Interactive visualization of the execution of object-oriented programs

Luijten, C.A.A.M.

Award date:
2009

Eindhoven University of Technology

Department of Mathematics and Computer Science

Master's Thesis

Interactive Visualization of the Execution of Object-Oriented Programs

by

C.A.A.M. Luijten

Supervisors: Dr. C. Huizing and Dr. R. Kuiper

Eindhoven, June 19, 2009

Abstract

There is a need for a visual tool for describing the execution of object oriented programs in an educational setting. A literature study for existing software visualizations is carried out to identify requirements to such a system.

A Visual Object and Execution Model based on the Contour Model of Block-Structured Processes is used to visualize program execution state. Additionally, rule-based selection methods are used to remove the less important parts of this visualization from view.

We have implemented our approach in a prototypical tool called CoffeeDregs. Preliminary experiments with this tool have shown that the tool can be used for a larger pedagogic experiment on freshmen students following a programming course.

Preface and Acknowlegdements

Last June I went to Huub van de Wetering, who is the internship and graduation coordinator of the Visualization group to ask whether he had any openings or ideas for a graduation project. Among the proposals was the CoffeeDregs project, which drew my attention immediately and pushed the other ideas aside.

Kees Huizing, Ruurd Kuiper and I sat down to talk things through and after letting it sink in during the weekend, I decided to jump right in. After a period of roughly two months doing literature study, I got a clearer picture of what is going on in the field of educational software visualization and began forming a vision of how CoffeeDregs could contribute to that. I decided to focus on certain aspects where CoffeeDregs would stand apart from the "competition".

Most of all, I would like to thank my supervisors Kees Huizing and Ruurd Kuiper for their useful advice, feedback, encouragement and guidance during the project. Kees and Ruurd did not always agree with each other, which turned out to be a fertile ground for healthy discussions with sometimes a clear conclusion.

Thanks also to Vincent Vandalon who worked on the backend of CoffeeDregs to improve the communication with the Java Debugging Interface. I thank Guy Dubois and Maarten Pennings for offering their cooperation with the user experiments and Rob Nederpelt for letting me use his office for these experiments.

I am grateful to my parents, friends and family for their years of encouragement in all sorts of ways. Finally, I would like to thank my girlfriend, Marly, who has been a source of support throughout my graduation project.

Thank you all.

Christian Luijten

Eindhoven, June 19, 2009

Contents

List of Acronyms

JDI Java Debug Interface

VM Virtual Machine

OOP Object Oriented Programming

VOS Visual Operational Semantics

CM Contour Model of Block-Structured Processes

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Much effort has been put into Software Visualization, the visualization of anything related to computer software. Most research is predominantly aimed at aiding the professional programmer in his/her workflow, leaving the beginning programmer in the dark.

The aim of the Master project associated with this thesis is to provide an educationally useful software visualization tool to be used during a programming course on Java.

In this thesis we explore and compare various tools with similar aims, in order to better understand those aims and to later be able to compare them with our own tool. We also assess a type of visualization model that uses the scope of program code for its structure, which results in a new idea for the visualization of the execution of programs.

To limit the overwhelming amount of objects visualized in larger programs, we must assess what objects are important to show and what not. We also decide whether a user should have influence on this process.

The ideas proposed are implemented in a software visualization tool called CoffeeDregs and then tested in a preliminary experiment with students.

The result is a new stable student version, ready to be tested in lecture situation.

1.1 Thesis outline

In this thesis we present a model for an educational software visualization. We first define the terms used in the thesis and propose a didactic vision for the teaching of Object Oriented Programming (OOP) in Chapter 2. We then discuss the existing work in the field and set it against our didactic vision in Chapter 3 and come to the conclusion that there is the need for a more specific solution.

A literature study for execution models in Chapter 4 is followed by a discussion about the specific solution we have in mind. This discussion is divided into a part about *how* objects are visualized in Chapter 5 and a part about *what*

objects are visualized in Chapter 6.

The software tool "CoffeeDregs" is introduced in Chapter 7. This tool is based on the observations made during the research phase and fits our didactic vision. After implementation, an experiment was carried out among students to identify flaws or weaknesses in CoffeeDregs that could influence the pedagogic experiments in the future. The purpose of these experiments is to address those issues *before* introducing the tool in the educational programme. The report of these experiments is found in Chapter 8.

Finally, Chapter 9 concludes the thesis with open issues and advice for future research and development.

1.2 Definition of terms

This section introduces the terminology used in the rest of the thesis.

1.2.1 Software visualization

A software visualization is a system which aims to visualize certain aspects of another software system. It can be a visualization of any abstraction level of the software (e.g. from a view of the electrons in a computer to the visualization of complex high-level datastructures).

This thesis, for example, describes a software visualization at a fairly high abstraction level of objects and methods.

1.2.2 Reverse stepping vs. Reverse execution

Reverse stepping and reverse execution enable the user to track back the execution by 'undoing' steps in the execution. The difference between reverse stepping and reverse execution is whether the Virtual Machine (VM) actually inverts the last execution step.

In reverse stepping, the system holds a collection of program states which can be visualized at any point during the execution. If the user steps along these program states in reverse order of execution, it appears as if the program itself is actually running backward. In reverse execution however, each program instruction is actually inverted and then executed. The resulting program state is then the actual state of the running program.

Reverse stepping is usually much easier to implement than reverse execution, but limits the possibilities of 'changing history'.

1.2.3 Programmer's code

With "programmer's code" we mean the source code that is written by the *end user* of the software visualization tool. The programs running in CoffeeDregs

are usually a combination of programmer's code and library code.

Similarly, "programmer's objects" are the objects that come from executing programmer's code and the "programmer's program" is the executable resulting from compiling the programmer's code.

Chapter 2

Didactic Vision and Problem Statement

2.1 A didactic vision for the teaching of object-oriented programming

This project, which was initiated by Kees Huizing and Ruurd Kuiper, aims to support the teaching of OOP to first year non-computer-science students of Eindhoven University of Technology. To be of any help, the proposed educational tool must fit the didactic vision of the teaching method and the book material[7].

Students are usually found running into problems when *starting* with OOP and programming in general. The problems we identified during the instruction lectures were of a fundamental programming nature: students sometimes have an incorrect perception of method scope; local variables and especially method arguments lead to misunderstandings, return values are not always correctly understood. Methods are not recognized as extended mathematical functions.

Therefore, although the course is about OOP, the method does not follow an "objects first" approach. Instead, the student is at first presented with only the most basic programming concepts that apply to every imperative programming language, such as boolean and arithmetic expressions, variable assignments, input and output, conditional execution (if-else) and repeated execution (while-do). When these concepts are well-understood, the student learns how to use and create methods. The student starts making a simple application with one class and a single instance of that class, without knowing it. The next step is to introduce objects (classes as well as instances) as environments of execution and to enter the world of OOP.

The teaching method is based on building up knowledge "incrementally, without branching or making de-tours"[7]. The course follows a "narrow path"[7], leading to a quick understanding of OOP concepts and Java "for the [conceptually] inclined traveler"[7]. It also leads to a thorough understanding of the subject

for the not-so-inclined traveler as s/he doesn't come accross side paths to get lost in.

The advocated approach is to start programming in a real environment right away, deferring explanation of the complex structure of Java program code to a later point. Every concept introduced has a clear and explicit "Aim", the "Means" to reach this aim (always with an example) and an "Execution model" to show visually what is happening inside the computer when it executes the newly learned means.

Even though this approach leaves the student with unexplained notation and terminology, we think it is better to only introduce those when the student has a good understanding of the basic programming concepts that are contained within them and can appreciate all the details of the new concept.

Summarizing:

- Narrow path approach: right to the point.
- First, give the student a good foundation of programming in general
- Then the main goal is a quick and well-founded understanding of OOP and Java
- Which can be deepened at the student's will.
- Every concept introduced serves a directly applicable goal and is introduced along with a model of the execution of programs.
- The approach is bottom-up: In every step making the existing knowledge part of something bigger.
- The target audience consists of students at a technical university and is not limited to future computer scientists.

2.2 Problem statement

The main objective of the project associated with this thesis is to create a software visualization for use in computer science education, particularly for a course in object-oriented programming with Java, fitting the didactic vision of the initiators.

The system will visualize the execution model of computer programs written in Java. Its emphasis lies on the concepts of basic object orientation (i.e. creating objects, understanding the difference between an object and a class, and referencing objects). It will not go into detail of expression evaluation as it is assumed to be understood by prior knowledge of this concept in mathematics.

The visualization must enable the student to reason about his/her program code. The student must be able to predict the result of each execution step, using the visualization, his/her own program code and his/her knowledge of the Java programming language.

As the system visualizes the *execution* of computer programs, programming concepts like inheritance, polymorphism, interfaces will not be a prominent part of the visualization. This might sound strange for a visualization that is aimed at an object oriented programming language, but we will see that for the particular application this is a good decision.

Chapter 3

Related Work

This thesis on the interactive visualization of the execution of object-oriented programs builds upon a number of techniques and ideas. In this chapter, we discuss other systems in the field which we compare to our didactic vision. We did not find systems that directly fitted our didactic vision to provide students with an appropriate execution model, but we discovered that one of these systems uses an interesting model which will be of use when developing CoffeeDregs, the tool we present in this thesis. We explore this model and its origins in the next chapter.

From a multitude of software visualization systems we selected the Jeliot[1], BlueJ[10] and JIVE[2] systems as reference-examples for three categories of systems that have aspects that are relevant to our aim. The first system is an algorithm animation framework, capable of animating the runtime of programs up until the level of expression evaluation. The second concentrates more on the educational aspect for the writing of object-oriented programs. The last system is a visual debugging environment for Eclipse, which can visualize the execution of programs.

Other software visualization systems include jGRASP[1], which gives a source-level understanding of programming and the discontinued IBM Jinsight. Since they focus on the development of program code and do not visualize the *execution* of programs, they will not be discussed here.

3.1 Comparing program visualization systems

When looking at software visualizations, we can classify them into different types based on various criteria. Price et al. suggest in [13] a taxonomy based on the answers to thirty questions about Scope, Content, Form, Method, Interaction and Effectiveness of the software visualization. The categories are discussed in detail, after which we will discuss the three selected visualization systems by using this taxonomy.

[1] `http://www.jgrasp.org/`

Scope

The questions in this category tell us something about the characteristics of the software visualization. Is the software visualization a system that functions as an example or can it generate visualizations of arbitrary programs, within a particular class or any program? How can the program class be described? In this thesis we are interested in a system that can visualize arbitrary Java programs. Does the software visualization scale with larger programs, can it even handle multiple programs simultaneously and how does it support concurrency? Does the system interrupt the running of the program or does it run alongside without interruption?

Content

The content category deals with what is visualized. The visualization can be designed to show something of a general algorithm or a specific program. The former does not depend on the implementation, while the latter does. This thesis focuses on program visualization systems.

If the system does program visualization, does it use the program code for this purpose? Can it visualize data structures? Does the system work in compile or run-time? How complete and true are the visualizations to the "reality" of the workings of the program or algorithm?

Form

The form category classifies the software visualization based on the elements used in it, like what medium is used (paper, computer screen, ...). It has questions on what graphical elements are used, what color and animation. The form is also determined by whether it uses other modalities than graphics, for instance hearing or touch. It also addresses whether there are multiple views or visualizations on the same program.

Method

How the visualization is specified is described in the method category. In the case of fully automated visualization, there is no specification, while an example system will *only* consist of specification. If the visualization requires a specification, does this require changing the program code or is it separate? Does the specification use the same programming language as the program it visualizes? Can the visualization be customized interactively?

Interaction

The interaction category describes how the visualization is navigated, whether there are methods of elision (the hiding of excess information with the purpose of removing clutter from the display) and how the time component of the subject is mapped to the visualization.

There are three types of mappings known: "static to static" in which the system creates a single snapshot of a certain instance of the program execution, "dynamic to static" where the system makes a single snapshot out of the complete execution of the program (e.g. a trace) and "dynamic to dynamic" in which the

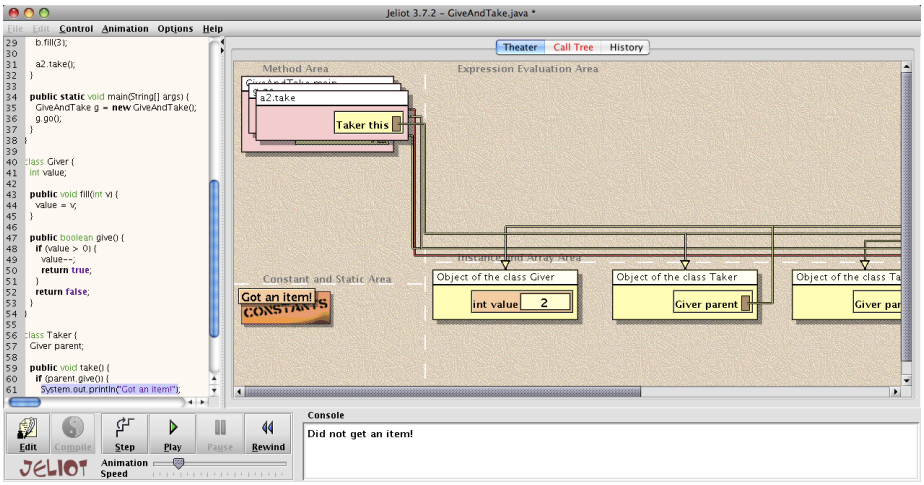


Figure 3.1: Screenshot of Jeliot. The main window is divided into four parts. The left top shows the active methods, the right top is used to show expression evaluations. The bottom left shows static values, the bottom right shows object instances.

system creates an animation out of the complete execution of the program. A fourth mapping, "static to dynamic" is not commonly used and would mean an animation of a single snapshot of a certain instance of the program execution.

Effectiveness

How effective is the visualization in reaching its goal? Have there been studies evaluating the system, or has it been in production use for a significant period of time?

We will now look at each of the three selected systems, put them into context, apply the taxonomy and come to a conclusion regarding our didactic vision of Section 2.1.

3.2 Jeliot

Jeliot is an animation system developed at the University of Joensuu, Finland. It is designed to help students understand the internal working of algorithms.

3.2.1 Taxonomy review

Scope

Jeliot animates *single file* Java programs. There are some limitations to the code that are mentioned in the Help function. It cannot run multiple programs simultaneously. Concurrency was not tested. The animation slows down the execution for visibility.

Content

The animation visualizes method executions, object instance and expression evaluations. The current execution point in the code is highlighted and there is limited syntax highlighting. The visualization is created in run-time and shows in high completeness and fidelity how expressions are evaluated. Constant values seem to get pulled out of a bin and output is visualized using a hand that grabs the value and pulls it down to the output window.

Form

Jeliot works on any computer with the Java SDK installed. The graphical elements are 2.5D graphic primitives, texts and arrows. Colour is used to distinguish scope types (method, instance, class, ...). Animation is used to emphasize the changes in two adjacent states.

Method

The animation is almost fully automatic, except for the input and output of values. For this the standard System.in/out and Scanner classes could be used, or the specialized Input and Output classes in the jeliot.io package. The animation is live and fixed (can not be changed by the user).

Interaction

Navigating through the animation is possible by speeding it up or slowing it down, pausing and making single steps instead of an automatic run. When the execution is paused, it can be rewound to start from the beginning. It is not possible to do reverse stepping or reverse execution.

Effectiveness

Jeliot has been investigated in educational settings. The direct predecessor to Jeliot 3, Jeliot 2000, was found most effective for the *mediocre* student. The strongest and weakest students did not get much out of it[11].

Two hypotheses posed in [12] are that new Jeliot's animations are comprehended better by those who have had previous experience with Jeliot and that student's expectations should be different when the tool is approached differently (as a learning aid or as a debugger in the case of the paper).

Students felt a "gap" in the tool, a point where the required knowledge suddenly makes "a big jump". The gap "appeared between theory and the application of that theory" and also "when students were not able to grasp a new concept". Other students thought that Jeliot had helped them a great deal, but later tests revealed they had an incorrect idea of the process of creating objects[12].

3.2.2 Conclusion

Jeliot is useful for explaining very low level programming concepts, such as expression evaluation and variable assignments. It is probably too low level for our purposes as we are mainly interested in objects and their relation to eachother, something Jeliot does not visualize.

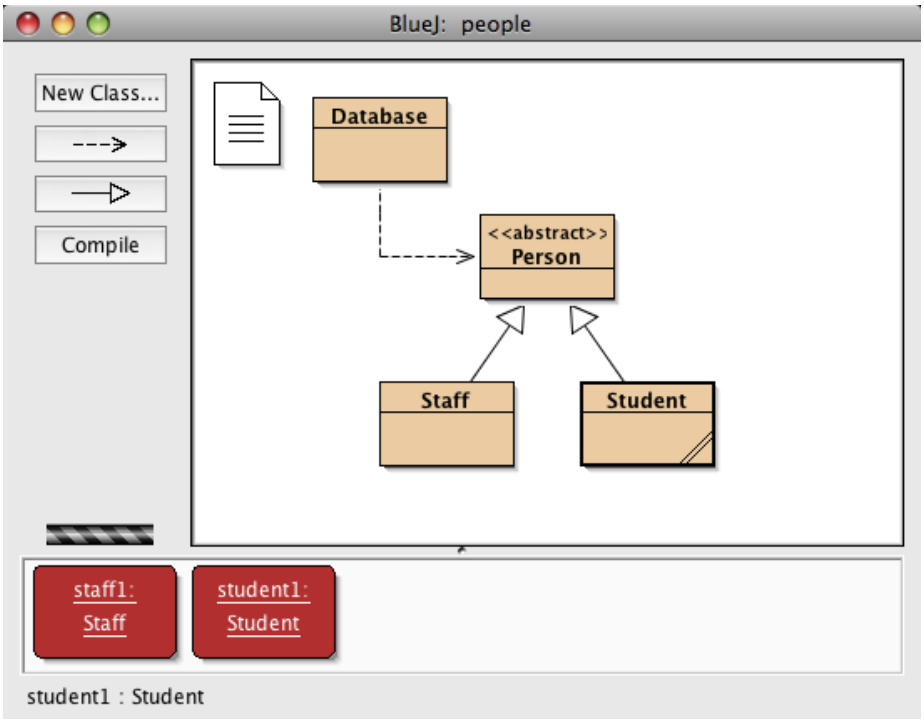


Figure 3.2: Screenshot of BlueJ. The main window shows object relations using UML notation, the bottom window shows two instances of Student and Staff. Static and instance methods can be called by right clicking the respective object and choosing from the pop-up menu. If a method requires arguments, a dialog window is opened to enter the values. The source code can be edited in the editor, which is opened via the pop-up menu.

3.3 BlueJ

BlueJ is developed at the University of Kent, UK. Its design is specifically aimed at teaching and learning OOP in Java. It is very different from JIVE and Jeliot in its workings and visualizations in the sense that it does not visualize program execution by running a program code and show what happens, yet its use as a learning-aid makes it interesting to review.

The user can in a graphical interface create classes, then add methods to those classes and run each of them from the user interface. When an instance of a class is made, it is placed on a shelve in the bottom of the screen, where it can be interacted with (methods can be run, values of instance variables can be changed).

3.3.1 Taxonomy review

Scope

BlueJ is (among other things) a visualization system designed to create simple Java programs in. The system is scalable enough for the class of programs it is designed for. It cannot handle multiple programs, nor does it treat concurrent programs different from non-concurrent programs.

Content

The visualization is of the program code structure. The system has syntax highlighting. Data structures are not visualized. The visualization shows design time, but the way the system works this means it works run-time.

The system gives a good insight in the relations between classes; their hierarchy as well as use-relations. On the object instance relations it falls short; pointers between objects are not visualized at all.

Form

BlueJ works on any computer with the Java SDK installed. The graphical elements are 2D graphic primitives and text. Colour is used to distinguish classes and instances. The system does not use animation and there is only a single view.

Method

The visualization is not generated from the code, rather the code is generated by using the visualization. Therefore there is no familiarity with Java needed to use the visualization, but to create a functional program it of course is necessary.

Interaction

In BlueJ, the user interacts directly with classes by dragging them around to reposition them, and right-clicking them to access their initializers, the source code editor and the class methods.

Object instances can be right-clicked to access their instance methods, inspect their fields.

Effectiveness

BlueJ is developed at the University of Kent, UK and is also used there. On the website is an list of 881 unconfirmed institutions using it to some extend.

There is a published evaluation[15] of the teaching and learning using BlueJ at Monash University. They evaluated the framework in a two unit course and concluded that in the second unit of the course — when the students were not forced to use BlueJ anymore, but could use any tool they like instead — “the fact that all of the respondents were using BlueJ indicates that they saw it as benificial to them”. The fact that no one switched from BlueJ to another tool might however also be for the same reason that most of the world is still using Windows.

The question whether skills learned in the course are easily transferable to other environments remains unanswered.

3.3.2 Conclusion

BlueJ is developed with a very specific pedagogy in mind. This approach treats objects first, giving the student a good understanding of classes and instances. Only then they introduce programming as an automated way to interact with objects.

This pedagogy is mirrored in the tool, presenting the user with an extended set of operations on objects, programming only being one of them. An advantage of this approach is that the novice programmer does not yet have to know what '`static void main(String[] args)`' means or that it even exists to be able to run a method on an object.

The pedagogy is much different from ours, where the emphasis lies on starting with basic imperative programming.

3.4 JIVE

JIVE is the Java Interactive Visualization Environment, developed at the University at Buffalo, New York, USA. It features *interactive* visualization, query based debugging and *reverse stepping*. It is integrated in the Eclipse[2] development environment and can not be run separately. Installation and maintenance of the software is handler by the Eclipse Software Updates feature.

Paul Gestwicki identified seven desiderata in his dissertation[5] which he used during the the design phase of his JIVE system:

- Depict Objects as Environments
- Provide Multiple Views of Execution State
- Capture History of Execution and Method Interaction
- Support Forward and Backward Execution
- Support Queries on the Runtime State
- Produce Clear and Legible Drawings
- Use Existing Java Technologies.

These desiderata might also prove useful for our own visualization system.

[2] `http://www.eclipse.org/`

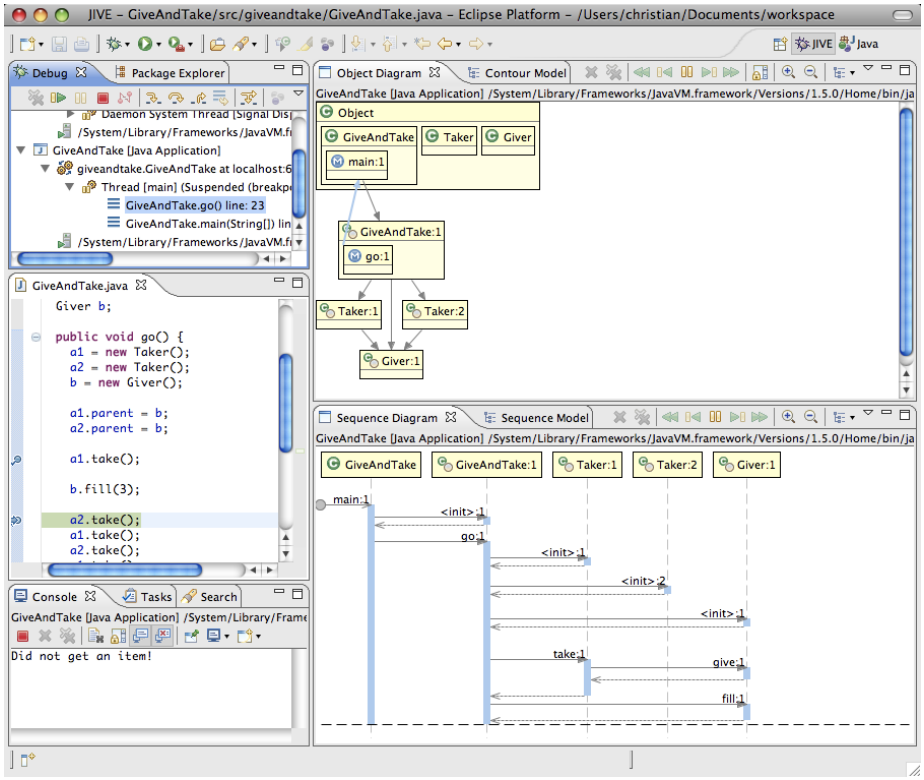


Figure 3.3: Screenshot of Eclipse debugging using JIVE. The right top window shows the object diagrams based on the Contour Model. The right bottom window show sequence diagrams based on the sequence model. JIVE is used as a standard debugger, so execution is controlled using breakpoints and step ins/step overs. After execution is finished, it is possible to revisit all states from beginning to end.

3.4.1 Taxonomy review

Scope

JIVE is a visual debugging system, designed to visualize the execution of (unaltered) Java programs. The system does handle larger software well, but the visualization is not very scalable. The "query based debugging" is limited, but useful. The system is not designed for running multiple programs simultaneously, but does support concurrency within a single program. The visualization is disruptive as it is mainly meant to be a debugger.

Content

JIVE visualizes the program execution. At the basis of the visualization lies the concept of Visual Operational Semantics [5], which builds on the Contour Model discussed in [9] of 1971 by John B. Johnston. The program execution is visualized by means of sequence diagrams which display calls from method to method *and* using object diagrams which display the "use" relations between objects and class hierarchy as contours[9]. The code itself is not visualized, and neither is the data visualized. The visualization uses data gathered at run-time.

Form

The primary target for the visualization are workstations running Eclipse IDE, since JIVE is integrated tightly in the Eclipse debugging environment. The graphical elements used in the visualization are various 2D graphic primitives, text and icons for the object diagram and labels, fat lines and arrows are used for the sequence diagrams.

Colour is not used for visualization purposes and there is no animation. There are multiple views on the software.

Method

JIVE is fully automated, data is gathered at the end of each execution step and then visualized. For small, terminating programs this usually means that the user can interact with the visualization only after the program has already terminated. The user does not need to be familiar with the code to generate a visualization, nor is it necessary to adapt the code.

Interaction

The visualization can be navigated through time using a set of "tape-deck controls". The backward execution mentioned in the design desiderata is an implementation of *reverse stepping*, JIVE does not feature true reverse execution. Scrollbars are shown if a lot of objects are created. Apart from query based debugging, there is no method for eliding[3] or culling[4] information. The visualization is the same without regard of scale. The user can zoom in and out, but

[3]From Wikipedia: Elision is the omission of one or more sounds (such as a vowel, a consonant, or a whole syllable) in a word or phrase, producing a result that is easier for the speaker to pronounce.

[4]From Wikipedia: Culling is the process of removing animals from a group based on specific criteria.

this also scales the labels, so it might not always be useful.

Effectiveness

The sequence diagram is very easy to read if the program to be visualized is fairly small. Once created, objects remain visible in the sequence diagram throughout the lifetime of the program, which can clutter the display considerably (the horizontal call lines will eventually become very long for calls to objects that were created very late in the execution). For the object diagram holds the same; garbage is not collected and all objects hang around for the entire execution, which also clutters the display.

The developers have no publications on the usability or usage of JIVE.

3.4.2 Conclusion

JIVE is a well-designed system with many features and possibilities, but these could easily overwhelm beginning programmers, who can not yet appreciate all the details of the execution of an object oriented software system.

The basis of the Contour Model[9] on which it builds is solid, so with well-aimed cuts the tool could be made very useful for beginners. We will look at the concepts behind JIVE that are relevant for our aims in more detail in Chapter 4.

3.5 A taxonomy preview for CoffeeDregs

To compare the selected systems with our requirements, we translate the aims and didactic vision to descriptions in terms of the taxonomy. It is the description of an, in our opinion, ideal software visualization tool aimed at the beginning programmer.

Scope

CoffeeDregs is a visual programming aid, designed to visualize the execution of (unaltered) Java programs. The system has a limited scalability, as it is mainly aimed at beginning programmers. The system is not designed for running multiple programs simultaneously, but does support concurrency within a single program. The visualization is disruptive, the user must invoke every execution step.

Content

CoffeeDregs visualizes the program execution. The program code is used in the visualization. It is not specifically visualized, but provided in a separate side window. Complex data structures are shown “as-is”, no specific visualization is applied to data structures. The visualization uses data gathered at run-time. The visualization tries to be as “true” as educationally useful, i.e. there are cases in which the fidelity is purposefully lowered.

Form

The primary target for the visualization are workstations running NetBeans, since CoffeeDregs is integrated in the NetBeans debugging environment. The graphical elements used in the visualization are various 2D graphic primitives, text and icons for the object diagram and labels, fat lines and arrows are used for the sequence diagrams.

Colour is used to support differences between instances, classes and methods. Animation is only used to smoothly display state transitions. There is a single view on the programmer's program.

Method

CoffeeDregs is fully automated, data is gathered at the end of each execution step and then visualized. The user does not need to be familiar with the code to generate a visualization, nor is it necessary to adapt the code.

Interaction

The visualization can be navigated forward and backward in time. Navigation in history is implemented using reverse stepping. Scrollbars are shown if a lot of objects are created. Objects can be collapsed and expanded at will to preserve space. The visualization automatically hides objects based on the collapsed/expanded state of the other objects. The user can zoom in and out, but this also scales the labels, so it might not always be useful.

Effectiveness

Two experiments with students, described in Chapter 8 proved the tool *as is* to be already quite useful. Research whether CoffeeDregs also works in an educational setting has yet to be started.

3.6 Summary of the taxonomy review

Table 3.1 summarizes the four systems discussed.

(a) Scope

A. Scope	JIVE	Jeliot	BlueJ	CoffeeDregs
1. System/ Example	System	System	System	System
2. Program class	Java source code	Java source code	Java source code	Java source code
3. Scalability	Good	Minimal	Minimal	Appropriate
4. Multiple Programs	No	No	No	No
5. Concurrency	Yes	No	No	Yes
6. Benign/Disruptive	Benign	Benign	Benign	Benign

(b) Content

B. Content	JIVE	Jeliot	BlueJ	CoffeeDregs
7. Program/Algorithm	Program	Program	Program	Program
8. Code	No	Yes	No	Yes
9. Data	No	No	No	Yes
10. Compile-/Run-Time	Run-Time	Run-Time	Run-Time	Run-Time
11. Fidelity and Completeness	Yes	Partial	Partial	Partial

(c) Form

C. Form	JIVE	Jeliot	BlueJ	CoffeeDregs
12. Medium	Workstation	Workstation	Workstation	Workstation
13. Graphical Elements	2D graph. primitives	2.5D graph. primitives	2D graph. primitives	2D graph. primitives
14. Colour	Yes	Yes	Yes	Yes
15. Animation	No	Yes	No	Yes
16. Multiple Views	Yes	No	No	Yes
17. Other Modalities	No	No	No	No

(d) Method

D. Method	JIVE	Jeliot	BlueJ	CoffeeDregs
18. Specification Style	Automatic	Automatic	By construction	Automatic
19. Batch/Live	Live	Live	Live	Live
20. Fixed/Custom.	Fixed	Fixed	Fixed	Fixed
21. Code Familiarity	N/A	N/A	N/A	N/A
22. Invasive	No	Yes (jeliot.io package required)	Yes (program is constructed from within application)	No
23. Customization Language	None	None	None	None
24. Same Language	N/A	N/A	N/A	N/A

(e) Interaction

E. Interaction	JIVE	Jeliot	BlueJ	CoffeeDregs
25. Navigation	Scrolling	None needed	None needed	Zooming, Scaling, Abstraction
26. Elision	Unknown	No	No	Yes
27. Temporal Control Mapping	Dynamic to Static	Dynamic to Dynamic	Static to Static	Dynamic to Dynamic

(f) Effectiveness

F. Effectiveness	JIVE	Jeliot	BlueJ	CoffeeDregs
28. Appropriateness and Clarity				-
29. Experimental Evaluation	[6]		[15, 14]	-
30. Production Use	No record	Used in programming course	Reasonably large user base (programming courses)	-

Table 3.1: Answers to taxonomy questions summarized

Chapter 4

A review of program execution models

When reviewing the JIVE software visualization system for Chapter 3, we noticed the creators had developed an execution model based on another fairly old method. This chapter discusses this old method and the extension by the creators of JIVE.

With the procedural programming languages in the 1960s came the eventual need for more structured programming and a strict separation of concerns. Program code was increasingly viewed as a structured combination of building blocks, and flowcharts were introduced as a way to structurally visualize the semantics of (parts of) an algorithm.

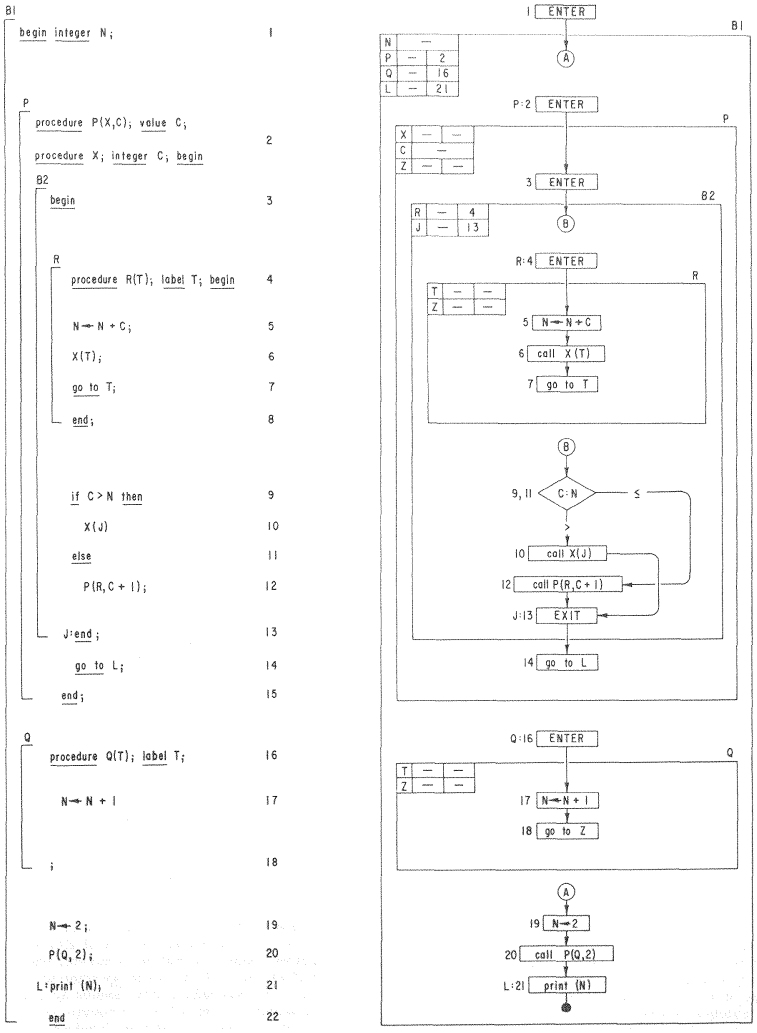
The mapping from flowchart to program is relatively easy compared to its inverse, mapping from program to flowchart. The levels of complexity of a program are often only implied in a flowchart and never made explicit. If one is to create a flowchart of a complete program on the level of execution, it is usually far too complex to understand it. To overcome this complexity, John B. Johnston introduced the Contour Model of Block-Structured Processes (CM) in [9], an "intuitive implementable cell-based model of the semantics of algorithm execution".

4.1 The Contour Model of Block-Structured Processes

The *Contour Model of Block-Structured Processes (CM)*[9] is developed with ALGOL 60 in mind, but works for any modern block-structured programming language. It uses and makes explicit the nested structure of algorithms and their records of execution. In this model a *process* is a series of *snapshots* containing a (time-invariant) *algorithm* and a (time-varying) *record of execution*. Each snapshot is the result of a single step of execution of the algorithm applied to the preceding snapshot. The CM is "a hypothetical computer designed to

support direct realization of block structured processes"[9].

Contours in the model are the structural equivalent of blocks in the program code. If we were to draw contours as rectangles, we would get a structure of nested contours. A contour holds local variables, execution steps, and zero or more nested contours. Figure 4.1 shows how contours enclose eachother based on the block structure of the program code; it is a visualization of the *static* structures in a program.

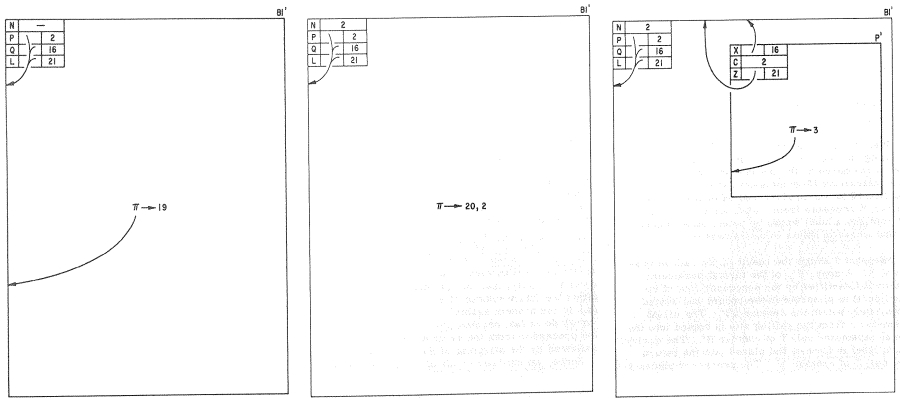


(a) Block structure (b) Static contour structure

Figure 4.1: The nested algorithm structure is reflected in the CM (from [9])

If we now start the program in Figure 4.1(a), we get a record of execution where each record contour can be linked back to its static counterpart. The first three snapshots of the program after initialization can be seen in Figure 4.2. We start with only the instruction pointer (π) which indicates at which line number we are currently executing (1) (not shown in the figure). If we take one step we

enter the B1' contour (based on the B1 contour in the static model), representing the complete program, the instruction pointer is placed inside the contour, with an arc pointing at the contour and the line number is updated to reflect the new position (19). Take another step, then the value 2 is assigned to the variable N and the line number is increased (20). In the next step a call is made to the procedure P, a contour P' is created, links are added and the line number is now 3. For a thorough description and illustration of the following steps, refer to Chapter 2 of [9].



(a) Enter contour $B1$ (b) $N \leftarrow 2$ (c) Call $P(Q, 2)$

Figure 4.2: First three snapshots of the program in Figure 4.1 (from [9])

The example in Figure 4.1 is specific for ALGOL 60, but the basic notation is flexible enough to be adapted to other programming languages. Linda Deneen tried already in 1987 to use the contour model as an instructional tool and adapted the model to fit the Pascal programming language. She declares that "the contour model of execution provides beginning programming students with a useful conceptual tool for understanding programs"[3], she "found a simplified model to be a valuable tool for teaching the fundamentals of scope rules and parameter passing."[3] Let us briefly look at an example of this.

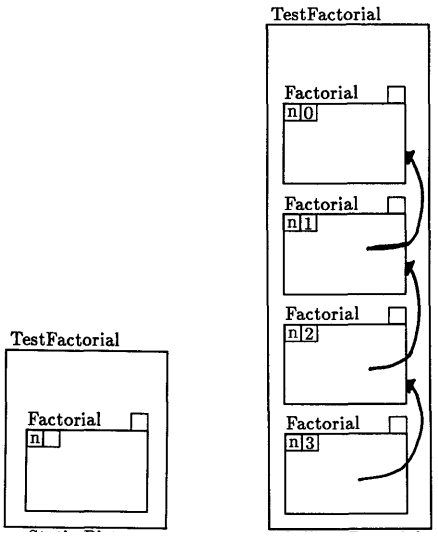
Listing 4.1: TestFactorial example

```
1   program TestFactorial (output);

    procedure Factorial(n: integer): integer;
    begin
      assert(n >= 0);
6     if n = 0 then
        Factorial := 1;
      else
        Factorial := n * Factorial(n - 1);
    end;
11
    begin
      writeln('3! = ', Factorial(3));
    end
```

In Listing 4.1 is a simple Pascal program, calculating the factorial value of 3. If we apply the CM to this program, we get the nested structure in Figure 4.3(a). When we run the program, the snapshot at the instant Factorial(0) is called looks like Figure 4.3(b).

Note that the dynamic contour structure follows the static nesting structure; so procedures are only nested in the dynamic model if they are nested in the static model (and thus in the source code). Recursive procedures are therefore never nested as they can not be nested inside themselves in the static model. ALGOL 60 and Pascal both support nested procedures. Java does not, so we would not see any of these nested procedures when we try to apply the CM to Java programs.



(a) Static contours showing nesting structure

(b) Dynamic behaviour

Figure 4.3: The CM with method executions (from [3])

4.2 Extending the Contour Model for OOP

Bharat Jayaraman and Charlotte Baltus found that the old CM was almost ready to account for object oriented and logical languages when they were looking for a good visual tool for describing the execution of programs written in those languages[8]. They conclude that a real extension of the model framework is only necessary for semantically visualizing inheritance. They found the CM to be very appropriate, since "the nesting structure of contours follow the scoping of environments of programs directly, [...] contours make explicit the important fact that objects are really environments." [8]

The actual extension to the model – the *Visual Operational Semantics (VOS)* – was developed by Paul Gestwicki, following Jayaraman's and Baltus' observations, with three desired properties in mind [5]. Firstly should it *clarify the method context* to emphasize that objects are the environments of execution in OOP. Secondly, the system must *support the Java language* to the extent that it can visualize classes, objects and interfaces, fields, methods and inner classes, and static variables. It must be able to show access control modi-

fiers, generic (template) and enumeration types, plus multiple threads using the `synchronized` keywords. Lastly, it should be *planar*, which specifically results in a drawing where no two diagrams need to overlap.

Method context

There is a visual difference between contours in the CM and the visual operational semantics. In the *contour model*, when execution leaves a contour, the contour is removed as the scope it represents does no longer exist. In the *visual operational semantics* the contours which represent object instances are kept even when no execution is active inside. This is explained by the fact that a reference to this scope still exists (i.e. a reference to the instance), so there is no reason to clean it up. Putting the method contours inside the instance contours (or class contours in the case of static methods) makes explicit that "an object is an environment within which method activations take place" [4], which satisfies the first desired property. Note that no change in the CM is necessary to accomodate for this!

Java support

Classes, objects, interfaces, fields and methods are trivial to support, they should each have a unique contour shape to be distinguishable and they should be properly nested following the source code of the program. Access control modifiers and other keywords are all part of the contour markup and are easily added. Part of the second desired property is hereby satisfied.

Inheritance is supported by using the hierarchical structure for the nesting of object contours. If we have for example an instance of a class A which is a child of class B, we would draw the instance diagram for A inside one of B, which in turn would be wrapped in an *Object* instance contour. This satisfies the rest of the second desired property.

Planarity

For the semantically correct display of the hierarchy, an instance of a class B which extends a class A should be drawn as in the top diagram of Figure 4.4, with static contours around the instances. This can lead to border crossing conflicts if there is more than one instance of B, as can be seen in the bottom diagram of Figure 4.4 (note that there is only *one* contour for each static object in the entire model!). By *separating* static space from instance space as shown in Figure 4.5, the conflict of two instances sharing the same static contour is resolved and the model is planar again, which satisfies the third desired property.

The model can now give a semantically valid visualization of any object oriented (Java) program.

4.3 Conclusion

The CM is a very intuitive and readable model for the visualization of an execution model. It provides separate snapshots for each execution state and brings the concept of scope very understandably. It is easily adapted to other program-

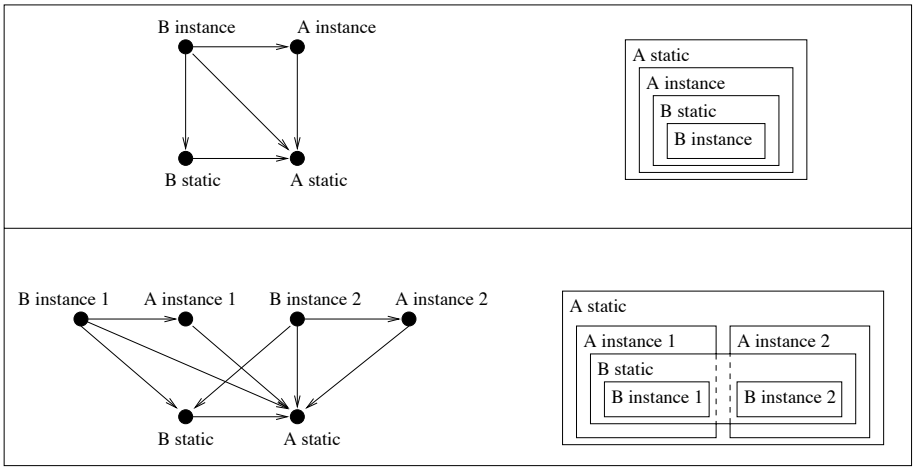


Figure 4.4: Illustration of how drawing static classes and instances objects in the same space can lead to border crossings and thus to non-planarity (from [5])

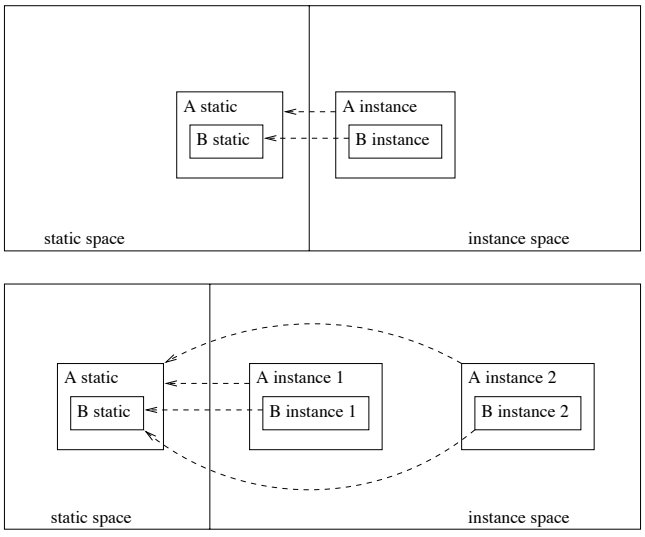


Figure 4.5: A solution to the contour conflict in Figure 4.4. (from [5])

ming languages because it uses a quite abstract model. The VOS, the extension to the CM to support OOP was useful to highlight the issues that play when integrating modern programming techniques in the older CM.

The CM has been used before in education, but was simplified for that purpose. For our educational aims, we also need a more specific solution, which is presented in Chapter 5.

Chapter 5

Visual Object and Execution Model

In this chapter, we discuss how we use the CM and its extension, the VOS, in an educational tool like CoffeeDregs. We present a variant on both models, which we call the *Visual Object and Execution Model* to emphasize we are interested mainly in objects and the execution of code in those objects.

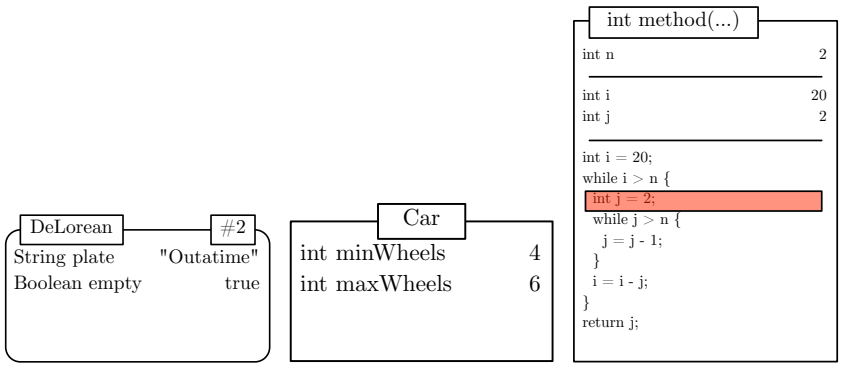
While the original complete CM breaks down method (or rather *procedure*) execution into smaller bits, we will not go this deep but rather break down the environment *around* method execution. We do this because we want to focus on objects as the environments of execution. In the model we use, method instances are therefore the innermost contours. Our model could in the future be extended to also show the internal method scoping, should the need occur.

5.1 Three types of contours

In our Visual Object and Execution Model we distinguish three different types of contours. Firstly, we define a contour for instances. Any object instance that is referenced, has one (but it is not necessarily visible, as we see in the next chapter). Secondly, there is a contour for classes, one for each loaded class in the program (again, they are not all necessarily visible). Thirdly, class and instance contours can *contain* zero or more method contours (which are *always* visible if they exist). An example of each of them is in Figure 5.1.

5.1.1 Instance contours

An Instance contour (see Figure 5.1(a)) contains the instance variables defined by the class (and all of its superclasses) along with their values, and a method contour for each active method in that instance. If an instance variable contains a reference to another object, a visual connection is made from the value to the object it is referring to. The Instance contours carry the class name of the



(a) An instance contour (b) A class contour (c) A method contour

Figure 5.1: Examples of object contours

dynamic type of the instance, so not necessarily the declared type.

The Visual Object and Execution Model is not affected by the planarity conflict mentioned in Section 4.2, because it does not show the hierarchical structure of the instances coupled with their static counterparts. It is decided to flatten the hierarchical nesting of contours, because it makes the visualization easier to follow. Because we aim to visualize the *execution* of programs, we only need to show the dynamic (i.e. runtime) type of all objects and references.

There is a major drawback to flattening the hierarchical structure, namely that it cannot be said anymore which instance variable stems from which class in the program code. This is a source for discussion which is continued in Section 5.2.

5.1.2 Class contours

A Class contour (see Figure 5.1(b)) contains the static variables defined in the class (and all of its superclasses) along with their values, and a method contour for each active (static) method in that class.

The Class contours are usually only visible if a (static) method is executing in its corresponding class, since every loaded class in the program has one and there would be too many of them without direct use.

5.1.3 Method contours

A Method contour (see Figure 5.1(c)) contains the method call arguments and the local variables, along with their values, and the complete program code of that method. A line is highlighted in the program code indicating the location of the execution within that method. A different highlight is used for the method that is on top of the execution stack to distinguish between active and waiting methods.

Methods are shown as a whole, and no smaller contours can be inside method contours. This is different from the CM, where every execution scope gets its

own contour. We merge the local variables of every scope in the method to a single list to display them in the method contour. This results in a more stable visualization, because the size of the contours only changes upon method entry and exit.

The CM does *not* put the complete program code in the dynamic model. We decided to do this, to emphasize even more that objects are the environments of execution. Also, we do not show a static model where we could put the program code, so we have to do it here.

In the CM, a special field value `rpdl` is added to indicate the "return point and dynamic link" of the method. The result is a pointer from the called method back to the line location in the calling method. We decided against this because it suggests that the programmer can choose to dereference this pointer at any time during execution, while in fact it is not a part of the program code abstraction level at all. Instead, we decided to turn the pointer around and draw an edge *from* the calling method location *to* the called method to suggest a directed flow of successive method activations.

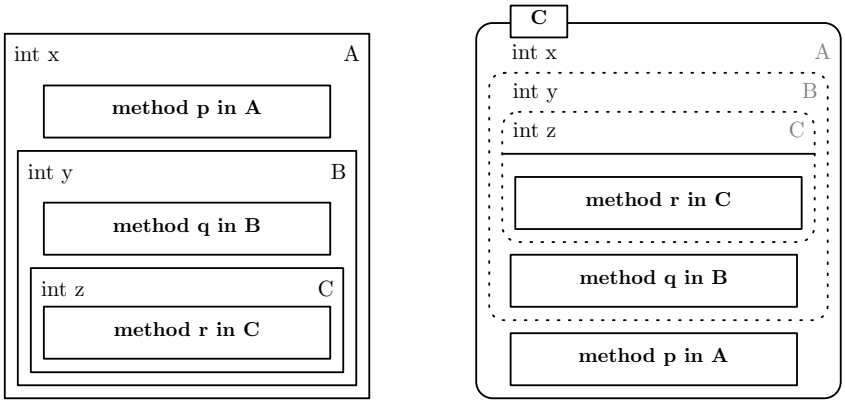
5.2 Hierarchical structure

We decided to merge the hierarchical scoping structure of an object instance into a single contour. With this, we risk to lose some valuable information regarding reachability of variables and methods.

We identified two specific situations in which this occurs. Firstly, through inheritance, an instance variable can get shadowed. A contour of an object of a shadowing class gets the 'same' variable twice, with different values and unclear is which comes from which class. This makes it impossible for the user to predict the next state based on the current state if the execution step uses one of these variables. It is our opinion however that variable shadowing is never done intentionally and even avoided if possible. A second situation occurs when a method is overridden. It can not be told from the method contour from which class this method is. Especially when the invocation triggers a chain of overridden methods to be executed, this can lead to confusion.

We suggest a method to be able to show method and variable scopes on demand, but that does not require scoping contours to be visible at all times. By slightly rearranging the contour structure as proposed in VOS, we can get a contour diagram that keeps variables of different hierarchy levels together and methods in order. In these reordered contour diagrams we can switch on and off the scoping contours without having to restructure its contents.

In Figure 5.2 we compare the hierarchical object instance structure of an object C which extends from a class B, which in turn extends from a class A. In VOS, instance variables and methods of the different hierarchy levels alternate. This happens because the nesting always takes place at the *bottom* of the contour. If we move the nested contour to the *middle*, in between the variables and methods of the enclosing contour, we get a contour diagram as in the right figure, where the instance variables are listed together, grouped by hierarchical class.



(a) Contour in VOS

(b) Contour in Visual Object and Execution Model

Figure 5.2: Comparing contour structure

Since we like to have a stable visualization, i.e. where the contents and size of contours does not change wildly, we want to avoid that method contours are added above other existing methods. For instance when a method in class B is active and another one in class C is executed, it would in theory end up *above* the existing method contour.

In practice, this does not occur too often to be an issue. Let us look at this in more detail.

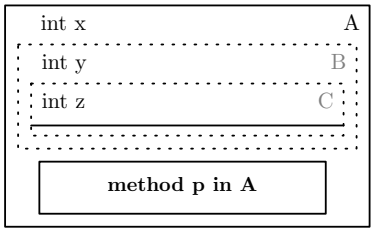


Figure 5.3: Only method p active

Suppose we have a method p in class A, a method q in class B and a method r in class C. If we execute method p, a method contour is added in the bottom of the object contour and we get a contour like in Figure 5.3. To get a method contour above the one we just added, we need to call method r or q, which would be added in the scoping contour for B or C respectively.

Since A is a superclass of B and C, methods in A can not call methods in classes B or C, except if it downcasts[1] itself to one of these classes. Downcasting has the drawback that it requires extra runtime type checking and is generally considered bad design if it is really necessary. As we are developing the model within an educational framework, we avoid these types of programs. Another way to execute one of the two methods is by running it from another thread. These two techniques seem to be *the only way* to get a method contour above an existing method contour in our scoping contours.

We thus have a method that does not require the scoping contours to be visible at all times, yet they can be displayed on demand to show how variables are shadowed and how methods are overriden.

[1] casting an instance of a class down to one of its derived classes

5.3 Model comparison

The models discussed in this thesis, while all visualizing execution of programs using contours, have different aims and use different means to satisfy those aims. Now, we compare the models.

5.3.1 Contour Model of Block-Structured Processes

The CM was developed in a time when there was not really a good method for visualizing software systems. The aim can therefore be called very straightforward.

Aims: To provide an intuitive, implementable cell-based model of the semantics of algorithm execution.

Means: The means to do this are nested containers called contours which are connected by arrows. Contours display variables along with their values. The contours must be interpreted as variable scopes: A process can access all variables that are in its own contour or in any of the enclosing contours.

5.3.2 Visual Operational Semantics

The VOS is part of a larger work for the interactive visualization of OOP.

Aims: To visually depict the current execution state, with support "for all major features" of the Java programming language.

Other parts of the work aim to show the history of the execution.

Means: It uses the same means as the CM, nested contours. Added to the contours is a separation between static and dynamic space.

5.3.3 Visual Object and Execution Model

Aims: To visually explain behavior as produced by executing code, using a notion of *state*. The state should, together with the static code, contain enough information to understand what the next state will be, using knowledge about the semantics of the programming language used.

More specifically, the following requirements support the aim:

- Object instances should be visualized as separate entities from the class they belong to.
- Object reference structures should be visualized.
- Lifetime and scope of data should be visualized (especially the differences between e.g. instance and local variables).

Means: A visualization where the state is depicted as a structure of various containers of data and where execution is associated with those containers, being a principal paradigm of OOP: Objects are the scopes of execution.

The requirements are translated into the following specific means:

- Objects are top-level containers. Inside the container the typed instance variables are represented.
- Methods (i.e., method instances) are depicted as subcontainers of objects.
- All active methods are visible and the call chain is visualized.
- The static part of a class is visualized as a separate class-object (in order to avoid clutter and data duplication).

5.4 Conclusion

By simplifying the CM and VOS, we get a useful model for beginning programmers who generally do not have a coherent model of the programs they create in their minds, yet.

The deliberate flattening of the hierarchical structure in our opinion makes the visualization of the model cleaner and more easy to follow. The rearranging of the instance contours compared to the VOS enables us to show the hierarchy when needed and hide it when not.

Chapter 6

Deciding Upon What Objects to Show

In the previous chapter we discussed *how* to show objects and methods. When executing a Java program, many auxiliary objects are created, resulting in a very large and incomprehensible visualization. In this chapter we will therefore discuss *what* objects and methods to show and how to responsibly hide the others.

6.1 Introduction

When a Java program is started, the virtual machine first creates a large amount of auxiliary objects to load the main class of the program. Before even the first line of the programmer's code is executed, there are many objects in the model, but that is not all. During program execution, the implementation of library components can also create many objects. Not all of these are very interesting for understanding the working of the program, but some might. Therefore we cannot just hide all of these extra objects and hope the execution state is still clear.

When objects in the model have no more references to other objects in the model, they are automatically removed. This leads to a problem with objects that are referenced by library objects, which are not present in the model. After all, the objects in the model are only the programmer's objects and the model does not contain the library objects.

We present a method to selectively show and hide objects, which both improve comprehensibility. The method is based on three levels of visibility, described in Section 6.4. The levels are determined by a number of rules described in Section 6.3. The rules use a set of dynamic properties of objects, which are identified in Section 6.2. Because the hiding of objects may lead to orphaned objects, we need to reconnect them to the object graph, this process is described in Section 6.5.

6.2 Dynamic properties of objects

In order to be able to define rules on the ordering of objects, we first need to identify the properties we want to use in those rules. We can distinguish between history-independent and history-dependent properties. Both types are dynamic properties, this means they only have a meaning during the execution of the program.

The history-independent properties can be checked without looking into the history of the object; these are properties like "There is a method active in the object right now" or "The object does not reference other objects". Below is a list of more examples. Note that not all of them might make sense or can even be computed, the list is merely an indication of what could be interesting properties to check for.

- One or more methods are currently executing in the object
- n methods are active in the object, but not currently executing
- The object is fully initialized (i.e. no init methods are active)
- The object is user-defined (i.e. "I wrote the program source")
- The object is user-selected (i.e. "I find this interesting")
- The object is in a specific package (e.g. com.sun.java)
- The object has n outgoing references to other objects (having a certain property)
- The object is referenced by n other objects (having a certain property)
- The object does not reference any other object
- The length of the longest path from this object back to itself following object references, without ever taking the same reference twice, is 42 (the calculation of which is in NP)

The history-dependent properties require some extra bookkeeping, especially when the system has support for reverse execution/stepping. To determine their value, we must be able to retrieve information from the past. These properties are therefore more complex, like "The reference count of the object is increasing" or "There have been n method activations in the object".

- The value of an instance/static variable has changed
- The object has had n method activations in its history
- Reference count of the object is increasing
- Reference count of the object is decreasing

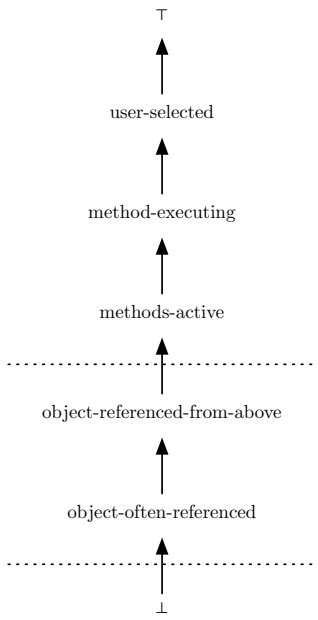


Figure 6.1: An ordering for objects

6.3 Rules for the ordering of objects

Now that we have identified a number of properties, we can use them to define rules to determine an order on the objects. The presented order in Figure 6.1 has five levels, but additional levels can be introduced if desired.

We first define a set of first-order rules in terms of dynamic object properties and previously assigned object order. The rules are applied in order for each object. The first-matching rule decides the order of the object.

1. A method is currently executing in the object, assign order *method-executing*
2. A method is active in the object, assign order *methods-active*
3. The object is user-selected, assign order *user-selected*
4. The object has no active methods and the object is referenced by at least 1 object of higher order than *methods-active*, assign order *object-referenced-from-above*
5. The object is referenced by x other objects, assign order *object-often-referenced*
6. No rule matches, assign order $\perp$

The result is that objects with active methods get a high order, while inactive objects get lower orders. If an inactive object is directly referenced by an active object or if it is heavily referenced, it gets a slightly higher order. In the next section we see what this means for the visualization.

6.4 Visual implications of object order

Each order has its visual representation. The objects of the highest order are fully expanded contours as they are discussed in Chapter 5, while the objects of the lowest order are completely hidden from view.

- Objects of order *methods-active* and higher are expanded.
- Objects of order between *object-often-referenced* and *object-referenced-from-above* are collapsed.
- Objects of order $\perp$ are hidden.

See the horizontal lines in Figure 6.1.

Between Expanded and Collapsed can be many levels of half-expanded and half-collapsed objects. We will discuss only the three levels here, other levels are derived from these three.

6.4.1 Expanded objects

A fully expanded contour. If a method is running, the user can opt to keep the object expanded after the method finished. If there is no method running, the user can collapse the object. The system can, based on user preference, decide to keep expanded objects expanded after a method has finished, or to automatically collapse it if nothing else is keeping it expanded. In Figure 6.2, the objects A and B are expanded.

6.4.2 Collapsed objects

A small empty box, which can be connected to and from, but which does not have any textual content. It is the smallest visible representation an object can have in the visualization. The user can select the object at will to become expanded (it will thus become of order *user-selected*). As a result of the rules, a collapsed object automatically becomes hidden if none of its referencing objects is expanded (i.e. it will become of order $\perp$). An example of a collapsed object is the object S in Figure 6.2.

6.4.3 Hidden objects

A hidden object which cannot be interacted with. It is used to resolve reference paths between collapsed and expanded objects. Hidden objects grow to collapsed objects if at least one of its referencing objects became an expanded object (*object-referenced-from-above*). T is a hidden object(s) between S and B in Figure 6.2.

6.4.4 Visual representations between Expanded and Collapsed objects

One could think of additional sublevels of expanded objects, in which for instance private variables are hidden if the object inherits from a non user-defined class. Beware that much of the functionality lies in these private variables and that, for example, a storage class can not show how the user-defined objects are stored within it if its private variables are hidden! The hiding of private variables should therefore probably only occur for "leaves" in the object reference graph.

Great care is therefore required not to confuse accessibility in OOP with reachability (and thus visibility) in the model. The aim of the model is to visualize the state of the program and *not* to visualize the structures in the program code. As a result, private variables may be visible, even though they are inaccessible from the program code. They nevertheless contribute to the execution model that is visualized and are therefore required to be visible.

If it is really necessary to hide variables, a better way probably is to merge them into a single 'super-variable' *private* which holds all references and thus the reference connections in the graph of the object model.

6.5 Tying objects back together

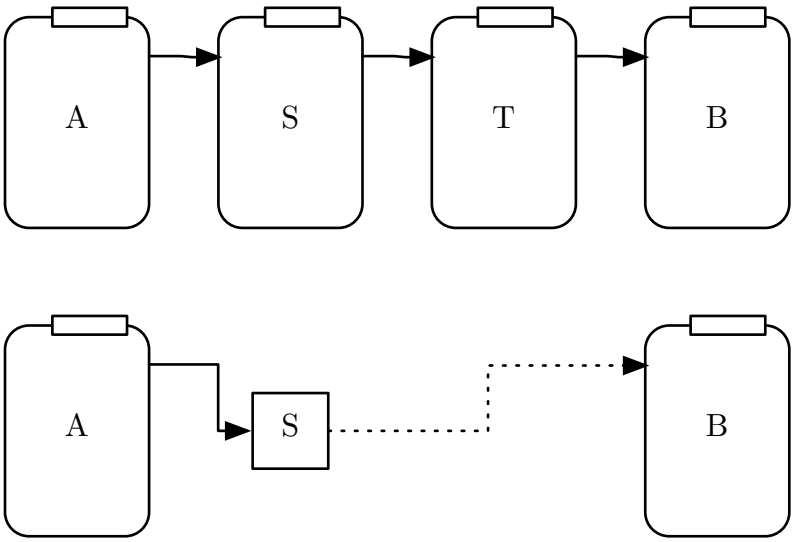


Figure 6.2: Using transitive links to reconnect expanded and collapsed objects. The objects in the structure in the top are ordered, which results in the image in the bottom: S is collapsed and all objects between S and B (only T in this case) are hidden. Object B is still expanded and to prevent dangling objects, all paths to B from a collapsed object are reconstructed by drawing transitive reference connections.

All objects have an importance assigned to them and as a result some of them now have become hidden. Now other higher-importance objects might have become dangling because they were only referenced from currently hidden objects. We will replace reference connections through objects of lower importance with "transitive" reference connections. They differ in appearance from the normal object references.

If there is a reference path from an expanded object A (see Figure 6.2), via one collapsed object S and then via one or more hidden objects to another expanded object B, then there a transitive reference from S to B is added. If there is a direct reference from a collapsed object S to an expanded object B (i.e. there are zero hidden objects in between them), then there is a normal reference from S to B.

6.6 Conclusion

The method of object selection through properties and rules enables us to be flexible in our selection approach. The important objects gain more screen space, while the display is not cluttered by the less important objects.

The same visualization applies for library code as it does for programmer's code. The only difference is that method invocations are not visible as the source code is not available to draw a method contour. The result is that a lot can happen in one execution step, but it is something the programmer has to become accustomed to since s/he will probably use black box library code a lot in the future. It also teaches about the separation of concerns: "you do not want nor have to know which storage implementation is used, as long as you know how you can store and retrieve it".

We discussed several alternative and chose the described method because it seems the best for our purpose. Following is a brief discussion of the alternatives.

Initially, the idea was to *show only the objects for which we have the source code*, thereby assuming that these are the programmer's objects. The first object in the visualization would then be the main class and only the few objects that are created in the program become visualized. This works fine for smaller projects that do not use libraries. As soon as we start using library code written by others, we will be missing part of the functionality of the program. It was decided that we would also be needing the other objects for which we do not have the source code.

Because we ran into this problem for the first time when using a list implementation from the Collections framework, and we would expect this to happen mainly with these types of objects, we decided it might be a good idea to have *a custom visualization of each collection implementation in the framework*. This however would make the visualization SDK dependent. Because of implementation differences[1] it could mean that our tool would work with Sun Java, but not with OpenJDK.

[1] For example, an ArrayList might in OpenJDK not be implemented using a variable called 'array' as in Sun JDK, but one called 'list'. The result is that a custom visualization depending on these variables would not work anymore.

Another way to visualize foreign classes is to use model variables as a way to abstract from the actual implementation. Model variables are values that can be stored and retrieved via methods on an implementation, which do not need to exist in the object, but rather in the model the object represents. To visualize objects in this way, however, goes against the idea that we want to show what is actually happening inside the computer.

Therefore, we can only treat library code as we would treat the programmer's code, but we still needed a way to keep the data structures comprehensible. Thus we have come to the approach presented before, automatically detecting and hiding 'less important' parts.

Chapter 7

CoffeeDregs

In Section 2.1 we proposed a didactic vision for the teaching of OOP to non-computer-science students. In this chapter we present a software visualization tool fitting this vision using techniques described in Chapters 5 and 6.

7.1 Introduction

CoffeeDregs is a software visualization tool aimed at helping the beginning programmer understand OOP. It makes use of the NetBeans Visual Library[1] for graphics and the Java Debug Interface for retrieving information from objects in the visualized program.

By using the Visual Object and Execution Model, it gives the user a visual representation of the execution of his/her program. The user can zoom in and out and rearrange objects to get a good view on the model.

7.2 Techniques

7.2.1 Visual Object and Execution Model

In CoffeeDregs we make use of the Visual Object and Execution Model, a variant on the CM [9] and VOS [5]. It is described in Chapter 5.

The Visual Object and Execution Model is implemented using the NetBeans Visual Library. This means that we can use the automatic routing of connection arrows and the automatic layout of object contours. Mouse navigation and zooming is included and it is also possible to export images directly from the visualization, for example for use in assignment reports.

Not every aspect of the Visual Object and Execution Model is implemented in CoffeeDregs, yet. The hierarchical contours as described in Section 5.2 have yet

[1] `http://graph.netbeans.org/`

to be added.

7.2.2 Deciding Upon What Objects to Show

To prevent overwhelming the user with large amounts of objects, CoffeeDregs has a method of selective information hiding. Objects are assigned importance levels and based on these values they are partly or completely hidden. The method is described in Chapter 6.

The result of applying object selection is seen in Figure 7.2, which shows the exact same state as Figure 7.3, only with all LinkedList instances and related classes hidden. We get a comprehensible model of the real object reference structure. It is immediately clear from the image that the Object instances are referenced from some object that is referenced by the name ll and type List. The exact way in which the Objects are stored is abstracted in favor of model clarity.

7.3 Usage

7.3.1 Installation

CoffeeDregs is available as a NetBeans plugin, which can be installed by adding the repository[2]. Any updates are installed and configured automatically through the NetBeans update facility.

7.3.2 Execution

There are two ways of executing a program with CoffeeDregs, comparable with the 'Run File' and 'Run Project' options of NetBeans. The former option runs the `static main` method in that class, while the latter runs the `static main` method in the main class for the selected main NetBeans project.

In either case, CoffeeDregs opens a tab (see Figure 7.1) with an object view on the left and the source code on the right. Along the bottom are buttons to control forward and backward stepping and to control the automatic layout of the object reference graph.

Objects can be repositioned and collapsed or expanded. The reference connections and the method execution thread will always follow the objects. They start from the variable value or source code line, respectively, and end in the referenced object or method.

If the program requires user input, it can be given via the input box just below the object window. Any program output is printed just above the input box.

[2] At the moment of writing, CoffeeDregs does not have a dedicated plugin repository, yet

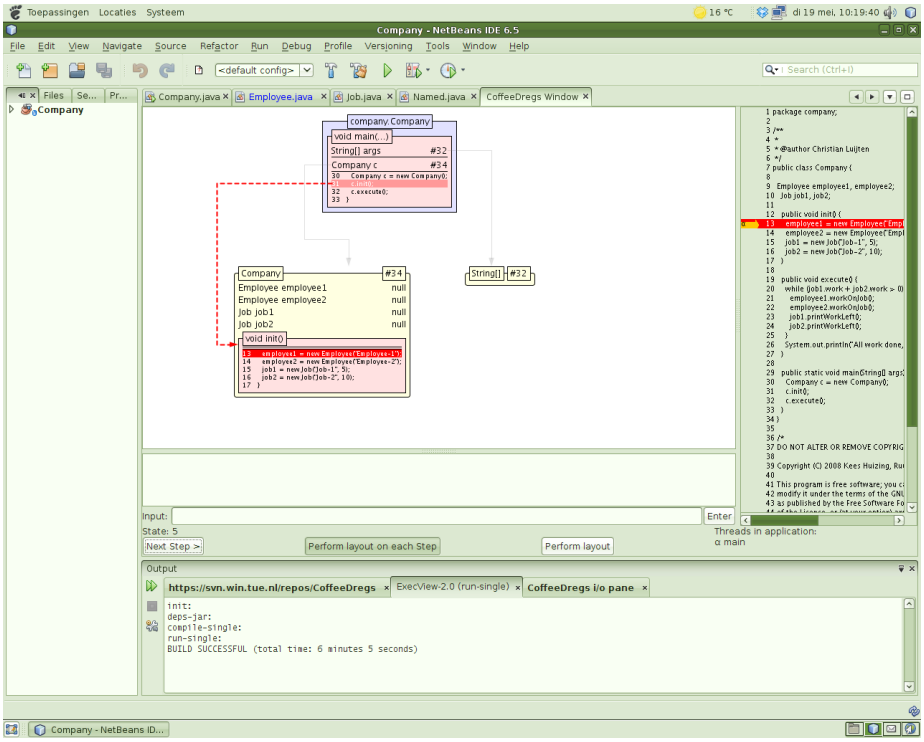


Figure 7.1: NetBeans running a program in CoffeeDregs

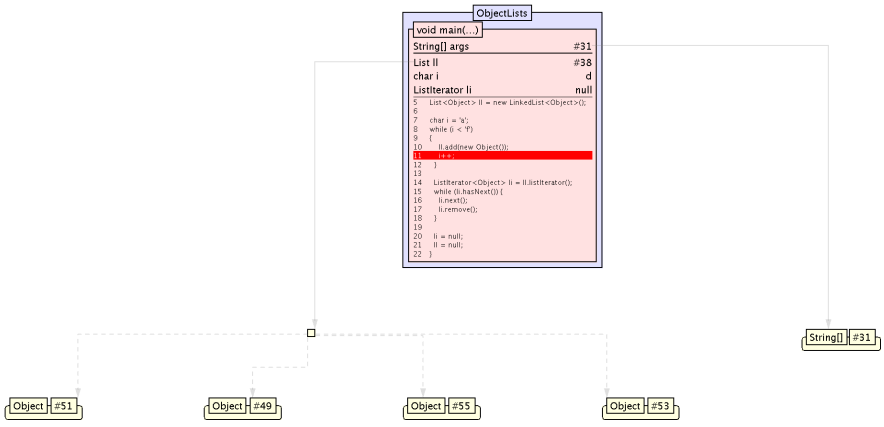


Figure 7.2: Lists example with object selection applied. The class ObjectLists has an active method main with one argument args. In the body of the method, three local variables ll, i and li are declared. Execution is currently on line 11 (for the fourth time, see the value of i) and the List ll contains four instances of Object. The instance of List is collapsed, so only a small square box is visible, with transitive reference connections to the objects in the list.

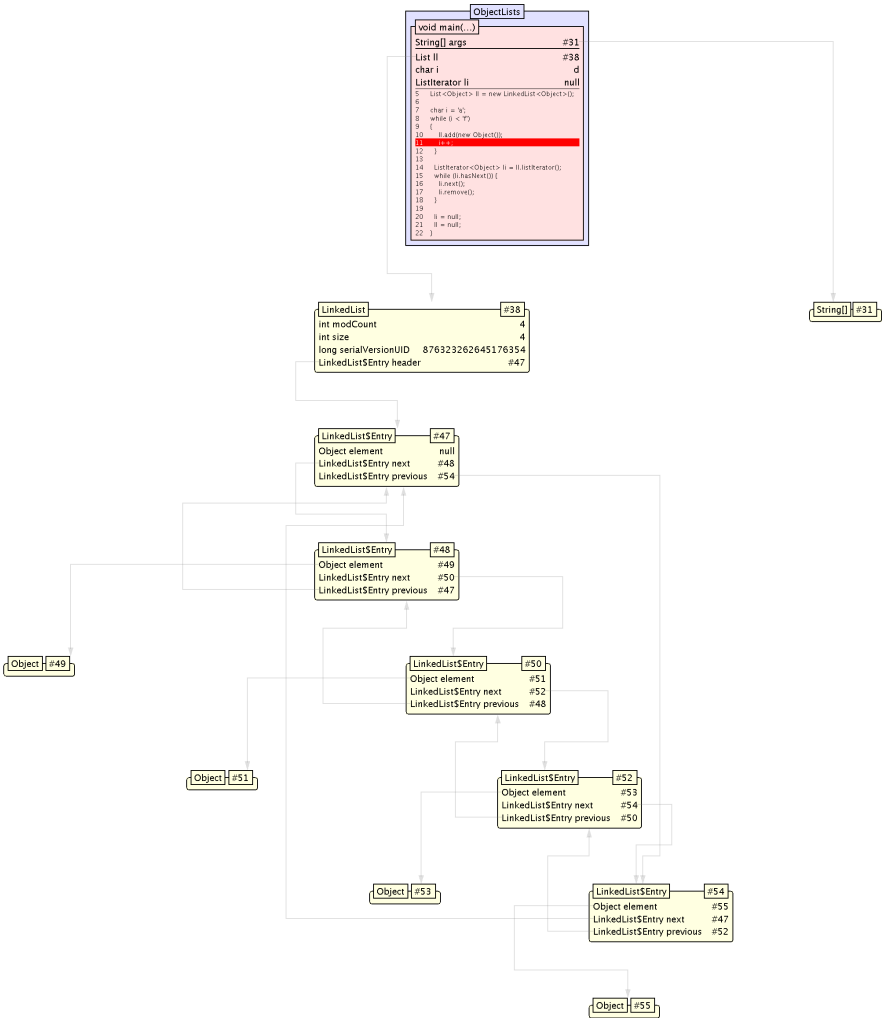


Figure 7.3: Lists example without object selection applied. The state is the same as in Figure 7.2, but the complete implementation of the List is visible, including its auxiliary objects.

7.4 Software architecture

This section is mainly aimed at the developer wanting to further extend and improve CoffeeDregs. We lightly explore the software architecture. Those interested in a more detailed discussion are encouraged to read the Developer Notes by Vincent Vandalon with contributions of the author of this thesis.

CoffeeDregs is made up of two main components thereby partly following the Model-View-Controller pattern: `debugmodel` and `debugview`. The `debugmodel` handles the communication with the Java debugger and keeps a consistent state model. The `debugview` makes a drawing of the state model and acts as a controller to `debugmodel`.

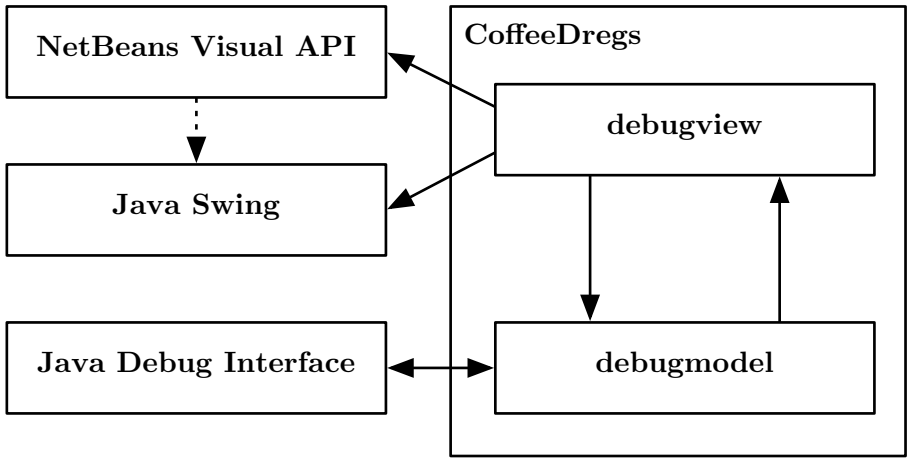


Figure 7.4: High level architecture diagram

When a programmer's program is loaded into CoffeeDregs, a VM is started for the program. CoffeeDregs then connects to the VM to subscribe to method entry and exit events and to execution step events. These events are reused as events within CoffeeDregs to notify the visualization of updates in the state.

When the visualization receives a notification, it updates its set of visualized objects. New objects in the programmer's program are added to the visualization. Old objects that do not exist anymore in the programmer's program are also removed from the visualization.

After the set of visualized objects has been updated, the objects in the set are updated to reflect changed values. If a method is active in an object, it is also added or its state is updated.

Following the update of the state is a reevaluation of the object's properties and applying the rules as described in Chapter 6, and then drawing the objects as described in Chapter 5.

The work on CoffeeDregs for this project is limited to the `debugview` component. Vincent Vandalon worked on the `debugmodel` component, improving the communication with the Java Debugger Interface and implementing reverse

stepping.

Chapter 8

User Experiments

We want to test whether CoffeeDregs is ready for prime-time. We do not intend to prove that CoffeeDregs is didactically valid right now, more research has to be done for that.

A group of students do a series of assignments using CoffeeDregs, both analysing as reproducing. The students have some experience with Java, having finished a programming course during fifteen weeks of four hours each.

In Appendix A is the form handed out to the subjects (in Dutch).

8.1 The setup

The installed plugin is a build of SVN revision 199 of the visualization branch. Secondary objects and transitive edges are not used, long lists of variables are not folded.

The computer runs Debian GNU/Linux 'lenny' with NetBeans 6.5 and all updates installed.

8.2 The assignments

The subject is first introduced with CoffeeDregs, how to launch it and how to use it. For this, the Arrays and Recursions projects are used. The experimenter first shows the subject around, then lets him/her play for him/herself.

Meanwhile, some introductory questions are asked to get an idea of the programming background of the subject.

Listing 8.1: ReverseTwice

```
3   public class ReverseTwice {

        void swapFirst(Obj x, Obj y) {
            Obj h;
            System.out.println("x = " + x.name + ", y = " + y.name);
8           h = x;
            x = y;
            y = h;
            System.out.println("x = " + x.name + ", y = " + y.name);
        }
13
        void swapSecond(Obj[] arr, int i, int j) {
            Obj h;
            System.out.println("arr[i] = " + arr[i].name + ", arr[j] = " + arr[j].name);
            h = arr[i];
18          arr[i] = arr[j];
            arr[j] = h;
            System.out.println("arr[i] = " + arr[i].name + ", arr[j] = " + arr[j].name);
        }

23      void reverseFirst(Obj[] arr) {
            for (int i = 0; i < arr.length / 2; i++) {
                swapFirst(arr[i], arr[arr.length - i - 1]);
            }
        }
28
        void reverseSecond(Obj[] arr) {
            for (int i = 0; i < arr.length / 2; i++) {
                swapSecond(arr, i, arr.length - 1 - i);
            }
33      }

        public static void main(String[] args) {
            ReverseTwice r = new ReverseTwice();
            Obj[] arr = new Obj[6];
38          arr[0] = new Obj("Obj-0");
            arr[1] = new Obj("Obj-1");
            arr[2] = new Obj("Obj-2");
            arr[3] = new Obj("Obj-3");
            arr[4] = new Obj("Obj-4");
43          arr[5] = new Obj("Obj-5");
            printArray(arr);
            r.reverseFirst(arr);
            printArray(arr);
            r.reverseSecond(arr);
48          printArray(arr);
        }

        static void printArray(Obj[] ar) {
            for (int i = 0; i < ar.length; i++)
53              System.out.print(ar[i].name + " ");
            System.out.println();
        }
    }

58  class Obj {
        String name;
        public Obj(String n) {
            name = n;
        }
63  }
```

8.2.1 ReverseTwice

In the ReverseTwice assignment, the subject must read and understand the code, predict the behaviour and then check using CoffeeDregs whether the actual behaviour matches with the prediction. The source code is listed in Listing 8.1.

The user should detect that the first swap function does only exchange references in local variables and not the actual references in the array.

8.2.2 GiveAndTake

GiveAndTake is an abstract variant of the often used Farmer example (there is a Farm with Animals, which can eat out of a shared Trough which can be empty and filled). The main instance GiveAndTake (Listing 8.2) creates two

Listing 8.2: GiveAndTake

```
public class GiveAndTake {
  Taker a1, a2;
  Giver b;

  public void go() {
    a1 = new Taker();
    a2 = new Taker();
    b = new Giver();

    a1.parent = b;
    a2.parent = b;

    a1.take();

    b.fill(3);

    a2.take();
    a1.take();
    a2.take();
    a1.take();

    b.fill(3);

    a2.take();
  }

  public static void main(String[] args) {
    GiveAndTake g = new GiveAndTake();
    g.go();
  }
}
```

Listing 8.3: Giver

```
class Giver {
  int value;

  public void fill(int v) {
    value = v;
  }

  public boolean give() {
    if (value > 0) {
      value--;
      return true;
    }
    return false;
  }
}
```

Takers and a Giver, lets a Taker (Listing 8.4) try to take something from the Giver (Listing 8.3) in which it fails. Then gives the Giver some resource and lets the Takers try again.

The idea is that the user understands better that the two Takers get from the same Giver.

8.2.3 Company

In the Company (Listing 8.5) assignment, two Employees (Listing 8.6) must be assigned to two Jobs (Listing 8.7). If all goes well, the employees do their job and after a few 'hours' they are done. The program however is flawed and no matter how much the employees work, there is no progress. The cause is that neither the job nor the employee have any reference to each other. The user should be able to diagnose and fix this problem by running it in CoffeeDregs.

Listing 8.4: Taker

```
class Taker {
  Giver parent;

  public void take() {
    if (parent.give()) {
      System.out.println("Got an item!");
    } else {
      System.out.println("Did not get an item!");
    }
  }
}
```

Listing 8.5: Company

```
public class Company {

  Employee employee1, employee2;
  Job job1, job2;

  public void init() {
    employee1 = new Employee("Employee-1");
    employee2 = new Employee("Employee-2");
    job1 = new Job("Job-1", 5);
    job2 = new Job("Job-2", 10);
  }

  public void execute() {
    while (job1.work + job2.work > 0) {
      employee1.workOnJob();
      employee2.workOnJob();
      job1.printWorkLeft();
      job2.printWorkLeft();
    }
    System.out.println("All work done, it's weekend!");
  }

  public static void main(String[] args) {
    Company c = new Company();
    c.init();
    c.execute();
  }
}
```

Listing 8.6: Employee

```
public class Employee {

  Job job;
  String name;

  public Employee(String n) {
    name = n;
  }

  public void workOnJob() {
    if (job != null) {
      System.out.println(this + " is working for an hour on " + job);
      if (job.work()) {
        System.out.println(this + "'s work on " + job + " is done!");
        job = null;
      }
    }
  }
}
```

Listing 8.7: Job

```
public class Job {

  Employee employee;
  String name;
  int work;

  public Job(String n, int w) {
    name = n;
    work = w;
  }

  public void printWorkLeft() {
    System.out.println(this + " has " + work + " hours of work left by " + employee + ".");
  }

  public boolean work() {
    if (work <= 0) {
      return false;
    }
    work--;
    return (work <= 0);
  }
}
```

8.3 Experiment reports

8.3.1 Experiment #1

Study: Innovation Sciences. Followed Programming 1 (2Z820) in 2007, currently doing Programming 2 (2Z830). In his opinion "an average student, which misses a bit of the routine others might have".

The subject is presented with the Arrays example project. He gets instructions on how to start a program in CoffeeDregs from within NetBeans. Then he starts playing around on his own to get a feeling for the tool.

ReverseTwice

Reads the program code thoroughly. Explains step by step what he thinks the program does, namely reversing the array of objects twice so that in the end it comes back to its original order.

Runs the program in CoffeeDregs and is very focused on the source code in the right pane. Thereby he misses that the first swap function does not work and that the order of the objects is unchanged after the first reverse function. When pointed at this ("did you notice something strange in the order of the printout?"), runs the program again and then sees what happens in the left pane and explains the error the program made.

GiveAndTake

Reads the program code thoroughly. Correctly predicts the outcome of the program, runs it in CoffeeDregs and explains the state in each step and predicts the next step.

Company

The subject first reads the code thorougly and notices that indeed the program should not work as it was meant because job and employee are not assigned to eachother. An execution using CoffeeDregs confirms this.

The subject finds it very convenient to see the instances next to eachother to compare their instance variables and to see that indeed the jobs and employees have no reference to eachother.

It is not directly recognized that CoffeeDregs can also be used to trace reachability of variables when fixing the problem. The result is trying to set the instance variables in a place where the jobs or employees are not reachable.

Conclusion for experiment #1

The level of the assignments was up to par with the subject. The subtle bug in the ReverseTwice was noted at the intended moment (be it with a little help). The whole experiment can therefore be seen as representative for the situation of a lecture setting.

8.3.2 Experiment #2

Study: Innovation Sciences. Followed Programming 1 (2Z820) in 2008, currently doing Programming 2 (2Z830). In his opinion "an average student".

ReverseTwice

Reads the program code thoroughly. Discovers that swapFirst does not actually change the array. This is confirmed when he runs the program in CoffeeDregs, references in the array are untouched, the subject mentions that he sees that swapFirst does not even hold a reference to the array.

GiveAndTake

Reads the program code thoroughly. Correctly predicts the outcome of the program, runs it in CoffeeDregs and explains the state in each step and predicts the next step.

Note: Both subjects recognized this as an abstract Farm program, which they had seen in the programming course.

Company

The subject reads the code thoroughly and quickly finds out that there are references missing, without having used CoffeeDregs. He fixes the problem and runs it in CoffeeDregs to confirm that is has been fixed.

Conclusion for experiment #2

The subject detected the flaws and subtleties of the assignments early, before having used CoffeeDregs. The assignments probably were too easy for the subject. The subject however did use CoffeeDregs to check its mental model with the actual model to make sure it was equal. The experiment can therefore be seen as representative for someone who understands OOP, but who wants to be sure of him/herself.

8.4 Results

The link between program code and visualization is not always recognized. Especially instance variables and method arguments pose a problem. It is expected however, that if the students are presented with the tool during the first few classes, those problems will not appear as they have no prior knowledge on the subject, and must be taught everything from the ground up, anyway.

The controls are clear. The functionality to drag objects to make a diagram more readable was not used as the examples were not that complicated. The subjects had little trouble to complete their tasks.

A stronger coupling with the NetBeans debugger might be desirable to be relieved of the need of having the program code along with the visualization (a better integration would make it possible to follow the debugging process in the source editor).

Chapter 9

Conclusions

In this thesis we have proposed (a model for) a tool for supporting the teaching of object oriented programming.

We have compared various tools with similar aims, to better understand our own and to identify the requirements for an educational software visualization system.

We discovered that the JIVE system comes close to our aims and uses a model we could adapt to our needs. We explored the model and identified the required changes needed for a more pedagogic approach.

We have created a variant on the model behind JIVE and added a method of information elision and culling to prevent overwhelming the beginning programmer with too many objects.

We have implemented a major part of the new model and the first experiments are promising that the system is ready to be used in the upcoming course.

9.1 Concluding Remarks and Future Work

CoffeeDregs can be further extended with a fundamental model for inner classes and to implement the idea discussed in Section 5.2 to better show the object hierarchies. More research has to be done to better specify what objects are interesting to the programmer and whether the interest changes over time as the programmer gets more experienced.

Code for stepping back in the history of the execution is already written by Vincent Vandalon, but still needs some work. The integration in the tool is a goal for the next months before introduction in the educational programme.

For intermediate programmers, the lack of a “run faster” option is a deal breaker. In a future version, there should be support for breakpoints and a fast forward mode in which the program runs until it ends or encounters a breakpoint.

Overall, the user interface can be improved to better match the look and feel of

NetBeans. A better integration with the NetBeans debugging system, probably much in the way JIVE is integrated in Eclipse's, would help in improving the user interaction.

Of course, the biggest issue still open is whether CoffeeDregs works in an educational environment. We hope the pedagogic research can start in the fall semester of 2009.

References

[1] Mordechai Ben-Ari, Erkki Sutinen, and Jorma Tarhio. Perspectives on program animation with Jeliot. In Stephan Diehl, editor, *Software Visualization : international seminar, Dagstuhl castle, Germany, May 20 – 25, 2001 ; revised papers*, volume 2269 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2002.

[2] Jeffrey K. Czyz and Bharat Jayaraman. Declarative and visual debugging in Eclipse. In *Eclipse '07: Proceedings of the 2007 OOPSLA workshop on Eclipse technology eXchange*, pages 31–35, New York, NY, USA, 2007. ACM.

[3] Linda L. Deneen. The contour model as an instructional tool in elementary computer science courses. *ACM SIGCSE Bulletin*, 19(1):170–178, February 1987.

[4] Paul Gestwicki and Bharat Jayaraman. Interactive visualization of Java programs. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, page 226, Washington, DC, USA, 2002. IEEE Computer Society.

[5] Paul V. Gestwicki. *Interactive visualization of object-oriented programs.* PhD thesis, State University of New York at Buffalo, NY, USA, June 2005. Adviser: Bharat Jayaraman.

[6] Paul V. Gestwicki and Bharat Jayaraman. Jive: Java interactive visualization environment. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 226–228, New York, NY, USA, 2004. ACM.

[7] Kees Huizing and Ruurd Kuiper. Object Oriented Programming - with Java. Available online on TU/e Studyweb, 2008.

[8] Bharat Jayaraman and Charlotte M. Baltus. Visualizing program execution. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 30, Washington, DC, USA, 1996. IEEE Computer Society.

[9] John B. Johnston. The contour model of block structured processes. *SIGPLAN Not.*, 6(2):55–82, 1971.

[10] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4):249–268, December 2003.

[11] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2002.

[12] Andrés Moreno and Mike S. Joy. Jeliot 3 in a demanding educational setting. In *Proceedings of the Fourth Program Visualization Workshop*, number 178 in Electronic Notes in Theoretical Computer Science, pages 51–59, 2007.

[13] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 2:597–606, Jan 1992.

[14] Vijayakumar Shanmugasundaram, Paul Juell, Curt Hill, and Kendall Nygard. Effectiveness of BlueJ in learning Java. In Craig Montgomerie and Jane Seale, editors, *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2007*, pages 3776–3781, Vancouver, Canada, June 2007. AACE.

[15] Kelsey van Haaster and Dianne Hagan. Teaching and learning with BlueJ: an evaluation of a pedagogical tool. In *Information Science + Information Technology Education Joint Conference*, Rockhampton, QLD, Australia, June 2004.

Appendix A

Form handed out during experiment (in dutch)

Kennismaking

Welke studierichting volg je?

Hoe schat je je eigen programmeervaardigheden in vergelijking met medestudenten in? (bovengemiddeld / gemiddeld / ondergemiddeld)

Kennismaking met CoffeeDregs

Je gaat nu met behulp van CoffeeDregs een programma starten en de uitvoering ervan bekijken. Probeer hardop te denken en vraag het als er iets niet duidelijk is. Denk eraan dat het doel van de experimenten *niet* is om te testen of jij het begrijpt, maar of CoffeeDregs het duidelijk laat zien!

Er is een aantal voorbeelden beschikbaar die verschillende facetten van CoffeeDregs demonstreren. Open deze, klik wat rond en kijk wat er gebeurt. Speel met de layout van de objecten op het scherm en kijk welk effect de verschillende knoppen hebben. Geef hardop commentaar terwijl je bezig bent.

ReverseTwice

Open het project ReverseTwice in NetBeans. Lees de code door en probeer te begrijpen wat deze doet. Voorspel – zonder het programma te starten – wat de uitvoer van het programma zal zijn.

Start nu het programma in CoffeeDregs en pas gaandeweg je voorspelling zonodig aan. Blijf hardop commentaar geven en stel vragen als iets je niet duidelijk is.

GiveAndTake

Open het project GiveAndTake in NetBeans. Lees de code door en probeer te begrijpen wat deze doet. Voorspel – zonder het programma te starten – wat de uitvoer van het programma zal zijn.

Start nu het programma in CoffeeDregs en pas gaandeweg je voorspelling zonodig aan. Blijf hardop commentaar geven en stel vragen als iets je niet duidelijk is.

Zelf aan de slag

Je gaat nu een aanpassing maken aan een bestaand programma, zodat het doet wat je wil.

Het gaat om een zeer vereenvoudigde weergave van een bedrijf (Company) met twee werknemers (Employee) en twee taken (Job). Voor beide taken staat een bepaald aantal uren werk en de werknemers moeten taken krijgen toegewezen.

Company heeft twee methodes, `init()` en `execute()`. `init()` maakt de twee Employee objecten en de twee Job objecten. `execute()` zet de objecten aan het werk zolang er werk te doen is. Zodra al het werk klaar is (`job1.work + job2.work <= 0`), stopt de while-lus in `execute()` en mogen de werknemers weekend vieren.

De Employee heeft een methode `workOnJob()` wat de werknemer een uur lang aan zijn toegewezen taak doet werken. De Job heeft een methode `work()` die wordt aangeroepen vanuit `workOnJob()` om te zorgen dat de hoeveelheid overgebleven werk afneemt. Job heeft ook nog een methode `printWorkLeft()` om af te drukken hoeveel werk er nog gedaan moet worden.

Start het programma. Je zult zien dat het werk niet minder wordt. Jouw taak is het om te achterhalen wat het probleem is en te zorgen dat het werk gedaan wordt en de twee werknemers kunnen genieten van hun weekend.

Bestudeer de code, probeer te begrijpen wat elke methode doet of zou moeten doen. Start de code in CoffeeDregs en kijk wat er gebeurt. Pas vervolgens de code aan en probeer het effect van je wijziging te begrijpen.

Nabespreking

Vond je de opdrachten makkelijk of moeilijk?

Voldeed het experiment aan je verwachtingen?

Denk je dat CoffeeDregs nuttig zou zijn geweest als je het tijdens je cursus programmeren had gehad?

Vind je CoffeeDregs fijn in gebruik? Waarom wel/niet? Mis je functionaliteit?

Heb je verder nog opmerkingen?