses, as can be seen in Table 1. Iava is small and easily adaptable into other systems. BeanShell contains many scripting features, is relatively small and is actively developed. DynamicJava is closest to Java language specifications and is only changed if Java language specifications change (Hightower, 2000). When looking for a Java language interpreter as close to the Java language specifications, DynamicJava is a good choice. Both Iava and BeanShell contain too many restrictions (e.g. no proper class declarations) and possibilities to run code that is far from Java language specifications (e.g. scripting features) to be used as interpreters in a visualization system visualizing Java programs for novices. Thus, the most important reason to select DynamicJava was the compatibility with Java language and the number of scripting features. DynamicJava contains only a small number of scripting features that the novices do not need to know or learn. In Iava and BeanShell the scripting features are in connection with, for example, the class declaration and thus novices need to learn a non-standard way to handle class declarations as well as the scripting features of the systems.

5.1.2 Introduction of the Selected System

DynamicJava was selected as Java source interpreter for Jeliot 3. We wanted to change DynamicJava as little as possible in the integration process. In this section, I introduce the system and tell how the modifications were done in general.

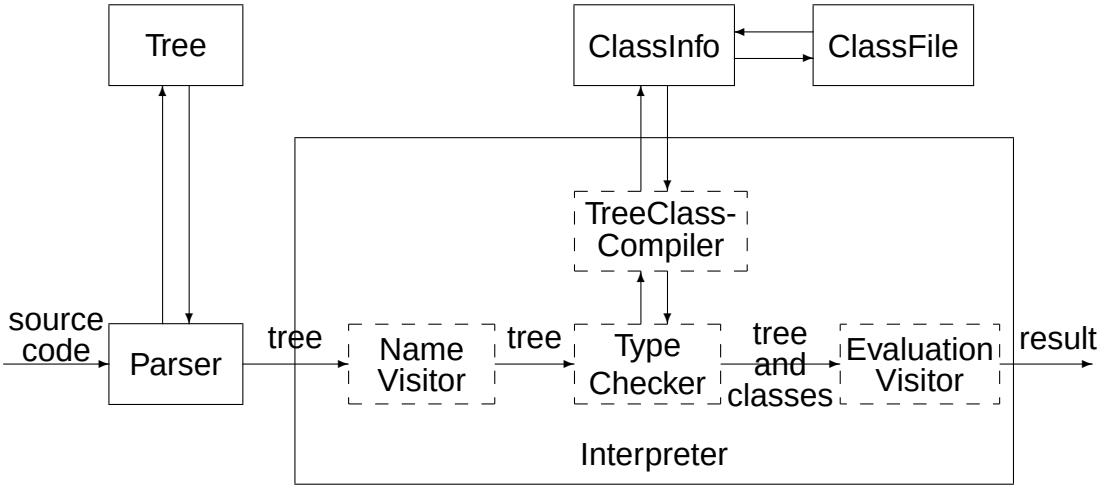

Figure 25: Packages, visitors and data flow in DynamicJava (Moreno and Myller, 2003b).

The functional structure of DynamicJava is shown in Figure 25. The figure explains the main relationships between the packages, the visitors in the interpreter package and the main data flow.

DynamicJava consists of seven different packages, where only five of them are mainly used during the interpretation: *classfile*, *classinfo*, *interpreter*, *parser* and *tree*. The other two packages, namely *util* and *gui*, are used, for example, to help the debugging of DynamicJava, to provide information during the interpretation and to create a user interface for the interpreter.

The *parser* package provides the classes that compose the default parser for the language. The Parser creates the nodes of an abstract syntax tree to be traversed later by tree visitors in the *interpreter* package. The *tree* packages contains classes and interfaces for producing an abstract syntax tree. The created tree consists of nodes that all have some common properties. Subclasses of the node class are defined to address the unique properties of each different Java construct (e.g. expressions and methods). For example, a node for any binary expression will also consist of the properties ‘left expression’ and ‘right expression’.

The *interpreter* package contains classes that help the interpretation of the given abstract syntax tree. As can be seen from Figure 25, after the source code is parsed and the abstract syntax tree is formed, three visitors go through the tree, namely *Name Vis-*

itor, *Type Checker* and, finally, *Evaluation Visitor*. Evaluation Visitor receives also the compiled classes from Type Checker as they are loaded into the Java virtual machine.

The *Name Visitor* is a tree visitor that resolves the ambiguity in identifiers in the abstract syntax tree. The *Type Checker* is a tree visitor that checks the typing rules and loads the classes, fields and methods. Type Checker class is not only concerned about typing rules. When visiting a class declaration in the tree, Type Checker invokes *Tree Compiler*, which compiles the class into Java bytecode by using the classes in *classfile* package. However, this compiling process *alters the class* and the generated bytecode *does not match the original source code* of the class. This introduces problems especially in the object allocation and inheritance. The reason for this compilation is that later on during the evaluation of the syntax tree the evaluation visitor do not need to separate the method or constructor calls made into the user made classes from the method or constructor calls made into the classes of Java API.

Classfile package contains all the classes for creating general purpose bytecode from the user defined classes. The *Class File* instance from classfile package controls the class's bytecode generation process. *Classinfo* package contains all the classes and interfaces for using reflections on Java or interpreted classes. This package is used during the compilation of the classes.

The *Evaluation Visitor* evaluates each node of the abstract syntax tree. This visitor performs the evaluation and execution of the program by traversing the syntax tree and using the compiled bytecode classes that are loaded into the Java virtual machine. The execution process can be observed when Evaluation Visitor traverses the syntax tree and the normal interpretation procedure can be used to extract information for the visualization.

The interpreter package also contains a class called *Tree Interpreter* that can be used to control the different stages of the interpretations. This class can be used as the entry point to the system. Also some of the interpretations are done in this class and in some special occasions (e.g. method invocations) information needs to be extracted from this class.

5.2 User Interface and Animation Engine

When considering the different possibilities for the animation engine and the user interface we had a few possibilities. We could have used one of the animation libraries, such as *Animal* (Rößling and Freisleben, 2002) or *Jawaa* (Pierson and Rodger, 1998) and built our system on one of them. The problem we saw with these systems was that we would have had to build the visual object handling from the beginning as these systems do not support this kind of program visualization but are more related to algorithm visualization. Animal supports relative object placement that would help this process but still the animations would have had to be designed from the beginning.

In Javavis, *Vivaldi* graphics kernel to render the animation of the program execution (Oechsle and Schmitt, 2002). During the development of Javavis, a visualization library using Vivaldi was implemented for the program animation. The visualizations of Javavis are on a higher abstraction level than what was supposed to be visualization level of Jeliot 3. It only shows the objects, variables and their values without showing the expression evaluation. This means that adapting the visualization library would require designing of new kinds of animations.

Also Jeliot 2000 was considered because it had been found usable and helpful in the empirical evaluation (Ben-Bassat Levy et al., 2003). As introduced in Section 3.3, the design of Jeliot 2000 is modular, but the problems were in the combined visualization engine and interpreter. However, it was possible to separate the visualization engine from the interpreter with some modifications. In this way, the modified version of the visualization engine could be produced with less effort than building the whole visualization engine from the beginning or using other existing visualization tools or libraries would have required.

The requirements for the features of the source code editor and the user interface were simplicity and ease of use. As the visualization engine was already used in Jeliot 2000, we decided to use its user interface as well. It is very modular in design and found usable by novices in the experiment (Ben-Bassat Levy et al., 2003). The user interface did not require any modification to work in Jeliot 3. However, many convenience modifications were done to make the usage even easier, especially during lectures. The user interface and the visualizations are discussed in Section 7.

6 Intermediate Languages in Program Visualization

As a consequence of the design for the Jeliot 3 system, an intermediate language between the Java language interpreter and the visualization engine needed to be designed. I introduce here the possible starting points for the design of the intermediate presentation. A few historically interesting intermediate languages and recent developments are briefly explained. Then, the language designed for Jeliot 3 is explained and the reasons for the approach taken are discussed. A more detailed discussion of the software visualization intermediate codes is found in Moreno (2004).

6.1 Different approaches

In this section three different approaches are discussed. Firstly, two systems that use a virtual machine and a assembly language like language to command the virtual machine. Secondly, new development that uses an XML-based language is introduces. Finally, the visual scripting languages are discussed.

Dynalab (Boroni et al., 1996) is a program visualization system which is build on a virtual machine called *Education Machine* (*E-Machine*). The virtual machine emulator runs a low level source code, *E-code*. This combination permits the reverse execution of programs which is the most appealing feature of the Dynalab system. Several compilers have been implemented for compiling programming language source codes to E-code. At the moment, full Pascal compiler and partial C and Ada compilers exist to compile programs to E-code. E-code is a low-level language, describing the program in a similar manner as assembler languages or Java bytecode. My description of the E-code is based on the work of Pratt (1995) who implemented a visualizer for E-Machine and modified E-code for these needs. For a more detailed description see Pratt (1995).

In the design of E-machine and E-code, the aspects of the program visualization are taken into consideration. E-code is divided into packets which correspond to a high-level programming language (e.g. Pascal) statements or expressions. By executing the whole packet of E-code sentences, E-Machine can resemble the execution of the high-level programming language. After executing one packet in E-Machine, the corresponding animation is shown by the visualizer. This procedure is repeated to produce the animation of the program execution.

E-code had three requirements: structures for easy implementation of high level programming language constructs, a simple way to implement everything related to methods and the ability to execute either forward or backward. The first two requirements could be solved with an assembler like language that had a limited set of commands, totally 43 instructions. The main problem was to make the reverse execution possible. The taken approach tries to minimize the stored information for the reverse execution. This is achieved by storing only the differences between each of the steps. However, as the reversing should be done on the level of a single high-level programming language statement or expression, the E-code sentences need to be marked either *critical* or *non-critical*. Sentences erasing information that is needed for reverse execution are critical and others non-critical. Thus each E-code instruction is composed of four fields in the given order:

- An opcode mnemonic (e.g. push, pop or add) telling the E-Machine what kind of an operation it should perform.
- A flag that marks the instruction either as critical or as non-critical. If the instruction is marked critical, it means that some information that is essential for reverse execution is deleted and should be saved before the execution. Otherwise, there is no need to take care of saving intermediate values. In this way, the amount of saved values is limited.
- A field that denotes the data type of the operand(s) of the instruction.
- A field that contains either a data value or an addressing mode. The addressing modes are developed for high level programming languages to handle, for example, variables, pointers and arrays in E-code. Depending on the executed operation either a data value or an addressing mode is required.

E-code sentences are written in the file in ASCII characters and the opcode is in the beginning of the line separated by spaces from the next fields. The rest of the fields are written one after the other and separated with a comma.

A similar kind of an approach was taken in *Leonardo* and its intermediate code by Crescenzi et al. (2000). In Leonardo, C programs that contain animation declarations in Alpha functional programming language are first compiled with C and Alpha compilers. Both compilations form a set of commands in a low-level assembly-like language. The format of the commands is: `opcode–operands–opcode`, when there

can be a various number of the operands. These commands can be then run by the virtual machine called virtual execution environment. The reverse execution is done in a similar way as in Dynalab. The reverse execution takes place in the virtual machine as in E-Machine, and the sentences of the language are marked either as sentences erasing information needed for the reverse execution or as sentences not erasing information. The instructions can be divided into four groups: control flow instructions, data flow instructions, logic-arithmetic instructions and cast instructions.

E-code and the intermediate code used in Leonardo differ from the needs of our intermediate language. In our case, the intermediate language should describe the execution of the program, whereas in both of the explained systems, the source code is just transformed into another presentation, virtual machine code, before the execution.

Stratton (2001, 2003) introduced an XML-based language, *a program visualization meta-language (PVML)*, that was made to describe the program state information during the execution of the program. The architecture of this solution is discussed in the previous Chapter 4.1. In this chapter, I concentrate on PVML communication between the visualization target and engine.

PVML is used to both describe the program state (e.g. variable values) and facilitate the communication between visualization target and engine (e.g. pausing or continuing the execution). The program state information is wanted to be kept as general in a sense that PVML does not have any visualization oriented and very little paradigm or programming language oriented commands. Thus, the use of it could be as wide as possible. However, depending on the level of detail in the execution data wanted, this can be a problem because some programming language concepts that should be visualized cannot be visualized from too general program state information. For example, if the program state is only described with the values of variables, which is normally the case in debuggers, it is impossible to visualize the expression evaluation from this information.

PVML has an XML DTD that contains 57 different tag names. However, at the moment, the language is still immature and does not contain enough support for program visualization on the level that would be needed in Jeliot 3. For example, at the moment it is not possible to extract information about expression evaluation in PVML. PVML is also very verbose and takes a lot of excess space.

The visual scripting languages such as Animal (Rößling and Freisleben, 2002) or Jawaa (Pierson and Rodger, 1998) provide a totally different viewpoint to the intermediate presentation. They provide some graphical primitives oriented to data structure visualization and simple graphics manipulation. The commands can be written in plain ASCII and they are then interpreted by the visualization engine. The visualization engine of Jawaa is also suitable for web viewing as an applet. However, this approach would require several data structures and modifications to the Java interpreter that would handle the locations of the different elements and the animation generation. Depending on the visual scripting language used, the absolute locations of the elements need to be used for every animation. Furthermore, the locations or some other reference to the currently shown visual elements needs to be stored for the animation of assignments and complex expressions. An intermediate presentation that is directly related to visual objects prohibits the use of the intermediate presentation for another kinds of visualizations. In addition to this, these systems offer no support for interactivity such as input handling. This means that the program needs to be run first before the visualization can be viewed. However, we wanted that Jeliot 3 is an interactive system that the user can control during the execution of the program and thus, this approach was not further studied.

6.2 Our approach

The interface between two systems in Jeliot 3, namely visualization engine and DynamicJava, is formed by using an intermediate code that is extracted during the evaluation of the program. First, our idea was to use an XML-based approach similar to Stratton (2003). However, we found a few problems with this approach. It would have made the program much larger because of XML-parser related issues. The well-formedness requirement of the XML would have caused some problems when describing the program state, especially with the programs requesting input from the user (Stratton, 2003).

After this, we decided to use a hybrid solution between the approach taken in E-code, intermediate code of Leonardo and PVML. We call this approach *m-code*. M-code is similar to E-code in a way that each m-code sentence is formed in the same manner. Every line contains a single command. Each of the commands can be divided into smaller pieces, tokens. The first token of the command is the opcode of the sentence.

Then comes the possible operands that differ for each opcode. However, m-code is not just another form of the program source like E-code is, but it describes the program state changes during the execution of the program as PVML does, only on a more detailed level.

There are 72 different opcodes in total. These opcodes can be divided into seven different categories:

- *Auxiliary opcodes* are used to help the interpretation of the complex expressions such as binary expressions and assignments. They are also used to inform the visualization engine about an error and the end of the program.
- *Expression opcodes* describe all kinds of expressions, such as binary and unary expression, assignments, literals and variable accesses.
- *Control structure opcodes* are used to describe the results of the control structures.
- *Method call opcodes* are used for static and object method calls as well as for return values and parameters.
- *Input and output opcodes* are used when input is requested from the user or output is generated.
- *Object and array opcodes* are used for object field and array accesses and array creations.
- *Class information opcodes* are used when the interpreter extracts information about each class (e.g. field names and types) before the execution of the program starts.

In the notation used, each token is separated by a single character that is not used elsewhere. To keep track of the different commands, a program counter was introduced. The counter is used to identify each of the commands. Normally, the program counter is the first operand of any command whose value is used. When the program counter is referenced, it means that the value of the referenced command is used as a operand of the referencing command. For example, in binary expressions, there are normally two program counter references and in unary expression there is only one expression

counter reference. In this way, the values of the complex expressions can be connected to each other. For location in the source code, an operand that contains the beginning line, the beginning column, the ending line and the ending column is used. Other operands can be the value of the expression, the type of the expression and a variable, a field, a method or a class name.

As the language is evaluated one line at the time, the addition of new commands is relatively easy and should not affect the execution of the other commands. The interpreter of the intermediate code needs information only of what it should do when each command and its parameters are read. It is also possible that some of the commands are not even process by the interpreter if that is not necessary. For example, a visualization of the method call tree would only need to process the commands related to methods. The relations between lines are handled mainly with the expression counter references. The only differences are binary and unary expressions as well as methods. The binary and the unary expressions use 'begin' command to indicate the beginning of the expression evaluation and the methods have separate commands for beginning and ending the method invocation. Those relations should be appreciated by the intermediate code interpreter to get correct results in the visualizations.

The abstraction level of the intermediate language is high enough to allow automatic generation of several different kinds of visualizations. The language is not connected to the visualization but it just expresses the data and the control flow of the program during the execution. An example of the language is show in Figure 26. The example shows what kind of an intermediate code is produced during the execution of one Java source code line `a = b + 1` in a larger program when `b` has value `1` before the execution of this statement.

The first command is `Begin` which has three parameters: the expression it begins (`Assignment`, the reference to the expression counter value of this expression (`2`) and the location of the expression in the source code `1,1,1,10`. This means that next the right-hand-side of the assignment is evaluated and the second `Begin` command is printed. Its parameters are the same as before, this time the values are just related to the addition expression (`AddExpression`). The command `Left` explains that its only parameter (`3`) is an expression counter reference to the left side of the previous expression, in this case the addition expression. This is needed especially with more complex expressions. The value of the variable `b` is given next in the command qualified name (`QualifiedName`) and its parameters. The first parameter is the ex-

Java code:

```
...
a = b + 1;  //when b has value 1.
...
```

Intermediate code of the execution:
(opcodes are changed to corresponding names)

```
Begin|Assignment|1|1,1,1,10
Begin|AddExpression|2|1,5,1,10
Left|3
QualifiedName|3|b|1|int|1,5,1,6
Right|4
Literal|4|1|int|1,9,1,10
AddExpression|2|3|4|2|int|1,1,1,10
To|5
QualifiedName|5|a|UnknownValue|int|1,9,1,10
Assignment|1|2|5|2|int|1,1,1,10
```

Figure 26: An example of the intermediate language.

pression counter value, second the variable name, third the variable value, fourth the type of the variable and finally the location of the expression in the code.

The command `Right` is similar to the command `Left` but it just informs the right side expression counter reference. Then the `Literal` command is printed out with its parameters: the expression counter value, value of the literal, the type of the literal and its location in the source code.

When the left side and the right side of the expression are known, the expression can be evaluated, and the next command `AddExpression` and its parameters explain the result. The first parameter is the expression counter value (cf. second parameter of `Begin|AddExpression` command). The second and third parameters are the left and right expression counter references respectively. Fourth parameter is the value of the expression and the fifth the type of the expression value. The last parameter is the location of the expression.

The command `To` is used in connection with assignment expression to indicate the variable or field in which the assignment is made (i.e. the left side of the assignment expression). Next the information about the target of the assignment, variable `a`, is given with the command `QualifiedName` as already introduced before. Finally, the assignment is evaluated and the result of the evaluation is shown in the command `Assignment` and its parameters. The first parameter is the expression counter value. Second and third parameters give the right side and the left side of the expression respectively. The fourth and fifth parameters express the value and the type of the expression and the sixth parameter explains the location of the expression in the source code.

7 Design of the User Interface and Visualization

Research has been done on the field of software visualization but only a small amount of this work has been dedicated to the design principles for the visualizations. I review some of the work done on this field and on a related field of educational psychology about multimedia learning. I outline our solution for user interface and visualization comparing it to these results. To provide an easy to use and quick to learn system, the design of it should be consistent in object, motion and control interface (Gloor, 1998). Consistency of object and motion are related to visualizations that are discussed in Section 7.2 and the control interface consistency in Section 7.1.

7.1 User Interface of Jeliot 3

The design of the user interface is a crucial aspect of a tool, especially for a novice computer user. The system should help the user to master the system in a way that the interface between the user and the computer would disappear (Shneiderman, 1997). This could mean consistency of the user control in a way that is already known to the user (Gloor, 1998). To provide the consistency we are using a VCR controller metaphor for the visualization control and a text editor like design for the code editor.

We used the user interface design similar to Jeliot 2000 as it was found usable and simple for novices (Ben-Bassat Levy et al., 2003). The user interface layout of Jeliot 3 and its actual appearance are illustrated in Figures 27 and 28.

The window is divided first vertically into two panes. The upper pane is again divide into two panes. On the upper left pane is the code editor or code view depending on whether user is in editing or visualization mode. The top of the code editor contains buttons for simple editing operations. Moreover, line numbering is added to the code editor and code view to make the referencing to the code easier.

The upper right pane contains the visualization panel, called theater. In this panel the animation of the program is shown. If an error occurs during the visualization, the theater is replaced by an error viewer explaining the error.

The lower part of the window contains a control panel on the left and an output console on the right. The control panel contains buttons for changing the mode from editing

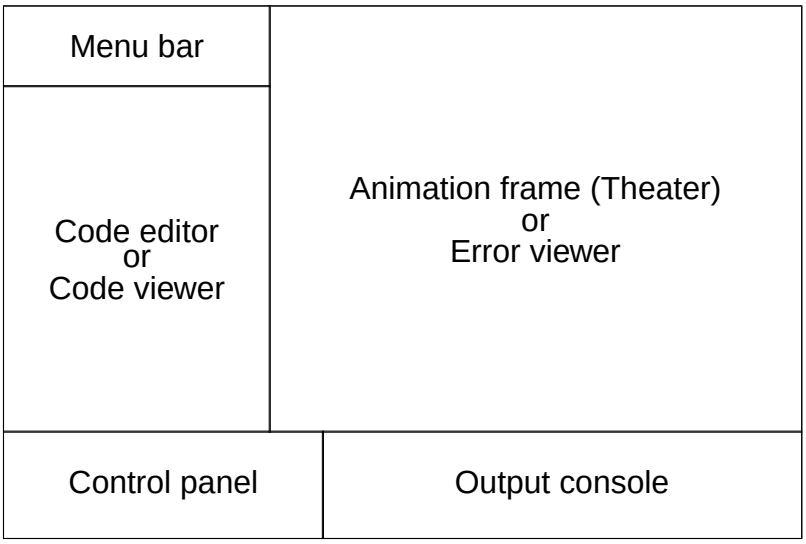

Figure 27: The structure of user interface in Jeliot 3.

to visualization and vice versa. The panel includes also buttons for controlling the visualization, for example, pausing, rewinding, playing or changing the speed of the visualization. The output console collects the output of the program just like command prompt does.

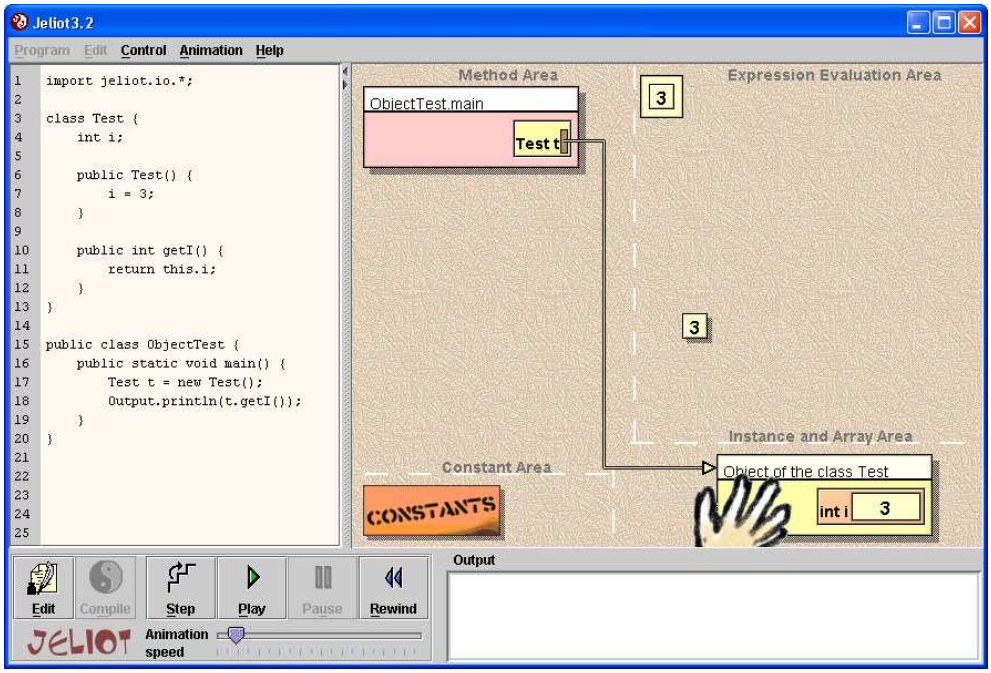

Figure 28: The user interface of Jeliot 3.

There are also menus on the top of the window. So all commands can be so done through menus. In the menu, there are also some additional commands that can help an expert user, such as the “run until...” feature. Shortcut keys are added to the menus to make the use of the software smoother, especially during lecture use.

7.2 Visualization

The visualization in Jeliot 3 was designed with three principles in mind, namely completeness, continuity and consistency. These principles come mainly from the research with previous versions of Jeliot and other research clarifying the effects of software visualization and other kinds of multimedia for learning. First, I introduce the theoretical findings about visualizations in general and then the domain specific results for software visualization. I conclude this section with the reasoning about the program visualization in Jeliot 3.

7.2.1 Theory about Visualizations

Only a little research has been done in the field of software visualization about the visualization design and the principles for the design. However, there is a growing base of knowledge of *multimedia learning* that deals with the design issues of multimedia. In broad sense, software visualization is also a kind of multimedia and the design principles for software visualization can be derived from the research in this area.

Multimedia learning is facilitated with *multimedia instructional messages* that contain for example pictures, animations and both spoken and written text. Mayer (2001) has researched the multimedia learning with a number of experiments and formed a theory of the basic requirements for it.

The theory of Mayer (2001) is based on three assumptions about the human cognition:

1. *Dual channels*. Humans have separate channels for visual and auditory information processing.
2. *Limited capacity*. Each of these channels is limited in capacity of processing.

3. *Active processing.* Active and meaningful learning happens by attending to relevant incoming information, by organizing selected parts of it into coherent mental representations and by integrating the mental representations with previous knowledge.

According to these assumptions and a series of experiments, Mayer (2001) has found seven principles for multimedia instructional messages that should be taken into consideration when multimedia for meaningful learning is designed. In this context, meaningful learning is defined as good results in retention (memorization) and transfer (applying) tests.

1. *Multimedia principle*: Words should be accompanied with pictures.
2. *Spatial contiguity principle*: Corresponding words and pictures should be as close to each other as possible.
3. *Temporal contiguity principle*: Corresponding words and pictures should be presented simultaneously.
4. *Coherence principle*: Extraneous words, pictures and sounds should be excluded.
5. *Modality principle*: Better learning takes place by using animation and narration rather than by using animation and on-screen texts.
6. *Redundancy principle*: Better learning takes place by using animation and narration rather than by using animation, narration and on-screen texts.
7. *Individual differences principle*: Design effects are stronger for low-knowledge learners than for high-knowledge learners and for high-spatial learners than for low-spatial learners.

Furthermore, Mayer (2001) also requires for *a structure for multimedia*. This means that multimedia should be a series of pictures or an animation that is accompanied by verbal explanations with a certain order, for example, the material could be ordered as a cause-and-effect chain. Finally, two other principles are found that help the students to perform better in transfer tests but not in retention tests. Firstly, the messages should be given in a personalized manner to help learning, *a personalization principle*

(Moreno and Mayer, 2000). Secondly, students should have control over the speed of the multimedia presentation they are viewing, *an interactivity principle* (Mayer and Chandler, 2001). Gloor (1998) presents similar kinds of principles but they are not based on extensive research but on implementation experience, thus these principles are also intuitive.

These principles can also be applied into software visualization and can form the basis for the visualization design. However, those principles that require spoken explanations are harder to be implemented, and it is especially hard to make them automatic.

In the field of software visualization, the difficulties in reading the visual displays is studied by Petre (1995). She found that there is a difference in the readership skill between novices and experts. This difference was characterized by the skill of reading the *secondary notation* encoded inside the displays. The secondary notation can be related to the layout, coloring or any implicit property of the display. The experts are able to use this information when solving problems, but novices seem to miss this extra information and thus fail in the problem solving. Similar differences between experts and novices are also found in research on multimedia learning by (Lowe, 2003; Kozma, 2003). Glaser and Chi (1988) state that novices concentrate on the surface structures, whereas experts tend to get deeper in the subject. These results indicate three issues that should be taken into consideration when designing visual displays. Firstly, novices need different kinds of visualizations compared to experts. Secondly, the visualizations for novices should be as consistent as possible and also make all the implicit secondary notation as explicit as possible. Finally, novices must still be taught how to read the visualization display. These issues are further discussed in Section 7.2.2.

One purpose for the use of visualizations is the formation of mental models. A *mental model* is a cognitive structure that can be used to represent real-world artifacts or phenomena (Ben-Ari, 2001b). The mental model can be either viable or non-viable depending on whether it really corresponds to the real-world behavior of the artifact or the phenomenon or not. The formation of the mental models is not clear but it seems that it contains visual and dynamic components and is connected to mental imagery. Nevertheless, an accurate visualization is probably one of the ways to facilitate the formation of a viable mental model.

7.2.2 Program Visualization in Jeliot 3

In the visualizations of Jeliot 3, we wanted to consider the visualizations of the previous versions and the theory presented above. The theory of visualization also influenced the design of Jeliot 2000, and some of the design principles are derived directly from its research and evaluation (Ben-Bassat Levy et al., 2003).

In the visualization, we wanted to be as consistent as possible to reduce the cognitive load of the student (Tudoreanu, 2003). With consistency we mean that each of the actions on the screen should happen in a similar way every time. For example, the expressions are evaluated in a same manner whether the values used in the evaluations come from the variables of the method or the object or whether they are constants.

Furthermore, we wanted to help the student to see the secondary notation by grouping all the similar kinds of language constructs (e.g. objects or constants) (Petre, 1995). All the visualized components have their own area on the screen and they always appear in that area. The layout of the visualization is shown in Figure 29.

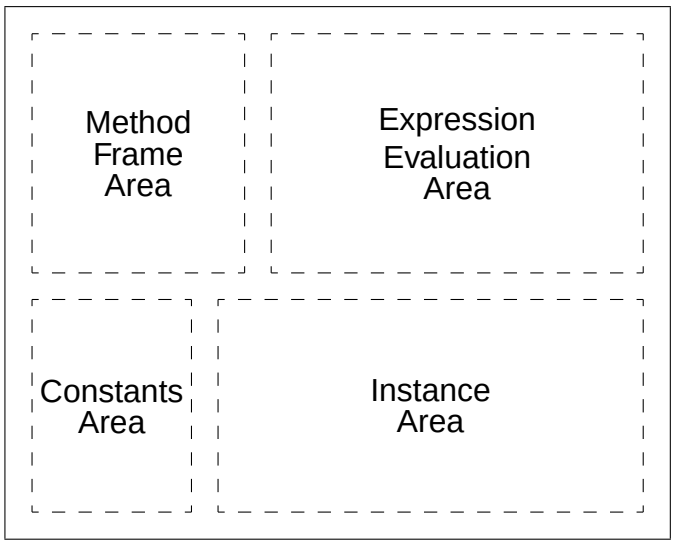

Figure 29: The structure of the animation frame (theatre) in Jeliot 3.

In the current version, we go further by explicitly showing the different areas to the user with dashed lines and titles. This can help the student to see the differences, for example, between a method call and method frame. However, at the same time it can make the comprehension of the visualization harder if the transitions between the areas are not clear to the user. For instance, it can be unclear for a novice when a method

call becomes a method frame or a constant becomes an expression and what is the difference between them. Thus, the transitions between the different areas should be paid more attention as they actually carry more meaning than the appearing on a certain area. Now the transitions are visible but their meaning is not and it can cause problems in the understanding of the animation.

The consistency of the visualization also means that the visualizations are formed as close as possible to the Java Language Specification (Gosling et al., 2000) as far as it has been pedagogically reasonable. This can help the student to map the executed code line to the current action in the visualization and see the cause-and-effect chain.

The design principles of multimedia learning were also used in the design (Mayer, 2001). Table 2 shows the implementation and relevance of the principles in Jeliot 3. We show the highlighted source code, the message texts and program animations simultaneously as they appear in the program. The messages are shown as close as possible to the corresponding values. To reduce the amount of information, the animation is shown in a sequential fashion, and one action at the time is visualized to reduce the cognitive load of the viewer. The interactivity principle is already taken into consideration in the user interface design with the VCR-like control panel. The modality and redundancy principles are not applied at the moment as we are not using any spoken messages. The personalization principle was not taken into consideration in the current design but could be implemented in the coming versions.

Table 2: The principles of multimedia learning and their relevance in Jeliot 3.

Principle of multimedia learning	**Implementation of the principle**	**Relevance of the principle**
Multimedia	Highlighted source code, message texts and animation of the program	The starting point for the visualization
Spatial contiguity	The messages are shown as close to the related values as possible	This reduces the eye movement during the visualization
Temporal contiguity	The messages are shown during the visualization simultaneously with the related values	This is important for the formation of the correct cause-and-effect chain or mental model

Table 2: The principles of multimedia learning and their relevance in Jeliot 3 (continued).

Principle of multimedia learning	**Implementation of the principle**	**Relevance of the principle**
Coherence	The animations are run sequentially one animation at the time	The area for the animation is limited and several moving visual objects could clutter the display and confuse the user
Interactivity	The VCR-like control buttons	Easy control of the program visualization is important for novices
Structure for multimedia	Implemented with the completeness, the continuity and the consistency of the visualization	Important for creation of the correct cause-and-effect chain and mental model
Modality	Not implemented	Should be tested in this context
Redundancy	Not implemented	Should be tested in this context
Personalization	Not implemented	Should be tested in this context
Individual differences	Not implemented	Support for diverse learners could be provided with different visualizations and textual representations

The recommendation of Mayer (2001) for the structure of the multimedia was found also meaningful in this context. It was addressed with three design principles: completeness, continuity and consistency. The first two concepts are discussed already in Section 3.3.2 in connection with Jeliot 2000. The consistency is discussed already in this section.

Jeliot 3 visualizes a large subset of novice-level Java programs. It visualizes the different aspects of data flow: values of primitive and reference type, variables in methods and in instances, one-dimensional arrays as well as method and object frames. It also

visualizes the control flow of the program by animating: static and object method calls (including recursion), expression evaluation of almost all binary and unary expressions, loop statements and conditional statements. To support the visualization of the control structures it also shows some explaining messages in the visualization. Jeliot 3 contains only one kind of visualization for each data object and thus the visualizations can not be adjusted by the user.

The code is visualized in the level of expressions and statements in a similar way as in Jeliot 2000. The code view also contains line numbering for easy referencing to the source code. As Jeliot 3 uses an interpreter as the basis of the system, even partially erroneous code can be visualized. If an error occurs during the visualization, the error view is shown. The error messages generated by the interpreter are shown to the user. If the error message indicates a certain place in the code, the source code is highlighted around that area.

8 Discussion

In this thesis, I have discussed different issues related to the design of Jeliot 3. In this chapter I try to evaluate those decisions in the perspective of the whole system, its possibilities and deficiencies. I also compare different versions of Jeliot to each other and the implemented version, Jeliot 3, to the requirements exposed in the literature. I conclude this chapter with future work where I introduce the possible new directions in which Jeliot 3 can be taken.

8.1 Discussion of the Design

In Jeliot 3, Extensibility was risen as one of the key design issues. This issue was addressed with two design decisions. Firstly, the modular structure of the system was designed in a way that gave us the possibility to use two separate systems. We decided to use DynamicJava, a Java interpreter, and the visualization engine and user interface of Jeliot 2000. Using a ready-made interpreter eases the internal extensibility because only small changes to the interpreter are needed to extract the intermediate representation when a new language construct is visualized if it can be handled with the current intermediate language. The addition of new graphical constructs into visualization engine can be made by extending one of the actor classes. The interpreter based design also allows a possibility to execute partially erroneous code because the interpreter can also execute source code that has semantic errors. In this way, the visualizations can show the errors to the user, hopefully in a more understandable way.

Secondly, the designed intermediate language is on a high enough abstraction level to enable the formation of different visualizations of it. For example, a method call tree or a diagram showing the amount of different operations performed could be generated during the visualization from the same intermediate presentation. The intermediate language is also extensible in a sense that new additions to it only require to the adding of the interpretation of the new command into the intermediate code interpreter. The rest of the intermediate code interpreter should stay the same.

The intermediate presentation also has another positive feature. The possibility to save the intermediate presentation after the first visualization makes the integration into the other systems possible. For example, these kinds of visualization presentations could

be posted on a discussion forum to facilitate the conversations on programming. All the communicators could view exactly the same visualization and thus the context of the conversation could be created through it. This issue will be discussed further in Section 8.3.

Another requirement for Jeliot 3 was that the system should support the visualization of as large a subset of programs written in Java language as possible. This was also realized through the design and using of Java language interpreter, DynamicJava. Although at the moment, several language features are supported already, there is still much work to be done in this area. The support for inheritance is still insufficient and static variables or multidimensional arrays are not supported. In addition, Java API methods are not supported if they return a reference type. This is due to the fact that at the moment there are no means to gather enough information on the references for visualization. However, the language coverage at the moment is still sufficient for most concepts taught in the introductory programming course with real novices for whom this tool was created.

The price of all these positive features is the complex design and implementation. Jeliot 3 consist of two systems, Java source interpreter and a visualization engine, containing several thousands source lines of code. The software also includes the interface between these two systems that contains a number of commands. This means that the further development of the systems will not be easy in all cases, and that the comprehension of the system is a demanding task. We try to ease this work with documentation that is still in preparation but will be published soon.

8.2 Comparison

Jeliot 3 is the fourth member of the Jeliot family. The other three members, Eliot, Jeliot I and Jeliot 2000, were introduced in Chapter 3. There are differences between that versions. In this section, I compare these versions to each other and explain the reasons for different approaches. I also compare Jeliot 3 to the pedagogical requirements of the software visualization tool proposed in the literature.

8.2.1 Differences between Jeliot Versions

The properties that different versions of Jeliot family have are shown in Table 3. At first, it is worth mentioning that Eliot and Jeliot I were developed more as a general purpose tool than just as tool to target specific user group, whereas Jeliot 2000 and Jeliot 3 were developed especially for real novice programmers. This difference explains especially the user interface, the scope of the visual objects and the control flow visualization. This also means that Eliot and Jeliot I can be used as algorithm animation tools and the user can just select certain visual objects and change their visual appearance. The control flow is not as important as the state of the data structures. On the contrary, the aim of Jeliot 2000 and Jeliot 3 is to provide as complete a visualization of the programs as possible, and explain all the aspects of the program as thoroughly as possible for novice programmers. This explains why Jeliot 2000 and Jeliot 3 are similar in the selection of visual objects and control flow visualization.

This division also applies to the other features. Eliot and Jeliot I visualize their own dialect of a common programming language. Eliot has Eliot-C, dialect of C, and Jeliot I has EJava, dialect of Java. Jeliot 2000 and Jeliot 3 visualize a subset of Java, but the only difference is that Jeliot 3 visualizes a larger subset of the programs. The aspect of code visualization is similar in Eliot and Jeliot I, the code visualization was not taken into consideration. In Eliot, the code is not even shown during the visualization and in Jeliot I, the code is shown and only the first line containing the evaluated statement or expression is highlighted no matter how many lines it consist of. Jeliot 2000 and Jeliot 3 highlight the currently evaluated area. It can be either a statement, an expression or any other Java construct. Furthermore, Jeliot 3 shows the line numbers of the source code to ease the referencing to the source code lines.

Table 3: Comparison between the different versions of Jeliot family.

	Eliot	**Jeliot I**	**Jeliot 2000**	**Jeliot 3**
Language	Eliot-C	EJava	Small subset of Java	Large subset of Java
Visual objects selectable	+	+	-	-
Visual objects' appearance adjustable	+	+	-	-

Table 3: Comparison between the different versions of Jeliot family (continued).

	Eliot	**Jeliot I**	**Jeliot 2000**	**Jeliot 3**
Possibility to add new visual objects	-	+	-	-
Active code area highlighted	-	the whole starting code line of the statement or expression	statements and expressions	statements and expressions
Line numbering	-	-	-	+
Run until line feature	-	+	-	+
Error messages	-	not enough informative	not enough informative	more informative
Number of stages	no limit	no limit	1	1
Number of windows	4 or more	4 or more	1	1
Visualized data types				
Numbers	+	+	+	+
Boolean	-	+	+	+
Character	+	+	+	+
String	-	-	-	+
1-dimensional Array	+	+	+	+
2-dimensional Array	+	+	-	-
Queue	-	+	-	-
Stack	-	+	-	-
Tree	+	-	-	-
Object	-	-	-	+
Control flow visualization				
Conditional statements	partial	partial	+	+
Loops	partial	partial	+	+
Static method calls	-	-	+	+
Object method calls	-	-	-	+
Recursion	partial	partial	+	+

Visualizing the object-oriented concepts is the issue that differentiates Jeliot 3 from all the other systems. Jeliot 3 visualizes objects, object variables and object method calls. Currently, inheritance is partially supported but will be fully supported in next releases. Another, issue that is taken into account in Jeliot 3 is the error message which should be more informative. Furthermore, even partially erroneous code can be visualized because of the interpreter based design.

8.2.2 Requirements for Program Visualization Systems

Different kinds of recommendations for the algorithm visualization systems have been stated in the literature (Rößling and Naps, 2002; Naps et al., 2003b). These requirements can be also stated for a program visualization system. Some of the requirements are from the pedagogical point of view and others from the utilization point of view. According to the recommendations, the program visualization system should:

1. reliably reach as large a target audience as possible;
2. be as general-purpose system as possible;
3. allow users to provide input data for the algorithm or the program;
4. show structural view of the algorithm or the program;
5. show smooth animation;
6. allow interactive predictions;
7. allow users to rewind the visualizations;
8. give hypertext explanations of the visual display;
9. be integrable with a database for course management;
10. map to existing teaching and learning resources;
11. be flexible for different kinds of utilizations;
12. provide comprehensive and integrated support;
13. have a supporting web site;

14. be registered to the tool repositories; and

15. be publicized in other forums.

Jeliot 3 supports eight of the recommendations fully (1, 2, 3, 5, 11, 13, 14 and 15) and three of them partially (6, 10 and 12). Four of the requirements are not currently supported (4, 7, 8 and 9). Jeliot 3 supports multiple platforms as it is written in Java and does not require any platform specific programs or information. It also uses resource files for all the images, texts and fonts thus allowing translation to other languages as well as changing images or fonts for a specific culture context.

Jeliot 3 can visualize a wide variety of Java programs, especially supporting programs that are important for novice programmers. This makes Jeliot 3 be as general-purpose a system as is possible in the context of novice programmers.

Jeliot 3 visualizes the programs with smooth animations that allow the user to input data during the visualization. The animation can be stopped at any time to pose a question by the teacher but Jeliot 3 does not currently support automatic predictive questions making the support partial.

Jeliot 3 is missing the capability to rewind the animation by single steps. It does not contain a structural view of the program that allows jumping backward or forward into a specific state in the program (e.g. method call or variable declaration). However, it contains a "run until" -feature that allows running the program until a specific line is executed.

Currently, there is no hypertext explanations for the visual display in Jeliot 3. However, the intermediate language could be used to form an explanation for the program execution. The database integration is not possible at the moment but can also be added.

Jeliot 3 is not used by any text book or any available learning material as the demonstration tool. However, Jeliot 3 can be used in class room settings as a demonstration aid by the teacher, as well as in active laboratory sessions by the students. The support for a large subset of Java programs gives the possibility to adapt Jeliot 3 with several novice-level text books or learning materials for Java programming.

A user guide and some examples are provided with Jeliot 3 distribution. However, the number of examples could be much higher and they could provide more support for novices to understand the different visual objects of Jeliot 3.

We have built up a web site to provide support for and information on Jeliot 3 and the previous versions of Jeliot (Moreno and Myller, 2003a). We have also registered Jeliot 3 to ACM SIGCSE web site (SIGCSE, 2004) and we have plans to add it to other repositories as well. We have also demonstrated the system in one conference (Moreno and Myller, 2003c) and more publications on Jeliot 3 are currently in preparation.

8.3 Future Work

After the implementation of the system is finished, a further development of some ideas should be started. In this section, I introduce some further development areas that were identified during the development process.

The most obvious direction for the future work is the addition of new visualization features. There are still several Java language constructs that are used on novice level but are not visualized in Jeliot 3. At least a comprehensive support for inheritance and support for static variables should added but other visualization such as multidimensional arrays should be considered.

Jeliot 3 contains only one visualization of the program even though the capabilities of the intermediate code allow several different kinds of visualizations to be generated. These could include, for example, a method call tree view, operations count view and object state diagram. New visualization displays should be easy to add into Jeliot 3. A plug-in mechanism could help this issue allowing extensions of the system.

As discussed already in Section 7.2.1, the learning is aided if the visualization is accompanied with either textual or spoken explanations (Mayer, 2001). The explanations and self-evaluation were also found important in the research of Naps et al. (2000). Verbal explanation support could be added to Jeliot 3 in two ways. The intermediate code could be used to generate low-level explanations about the execution, for example, statement `x = x + 1;` could be explained with a sentence: “A value of the expression `x + 1`, which is `4`, is assigned to the variable `x`.”, if `x` had a value `3` before the assignment. We could also create a special source code comment type that could be identified as a message to be visualized. For instance, when this kind of a comment is inserted into the source code line, we could visualize the message of the comment every time the corresponding line is executed. This design would allow the possibility for stop-and-think questions to support active prediction of the program execution.

The adaptability and adaptiveness of the visualizations should also be taken into consideration. *Adaptive program visualization* (Brusilovsky and Su, 2002) takes into account the skills of the user and adapts the visualization according to the skill levels. For example, some of the visualization steps could be emphasized or taken out depending on the skill level of the user. This would require collection of information about the student and student modeling. This would allow adapted visualization styles for different levels of knowledge.

Research done with visualization of programs by programmers and the visualizations as demonstration aids has shown that several visualizations exist for the same language construct (Ford, 1993; Lattu et al., 2003). Currently, Jeliot 3 provides only one kind of visualization for each language construct. However, the number of visualizations could be increased by programming several actors for same language construct and letting the user to select a visualization she wants to use. This would be an option that is not shown to the novices but provided only for more advanced students through menus.

The research in psychology of programming has introduced a new concept, roles of variables, and their use as visualization objects (Sajaniemi and Kuittinen, 2003). The roles of variables make explicit the tacit expert knowledge that can be taught to novices and ten roles are enough to describe almost all novice-level programs. Sajaniemi and Kuittinen (2003) has also implemented a series of program visualizations of Pascal programs illustrating the roles of variables. The roles of variables could also be visualized in Java programs by Jeliot 3. Only two main modifications would be needed. Firstly, the roles of each variable should be detected. Currently, there is no possibility to detect the roles automatically. Thus, source code annotations as special comments should be used to explain the visualization engine the roles of each variable. However, the intermediate code could also be used to detect the roles of each variable with some heuristics. Secondly, the assignment for each role should be animated differently which requires reprogramming of the variable animations. Moreover, currently the roles are only defined for novice-level procedural programming and it is not clear whether the roles are sufficient for novice-level object-oriented programs.

Although the reverse execution of the visualization has been found important in the evaluations, there currently exists only a small number of software visualization systems that can do it (Rößling and Naps, 2002). This feature would increase the pedagogical value of the tool as students could rewind the visualization a few steps and

start the viewing again when lost in the visualization. This feature can not be added to Jeliot 3 in a way described by Boroni et al. (1996) and Crescenzi et al. (2000) because the current implementation of Jeliot 3 uses an interpreter and not a virtual machine as the basis. However, it could be done in the level of animation by using serialization of the animation data into an undo structure.

Currently, there are only a few, if any, tools to support collaborative programming. The current collaboration environments do not support features that are needed to discuss a program and its behavior. A *collaborative program visualization system* could be a tool for supporting programming in collaboration. This kind of a tool could also facilitate learning as a group work tool for programming courses. In collaborative program visualization system, the user can code a program, visualize it and post the source code and the visualization of the program on the platform for other users. The other users can then view the visualization of the program, discuss it and look for better solutions or possible improvements in the coded algorithm or learn from it.

Woven Stories 2 (Gerdt et al., 2001) is a learning environment that is based on the concept of interleaving texts that can be connected to each other. This means that the learning environment consists of a graph whose nodes are the learning objects (e.g. HTML-pages or plain text) and the edges are the relations between them. In this kind of a learning environment, the user can add new nodes and connections between nodes that are shown to all users in real-time.

By combining Woven Stories 2 and Jeliot 3, we could form an environment where users could share their visualizations of programs and discuss them as mentioned above. The development of an algorithm and its visualization could be illustrated as a path or paths in the graph of descriptions, visualizations and comments related to the algorithm. The environment could facilitate collaboration because many users could simultaneously watch the same animation and make their own modifications to the programs and animations available to other users. Moreover, the environment could support learning because the users can form different kinds of learning paths through these networks of texts, algorithms and their visualizations.

All the previous propositions are linked to the new technical developments of Jeliot 3 but also empirical evaluation of the tool is needed. The empirical evaluation should address three different issues: the use of the tool individually in isolation (e.g. in distance education), the use of the tool in class rooms (e.g. lectures and laboratory sessions) and

the use of the tool in collaboration as in collaborative programming. The important issues to find are those that could be identified as needed in these different contexts. This would require qualitative research whose findings could be then specified with the quantitative research.

9 Conclusions

As found in the previous research, the software visualization tool must cognitively engage the student, to enhance the learning of the subject in hand. Furthermore, the visualization must be designed according to certain principles and it should be suitable for the level of knowledge of the particular student. Although these are hard requirements to fulfill, Jeliot 3, the new member of the Jeliot family, is trying to answer all of them as well as possible.

The history of the Jeliot family is more than ten years long. Three other systems have been implemented before Jeliot 3, namely Eliot, Jeliot I and Jeliot 2000. All these versions have had their own objectives to be achieved. The history also indicated the direction for this new version, Jeliot 3, because it was found in the previous study with Jeliot 2000 that novice students learning programming benefit from the program visualization. However, the design of Jeliot 2000 was inappropriate for further development and thus development of a new version was required.

Jeliot 3 is ready to be used at introductory programming courses by teachers and students. Jeliot 3 can help in the early stages of the courses by providing clear semantics and by engaging students into the learning process. With Jeliot 3 visualizations, teachers and students can share a graphical and a verbal vocabulary that eases the discussion of the programming concepts (Ben-Bassat Levy et al., 2003). Jeliot 3 has improved features of the previous versions of the Jeliot family and incorporated a few new features, for example, support for object-oriented concepts visualization, acceptance of a larger subset of programs, improved error messaging, improved design and better extensibility.

The visualization of the system is designed according to the principles of multimedia learning (Mayer, 2001) and other research related to the graphical displays (Petre, 1995). This gives the visualization a more theoretical basis. From this research and evaluation of Jeliot 2000, three other principles were identified; the visualization of the programs should be complete, continuous and consistent. However, these principles should also be validated with empirical evaluations.

The requirements for the new version of Jeliot that were stated in Section 1.2 are also met in Jeliot 3. The system is easy to use as it uses the user interface that was already found relatively easy to use by novices (Ben-Bassat Levy et al., 2003). The visual-

ization is complete, continuous and consistent as much as possible. Jeliot 3 supports visualization of a large subset of novice-level Java programs. Furthermore, we have developed a new language called intermediate code. This language allows us to form new kinds of visualization and extend the system.

The framework on which Jeliot 3 is built on is a hybrid model of the previously designed program animation systems. It combines two systems DynamicJava, a Java source interpreter, and the visualization engine and the user interface of Jeliot 2000. These systems are combined with an intermediate language that allows formation of the different kinds of visualizations. Thus, Jeliot 3 also aspires to be the base for future developments. Maintaining it stable and documented will, hopefully, encourage developers to create new visualizations into Jeliot 3. Those new features will adapt Jeliot 3 to different contexts and can make it a more valuable tool for different user populations. The open development of Jeliot 3 will stand on the basis of public licensing with General Public License (GPL).

First evaluation of Jeliot 3 will be carried out during the spring term of the year 2004. Results of this evaluation will show us how to proceed in the further research and development. It will be seen in the future what kind of a tool is going to be developed around Jeliot 3.

References

AAPS-project, 1997. Jeliot I. WWW-page, http://www.cs.joensuu.fi/jeliot/jeliot.html (Accessed 2004-13-02).

Allen, E., Cartwright, R., Stoler, B., 2002. Drjava: a lightweight pedagogic environment for Java. SIGCSE Bulletin 34 (1), 137–141.

Baecker, R., 1981. Sorting out Sorting. Videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excerpted in ACM SIGGRAPH Video Review #7.

Barnes, D. J., Kölling, M., 2003. Objects First with Java — A Practical Introduction using BlueJ. Prentice Hall / Pearson Education, Harlow, England.

Ben-Ari, M., 2001a. Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching 20 (1), 45–73.

Ben-Ari, M., 2001b. Program visualization in theory and practice. Informatik / Informatique 8 (2), 8–11.

Ben-Ari, M., Myller, N., Sutinen, E., Tarhio, J., 2002a. Perspectives on Program Animation with Jeliot. In: Diehl, S. (Ed.), Software Visualization. Vol. 2269 of Lecture Notes in Computer Science. Springer-Verlag, pp. 31–45.

Ben-Ari, M., Noa Ragonis, Ronit Ben-Bassat Levy, 2002b. A Vision of Visualization in Teaching Object-Oriented Programming. In: Ben-Ari, M. (Ed.), Proceedings of the Second Program Visualization Workshop. University of Aarhus, HornstrupCentret, Denmark, pp. 83–89, http://stwww.weizmann.ac.il/g-cs/benari/pvw/pvw2002-pdf.zip (Accessed 2004-13-02).

Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P. A., 2003. The Jeliot 2000 program animation system. Computers & Education 40 (1), 15–21.

Bergin, J., Jimènez-Peris, R., Brodie, K., Patiño-Martínez, M., McNally, M., Naps, T., Rodger, S., Wilson, J., Goldweber, M., Khuri, S., 1996. An overview of visualization: its use and design. ACM SIGCSE Bulletin 28 (SI), 192–200.

Berk, E. J., Ananian, C. S., 2003. JLex: A Lexical Analyzer Generator for Java(TM). WWW-page, http://www.cs.princeton.edu/~appel/modern/java/JLex/ (Accessed 2004-13-02).

Birkheim, A., 2002. Automatic Visualization of Java Programs to Be Used in the Java Teaching. Master's thesis, University of Applied Sciences Cologne, Cologne, German, (In German).

BlueJ, 2003. BlueJ — The Interactive Java Environment. WWW-page, `http://www.bluej.org` (Accessed 2004-13-02).

Boroni, C. M., Eneboe, T. J., Goosey, F. W., Ross, J. A., Ross, R. J., 1996. Dancing with DynaLab: Endearing the Science of Computing to Students. SIGCSE Bulletin 28 (1), 135–139.

Brusilovsky, P., Su, H., 2002. Adaptive Visualization Component of a Distributed Web-Based Adaptive Educational System. In: Stefano A. Cerri and Guy Gouardères and Fábio Paraguaçu (Ed.), Intelligent Tutoring Systems. Vol. 2363 of Lecture Notes in Computer Science. Springer-Verlag, pp. 229–238.

Crescenzi, P., Demetrescu, C., Finocchi, I., Petreschi, R., 2000. Reversible execution and visualization of programs with LEONARDO. Journal of Visual Languages and Computing 11 (2), 125–150.

Demetrescu, C., Finocchi, I., Stasko, J. T., 2002. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In: Diehl, S. (Ed.), Software Visualization. Vol. 2269 of Lecture Notes in Computer Science. Springer-Verlag, pp. 16–30.

Domingue, J., Price, B. A., Eisenstadt, M., 1992. A Framework for Describing and Implementing Software Visualization Systems. In: Jaffe, N. (Ed.), Proceedings of Graphics Interface '92. Canadian Information Processing Society, Vancouver, British Columbia, Canada, pp. 53–60.

Ford, L., 1993. How programmers visualize programs. Research Report 271, Department of Computer Science, University of Exeter, Exeter, England.

Gerdt, P., Kommers, P., Looi, C.-K., Sutinen, E., 2001. Woven Stories as a Cognitive Tool. In: Cognitive Technology. Vol. 2117 of Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 233–247.

Glaser, R., Chi, M. T. H., 1988. What is it to be an expert? In: Chi, M. T. H., Glaser, R., Farr, M. J. (Eds.), The nature of expertise. Erlbaum & Associates, Hillsdale, New Jersey, USA, pp. xv–xxiix.

Gloor, P. A., 1998. User Interface Issues for Algorithm Animation. In: Stasko, J., Domingue, J., Brown, M. H., Price, B. A. (Eds.), Software Visualization — Programming as a Multimedia Experience. The MIT Press, Cambridge, Massachusetts, USA, pp. 145–152.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2000. The Java Language Specification, Second Edition. Sun Microsystems, Inc.

Grissom, S., McNally, M. F., Naps, T., 2003. Algorithm visualization in CS education: comparing levels of student engagement. In: Proceedings of the 2003 ACM symposium on Software visualization. ACM Press, pp. 87–94.

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

Hightower, R., 2000. BeanShell & DynamicJava: Java Scripting with Java. Java Developer's Journal 5 (7), 86–95, `http://www.sys-con.com/java/pdf/ejbs.pdf` (Accessed 2004-13-02).

Holland, S., Griffiths, R., Woodman, M., 1997. Avoiding object misconceptions. ACM SIGCSE Bulletin 29, 131–134.

Hudson, S. E., Flannery, F., Ananian, C. S., Wang, D., Appel, A. W., 1999. CUP Parser Generator for Java. WWW-page, `http://www.cs.princeton.edu/~appel/modern/java/CUP/` (Accessed 2004-13-02).

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

Koala Project, 2002. DynamicJava. WWW-page, `http://koala.ilog.fr/djava/` (Accessed 2004-13-02).

Kölling, M., Quig, B., Patterson, A., Rosenberg, J., 2003. The BlueJ system and its pedagogy. Journal of Computer Science Education 13 (4).

Korhonen, A., 2003. Algorithm Visualization and Simulation. Ph.D. thesis, Helsinki University of Technology.

Kozma, R., 2003. The material features of multiple representations and their cognitive and social affordances for science understanding. Learning and Instruction 13 (2), 205–226.

Lahtinen, S.-P., Lemström, K., Litola, J., Porttikivi, A., 1994. Salsa++ animator. Report C-1994-66, Department of Computer Science, University of Helsinki, Helsinki, Finland, (In Finnish).

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

Lattu, M., Meisalo, V., Tarhio, J., 2003. A visualization tool as a demonstration aid. Computers & Education 41 (2), 133–148.

Lattu, M., Tarhio, J., Meisalo, V., 2000. How a Visualization Tool Can Be Used - Evaluating a Tool in a Research & Development Project. In: 12th Workshop of the Psychology of Programming Interest Group. Corenza, Italy, pp. 19–32, `http://www.ppig.org/papers/12th-lattu.pdf` (Accessed 2004-13-02).

Lowe, R. K., 2003. Animation and learning: selective processing of information in dynamic graphics. Learning and Instruction 13 (2), 157–176.

Markkanen, J., Saariluoma, P., Sutinen, E., Tarhio, J., 1998. Visualization and imagery in teaching programming. In: Domingue, J., Mulholland, P. (Eds.), 10th Annual Meeting of the Psychology of Programming Interest Group. Knowledge Media Institute, Open University, Milton Keynes, UK, pp. 70–73.

Mayer, R. E., 2001. Multimedia Learning. Cambridge University Press, Cambridge, UK.

Mayer, R. E., Chandler, P., 2001. When Learning is Just a Click Away: Does Simple User Interaction Foster Deeper Understanding of Multimedia Messages? Journal of Educational Psychology 93 (2), 390–397.

Meisalo, V., Sutinen, E., Tarhio, J., Teräsvirta, T., 1998. Combining algorithmic and creative problem solving on the web. In: Davis, G. B. (Ed.), Proceedings of Teleteaching ’98 - Distance Learning, Training and Education, IFIP World Computer Congress 1998. Austrian Computer Society, pp. 715–724.

Moreno, A., 2004. Taxonomy of Intermediate Codes in Software Visualization and Its Application on Jeliot 3. Master's thesis, Department of Computer Science, University of Joensuu, Joensuu, Finland, (To be published).

Moreno, A., Myller, N., 2003a. Jeliot. WWW-page, `http://cs.joensuu.fi/jeliot/` (Accessed 2004-13-02).

Moreno, A., Myller, N., 2003b. Jeliot Program Animation System – Internal Documentation.

Moreno, A., Myller, N., 2003c. Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In: Proceedings of International Conference on Networked e-learning for European Universities. pp. (CD–ROM publication).

Moreno, R., Mayer, R. E., 2000. Engaging students in active learning: The case for personalized multimedia messages. Journal of Educational Psychology 92 (4), 724–733.

Myers, B. A., 1986. Visual Programming, Programming by example, and Program Visualisation: A Taxonomy. In: CHI '86: Human Factors in Computing Systems. pp. 59–66.

Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rößling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R. J., Anderson, J., Fleischer, R., Kuittinen, M., McNally, M., 2003b. Evaluating the educational impact of visualization. In: Working group reports from ITiCSE on Innovation and technology in computer science education. ACM Press, pp. 124–136.

Naps, T. L., Eagan, J. R., Norton, L. L., 2000. JHAVÉ — an environment to actively engage students in Web-based algorithm visualizations. SIGCSE Bulletin 32 (1), 109–113.

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

Niemeyer, P., 2003. BeanShell. WWW-page, `http://www.beanshell.org/` (Accessed 2004-13-02).

Oechsle, R., Schmitt, T., 2002. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In: Diehl, S. (Ed.), Software Visualization. Vol. 2269 of Lecture Notes in Computer Science. Springer-Verlag, pp. 176–190.

Petre, M., 1995. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. Communication of the ACM 38 (6), 55–70.

Petre, M., Green, T. R. G., 1993. Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display Is an Acquired Skill. Journal of Visual Languages and Computing 4 (1), 55–70.

Pierson, W. C., Rodger, S. H., 1998. Web-based animation of data structures using JAWAA. SIGCSE Bulletin 30 (1), 267–271.

Pratt, C. M., 1995. An OSF/Motif Program Animator for the DynaLab System. Master's thesis, Department of Computer Science, Montana State University, Bozeman, Montana, USA.

Price, B. A., Baecker, R. M., Small, I. S., 1993. A Principled Taxonomy of Software Visualization. Journal of Visual Languages & Computing 4 (3), 211–266.

Richter, M. W., 2000a. Iava: yet another interpreter for scripting within the Java platform. Software: Practice and Experience 30 (2), 81–106.

Richter, M. W., 2000b. The Iava Homepage. WWW-page, `http://members.tripod.com/mathias/IavaHomepage.html` (Accessed 2004-13-02).

Roman, G.-C., Cox, K. C., 1992. Program Visualization: The Art of Mapping Programs to Pictures. In: Proceedings of the 14th international conference on Software engineering. Melbourne, Australia, pp. 412–420.

Roman, G.-C., Cox, K. C., 1993. A Taxonomy of Program Visualisation Systems. Computer 26 (12), 11–24.

Roman, G.-C., Cox, K. C., Wilcox, D., Plun, J. Y., 1992. Pavane: a System for Declarative Visualization of Concurrent Computations. Journal of Visual Languages and Computing 3 (2), 161–193.

Rößling, G., Freisleben, B., 2002. Animal: A system for supporting multiple roles in algorithm animation. Journal of Visual Languages and Computing 13 (3), 341–354.

Rößling, G., Naps, T. L., 2002. A testbed for pedagogical requirements in algorithm visualizations. SIGCSE Bulletin 34 (3), 96–100.

Sajaniemi, J., Kuittinen, M., 2003. Program animation based on the roles of variables. In: Proceedings of the 2003 ACM symposium on Software visualization. ACM Press, pp. 7–16.

Shneiderman, B., 1997. Designing the User Interface: Strategies for Effective Human-computer Interaction. Addison-Wesley, Reading, Massachusetts, USA.

SIGCSE, A., 2004. Sigcse: Education links. WWW-page, `http://www.sigcse.org/topics/` (Accessed 2004-13-02).

Smith, P. A., Webb, G. I., 2000. The Efficacy of a Low-Level Program Visualisation Tool for Teaching Programming Concepts to Novice C Programmers. Journal of Educational Computing Research 22 (2), 187–215.

Stasko, J. T., Kraemer, E., 1993. A methodology for building application-specific visualizations of parallel programs. Journal of Parallel and Distributed Computing 18 (2), 258–264.

Stoler, B. R., 2002. A Framework for Building Pedagogic Java Programming Environments. Master's thesis, Rice University, Houston, Texas, USA.

Stratton, D., 2001. A Program Visualisation Meta-Language Proposal. In: Lee, C. (Ed.), Proceedings of the 9th International Conference on Computers in Education ICCE/SchoolNet2001. Soeul, South Korea, pp. 601–609.

Stratton, D., 2003. Program Visualization Meta Language. Ph.D. thesis, School of Information Technology & Mathematical Sciences, University of Ballarat.

Sutinen, E., Tarhio, J., 1993. String matching animator SALSA. In: Mati Tombak (Ed.), Third Symposium on Programming Languages and Software Tools. University of Tartu, pp. 120–129.

Sutinen, E., Tarhio, J., Lahtinen, S.-P., Tuovinen, A.-P., Rautama, E., Meisalo, V., 1997. Eliot – an Algorithm Animation Environment. Report A-1997-4, Department of Computer Science, University of Helsinki, Helsinki, Finland, `http://www.cs.helsinki.fi/TR/A-1997/4/A-1997-4.ps.gz` (Accessed 2004-13-02).

Sutinen, E., Tarhio, J., Teräsvirta, T., 2003. Easy Algorithm Animation on the Web. Multimedia Tools and Applications 19 (2), 179–184.

Tudoreanu, M. E., 2003. Designing effective program visualization tools for reducing user's cognitive effort. In: Proceedings of the 2003 ACM symposium on Software visualization. ACM Press, pp. 105–114.