TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

Visualization of Dynamic Program Aspects

By

Pieter Deelen

Supervisors:

dr. ir. H.M.M. van de Wetering (TU/e)
dr. C. Huizing (TU/e)
dr. ir. F.J.J. van Ham (TU/e)

Eindhoven, June 2006

Abstract

Object-oriented software is designed by introducing classes and their relationships. This design is then transformed into source code by a software developer. But when the software is executed, the developer loses sight of the classes he created during the design and coding of the software.

This thesis describes the design and implementation of a tool which shows the developer which classes interact during execution. This tool, called TraceVis, allows the developer to study the executions of Java programs. It uses bytecode instrumentation techniques to extract relevant information from these executions. This information is presented using several visualizations.

To determine the usefulness of TraceVis it was evaluated at the end of the project, by comparing it to similar tools, and by running it through a few test cases.

Contents

List of Figures

List of Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BCEL	Byte Code Engineering Library
BCI	Byte Code Instrumentation
JAR	Java Archive
JDI	Java Debug Interface
JDK	Java Development Kit
JDWP	Java Debug Wire Protocol
JPDA	Java Platform Debugger Architecture
JUNG	Java Universal Graph/Network Framework
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface
MVC	Model-View-Controller
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

Nowadays, much software is developed in an object-oriented fashion. The idea behind object-orientation is that, during execution, a program is comprised of a collection of objects, which contain data and interact by sending each other messages. Each object has a class, which defines the data and behavior of the object. When working in an object-oriented way, a software developer first designs a structure for his program by introducing classes and their relationships. He then implements these classes by writing source code in one of the many object-oriented programming languages. To construct this software, the developer reasons about classes and objects using a mental model. To check whether this reasoning is correct, the software is executed. During execution, however, the concept of classes and object is hidden. This is reasonable, because the end user, the person the developer creates the software for, is not interested in how a program works, but rather in what it does. However, the developer might want to know what the software does internally.

The study of program executions can be interesting, because through execution many facts can be found which are not (immediately) apparent from the source code. Such facts include the number of objects which are created from a certain class, or the amount of time which is spent in a certain method.

Several kinds of tools already exist to inspect the inner workings of an executing program. The most prominent of these are the debugger and the profiler. As its name suggests, a debugger is meant to find bugs. With a debugger, a programmer can suspend the execution of a program at certain points, inspect the program's state, and resume the program's execution again. A debugger gives the developer a fine-grained control of the program, but it does not provide any overview of the execution. A profiler is meant to find performance problems in software. To this purpose, a profiler gathers certain statistics about the execution of a program, such as the time spent in a certain class or the memory usage over time. After the program has finished, the profiler presents these statistics to the developer. This gives a high-level overview of the execution, but not many details.

One method to provide insight into executions is to create a visualization. Visualization is the process of using computer imaging techniques to provide insight into abstract data, like executions. A visualization can be used make complex information presentable. The tools mentioned previously do not use visualization usually. The visualization of software is called software visualization. Two kinds of software visualization can be discerned, viz.:

- visualization of static software aspects, and
- visualization of dynamic software aspects.

Static software aspects are software aspects which can be determined before the execution of the software. Source code and UML class diagrams are examples of static software aspects. Dynamic software aspects are software aspects which can only be determined during the execution of the software. The number of object creations and the number of sent messages are examples of dynamic software aspects. A dynamic software visualization can still use static aspects of the software, such as the classes which are defined in the source code. In fact, it is beneficial to do so, because this creates a link between the artifacts the developer has created and the execution of the program, which aids understanding.

This thesis describes a tool which uses visualization to show aspects of program executions. An execution can have many different aspects, so a choice had to be made. After some experimentation, we chose to focus on class interaction. In other words, we want to visualize which classes send messages to each other, and to show which classes create objects of each other.

1.1 Project Goals

The main goal of this project is to design and implement a tool through which a user can gain insight into the dynamic interaction between classes. This information should be presented using visualization. The tool, which we called TraceVis, should be able to collect relevant data from the execution of programs. In theory, this tool could visualize programs programmed in any object-oriented programming language. However, it would then be necessary to implement a separate data collection module for several languages. Therefore, we chose to focus on one language for practical reasons, viz. Java. Java is a modern and popular object-oriented programming language. Many programs have been written in Java, and it provides interfaces which make collecting the right data possible. While it is not a strict requirement that the tool is also implemented in Java, it makes it easier to use these interfaces.

In the beginning of the project, we decided to target a group of users which is interested in program executions, and try to create a tool which tries to answer questions these users might have. The following section will discuss the target user group and the user questions.

1.1.1 User Questions

The study of program executions is something which mainly interests programmers. Therefore, our target users are programmers. Different programmers might have different purposes for studying executions. One programmer might try to gain insight into a program he is not familiar with. Another programmer might be trying to solve a resource problem.

As stated before, we would like to give programmers insight into the dynamic interaction between classes. More precisely, using TraceVis, the programmer should be able to answer the following questions:

1. Which classes interact, i.e., call each other?
2. How frequently do these classes interact?
3. Which methods are called?
4. How many instances does each class have?

5. Which classes are high-level (mainly send calls) and which are low-level (mainly receive calls)?

The first four questions can not only be applied to the whole execution, but also to subranges of the execution. For instance, a programmer might ask which classes interact during the initialization of a program.

1.2 Document Structure

The remainder of this thesis is structured as follows. Chapter 2 discusses several design aspects. Chapter 3 describes which data is needed for the visualization and how it is collected. In Chapter 4, the visualizations used in TraceVis are discussed. The resulting tool is evaluated in Chapter 5. This thesis is concluded by Chapter 6. In addition, this thesis contains two Appendices, viz., Appendix A, which discusses some aspects of the Java language, and Appendix B, which describes the input format for TraceVis.

Chapter 2

Design

This chapter discusses several design aspects, viz.:

- The data model underlying the visualization (Section 2.1).
- The visualization process (Section 2.2).
- The structure of the visualization tool (Section 2.3).

2.1 Data Model

The data model provides the data which the visualization shows. As such, it should contain enough information to allow the visualization to answer the user's questions, as defined in Section 1.1.1. These questions are mainly concerned with classes and the calls between them, i.e., they deal with entities and the relationships between them. The data model is therefore structured as a directed graph. This directed graph is called the *call graph*, and is defined as follows:

- Vertices represent classes.
- A directed edge from the vertex representing `A` to the vertex representing `B` represents all calls from a class `A` to a class `B`.

All elements of the call graph store information about the entities they represent. Each vertex has attributes concerning the class it represents, viz.:

- the number of calls this class has made,
- the number of calls this class has received, and
- the number of instances of this class.

In this case, and in the sequel, the number of instances of a class means: the number of objects whose class is this class, where the class of an object is the class it was created from, not any of its superclasses. Each edge has information about the calls made between its target and its source, viz.:

- the total number of calls from the source to the destination, and
- the number of times each method from the destination class has been called by the source class.

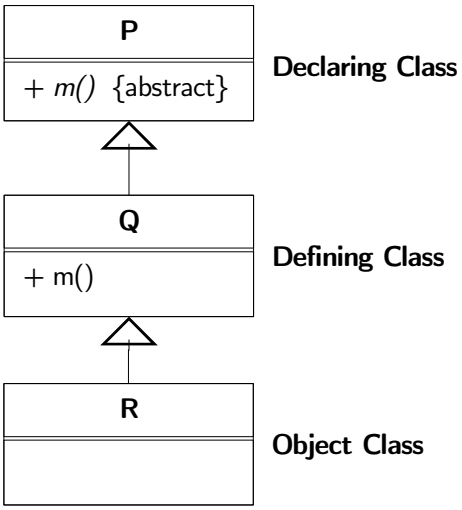


Figure 2.1: An example class hierarchy, depicted as a UML class diagram. The class `P` declares the abstract method `m`. Its subclass `Q` redefines `m`. The class `R` is a subclass of `Q`, but does not redefine `m`.

2.1.1 Call Assignment

The phrase "a call from class `A` to class `B`" is not clear enough. It can have several meanings, which differ in the way calls are assigned to classes.

For instance, consider the class hierarchy, depicted as a UML class diagram, in Figure 2.1. The class `P` declares the abstract method `m`. Its subclass `Q` redefines `m`. The class `R` is a subclass of `Q`, but does not redefine `m`. Suppose that an object of another class calls the method `m` on an object of class `R`, which class was called? It might be `P`, because this is the class which declares `m`. It might also be `Q`, because it defines the called method, i.e., the actual code which was called is contained in the class definition of `Q`. The final candidate is `R`, because an object of this class was called. So, to summarize, a call can be assigned to several classes, viz.:

1. The *declaring* class (or interface): if a method `m` is called on an instance of class `C`, the declaring class of `m` is the highest superclass of `C` which contains a definition or declaration of `m`. In the preceding example, `P` is the declaring class for the call.

2. The *defining* class: if a method `m` is called on an instance of class `C`, the defining class of `m` is the lowest superclass of `C` which contains a definition of `m`. In the preceding example, `Q` is the defining class for the call.

3. The *object* class: if a method `m` is called on an instance of class `C`, the object class is `C`. In the preceding example, `R` is the object class for the call.

Note that these classes need not be different classes. A single class might fulfill multiple roles. For instance, if a call is made to method `m` of an object of class `Q`, as described above, then this class is both the defining class and the object class for this call.

Each of these three options provides a different view on the interaction between classes. Each one has its own drawbacks and merits. Assigning all calls to the declaring class provides a high-level view, because declaring classes are high in the inheritance hierarchy. This option is mainly relevant from the point of view of class design. It can be used to view dependencies

between classes. If all calls are assigned to the defining class, the call graph provides a strong link with the actual code, because the code of the called method is contained in the source file of the class this call is assigned to. Therefore, this option is particularly interesting if the user wants to see which parts of his code are related and which parts are being used the most frequently. If all calls are assigned to the object class, the call graph emphasizes the interaction between objects, which is central to the object-oriented paradigm. The view this option provides is that a class is the collection of its instances.

The first option creates a static view of the execution. This conflicts with our wish to create a tool which visualizes dynamic behavior. We, therefore, did not consider it worthwhile to implement this option. The other two options create distinct views, which are both useful to gain insight into the dynamic behavior of the program.

2.1.2 Time

The execution of a program is a process. Hence, time is an important aspect of an execution. Two important times in the execution are:

- the start time of an execution, and
- the end time of an execution.

The execution is said to be started when the JVM has been initialized. The termination of the JVM is taken to be the end.

As is mentioned in Section 1.1.1, the user questions can be applied not only to the whole execution, but also to subranges of this execution. To allow users to select these subranges, the data model defines two times, viz.:

- the metric start time, and
- the metric end (or current) time.

The metric start and end times define the range in which calls are counted. For instance, the class attribute which counts the amount of calls which a class has sent, will only count those calls which have been made between the metric start and end times. These two times can be changed by the user to specify a subrange of the execution.

The user will not want to input abstract numbers to specify the metric times. To give the user a better idea the data model defines a list of execution events, which a visualization can use to show what happens in the execution. These events are:

- class loads,
- object allocations,
- object deallocations,
- method entries, and
- method exits.

All events are tagged with a time stamp, to determine when the event happened. The list of events is ordered by this time stamp. Class load events can be used to show the user when a class has been loaded. Object allocation and deallocation events can be used to give the user an idea of the number of instances a class has at a certain point in the execution. Method entry and exit events can be used to show when classes are active.

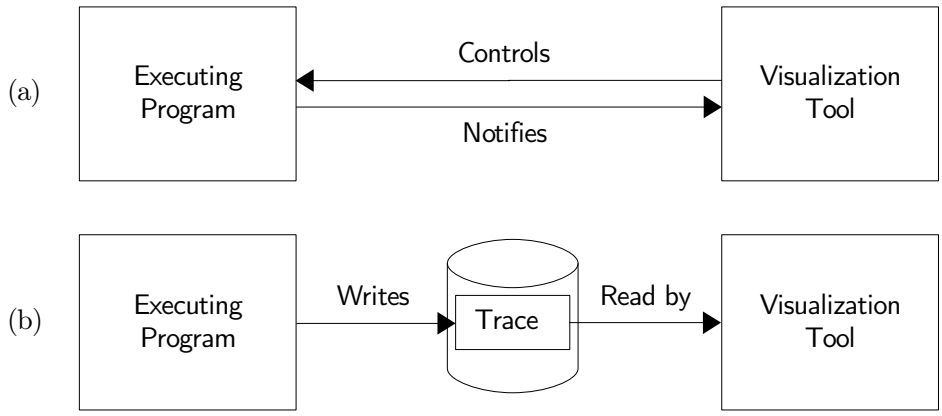


Figure 2.2: Two approaches to the visualization of program executions (see Section 2.2): (a) *Online visualization*: the executing program runs alongside the visualization tool and notifies the tool of interesting events. The tool controls the execution of the program. (b) *Offline visualization*: the executing program writes a trace file, containing interesting events, to disk. After the execution has terminated, the visualization tool reads this trace file and visualizes it.

2.1.3 Filters

To reduce the amount of information which is presented to the user, the data model allows the user to specify filters. A class can be filtered out of the list of events, which means that all events directly or indirectly related to this class are removed from the list of events. The result of applying a filter is that the attributes of all classes are modified to reflect the new situation.

The data model also supports a filter which shows just constructor calls. This can be used to show which classes allocate objects of other classes. In addition, it has a filter which hides all constructor calls.

2.2 Visualization Process

A tool which visualizes the execution of a program can be run either during or after the execution of this program.

In the first case, called *online visualization* (shown in Figure 2.2(a)), the visualization tool runs alongside an execution of the program under study. The program transmits information about its execution to the visualization tool. The visualization tool updates its views continuously to show the program's state. It allows the user to suspend and resume the executing program.

In the second case, called *offline visualization* (shown in Figure 2.2(b)), the executing program stores information about its execution in a trace file. After the program has terminated, the visualization tool can read this trace file and visualize it.

Both alternatives have their advantages and disadvantages. With an online visualization the user can associate external behavior (e.g. the response to clicking on a button) with internal behavior (e.g. a method call). This works best if the executing program has a user interface or responds to input in some other way (e.g. a web server handling page requests). The feedback this provides is immediate, which allows the user to play with the program and

see what happens. Because most programs handle their input quickly, the user should be able to suspend the executing program and replay at least part of it in slow-motion.

With an offline visualization, it is more difficult to make associations between external and internal behavior. On the other hand, it can provide an overview of the whole execution of the program. The trace which is produced can be studied in its entirety.

Because an online visualization tool and the program under study are run simultaneously, they have to share resources: screen space and processor time in particular. Sharing screen space can be a problem for programs with large user interfaces, because there may not be enough space left to provide a useful visualization. Sharing processor time makes both the program under study and the visualization tool run slower. To keep both programs responsive, special attention must be paid to the performance of the collection of execution data and the visualization. Due to this performance constraint, it might not be possible to collect certain data or to use certain visualization techniques. Note that this is not a fundamental problem, but merely a practical one. One might use two screens and two processors, one screen and one processor for the visualization tool, and one of each for the program under study. For an offline visualization, using a fixed set of resources, more data can be collected and it can be visualized using more processor intensive visualization techniques, before the slow down becomes unacceptable.

Online and offline visualization are not mutually exclusive. They can be combined in a single tool. With such a tool, the user first runs the program under study alongside the visualization tool, and interacts with the program and the visualization. During execution, the program stores a trace with data about its execution. The user can then load this trace into the same tool and review the program's execution. Such a tool can offer the best of both alternatives. In online mode, the user can interact with the program and see what happens internally. In offline mode, he can study the program's execution in more detail.

TraceVis was an online visualization tool initially. After some experimentation with this tool, we experienced that, while an online visualization tool provides interesting opportunities, it would require a high performance data collector to be practical. We, therefore, decided to transform it into an offline visualization tool in order to focus more on the visualization aspects. Because an online visualization tool (and, hence, the online mode of a combined tool) is impractical without a high performance data collector, we did not consider it worthwhile to build a tool which combines online and offline modes.

2.3 Program Structure

TraceVis is programmed in Java 1.5. It uses the standard Swing library for the user interface, Java2D for graphics, and version 1.7.0 of the JUNG library [14] for graph data structures and algorithms.

The tool is structured around the *Model-View-Controller* pattern (MVC). Section 2.3.1 describes this pattern. Section 2.3.2 describes how this pattern is implemented in the visualization tool. The resulting design is discussed in Section 2.3.3.

2.3.1 Model-View-Controller Pattern

The Model-View-Controller pattern defines three components: a *model*, a *view*, and a *controller*. The model encapsulates core data and functionality. The model data is visualized by one or more views. Each view has an associated controller, which handles user input.

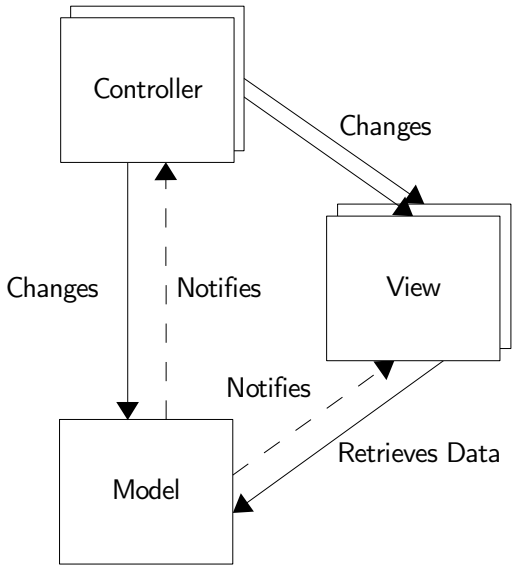


Figure 2.3: The Model-View-Controller pattern, as described in Section 2.3.1

The relationships between these components are depicted in Figure 2.3. Controllers receive user input, which is translated into requests for the model or the view. If the model changes, either by these requests or by itself, it notifies[1] the views and controllers. The views then retrieve data from the model to update their display to the model's current state.

The MVC pattern strictly separates the core functionality from the user interface components. This allows the implementation of multiple views, which are synchronized by the model's notifications. A view or controller can also be changed easily, without affecting the model.

The most important benefit of the separation of the view and the controller is that a controller could be used in combination with different views, and vice versa. They are, however, closely related. In practice, it is often difficult to separate them completely. The *Document-View* pattern, a variant of MVC, relaxes the separation between these two components. It defines two components: a *document*, which is equivalent to the model in MVC, and a *view*, which combines the view and the controller of MVC in a single component. Document-View sacrifices the interchangeability of views and controllers, but still separates the core functionality from the user interface.

The Model-View-Controller and Document-View patterns are described in more detail in [5, Section 2.4].

2.3.2 Tool Components

In TraceVis, the MVC pattern is implemented. Figure 2.4 shows its structure in a UML class diagram.

The class `tracevis.model.Program` implements the model. It maintains the call graph and the list of execution events. It uses the class `tracevis.model.TraceGenerator` to generate traces and the class `tracevis.model.TraceReader` to read them. By implementing the

[1] By using the *Publisher-Subscriber*, or *Observer*, pattern [5, Section 3.6].

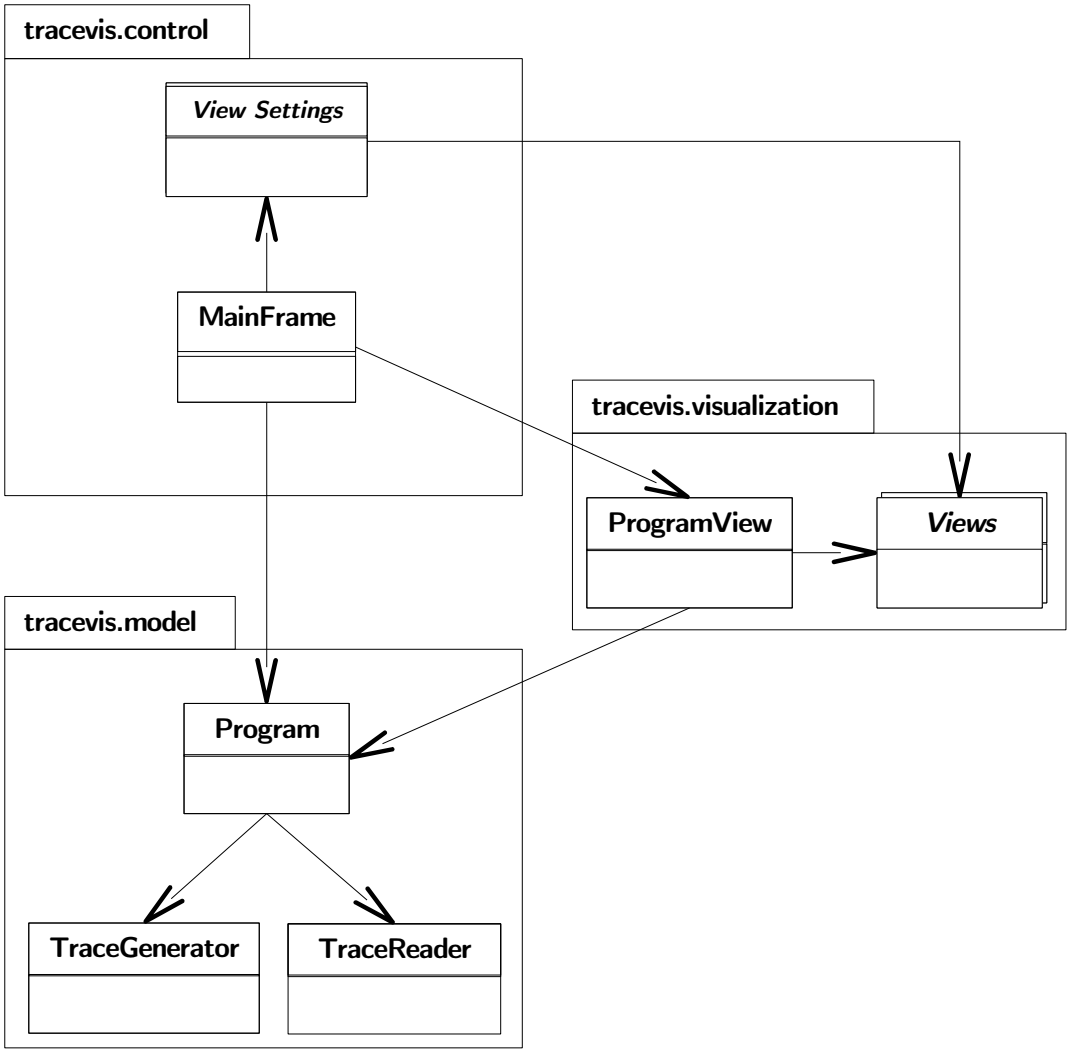


Figure 2.4: The components of the visualization tool, as described in Section 2.3.2.

`tracevis.model.ProgramListener` interface, views and controllers can register themselves for notification of changes in the call graph.

The `tracevis.visualization` package contains the view classes. The main class of this package is `tracevis.visualization.ProgramView`, which is a container for all views.

The `tracevis.control` package contains the controller functionality. Its main class, `tracevis.control.MainFrame` provides the user interface for the visualization tool. Part of this user interface are the view settings dialogs, which allow the user to configure the views.

2.3.3 Discussion

The previous sections described the initial program structure. During the project, this structure has held up fairly well, even when large changes to the tool's functionality have been made.

However, the controller and view components are not as separated as they could be. In

fact, most views handle their user input by themselves. Doing otherwise would have required that these views expose much of their internal parts, which would make changing them much harder. Because the implemented views have very different functionality, reusing controllers would not have been practical. So, in the end, the resulting program structure looks more like Document-View.

Chapter 3

Data Collection

To implement the data model as it is described in Section 2.1, certain data about a program's execution needs to be collected. This chapter describes which data this is and how it is collected.

3.1 Data Requirements

The data needed for the data model is collected by the data collector. The data collector tracks events in the execution of the program under study and records them in a trace, as described in Section 2.2. As described in Section 2.1.2, the data model needs the following events:

- Class loads, to determine which classes are loaded during execution. The data collector should determine the fully qualified name of the loaded class.

- Object allocations, to determine which instances a class has at a certain point of time in the execution. For each object allocation event, the data collector should determine the fully qualified class name of the allocated object and a unique identifier, which can be used to identify the object in later events.

- Object deallocations (due to garbage collection), to determine which objects are no longer used. For each object deallocation event, the data collector should determine the object's unique identifier.

- Method entries, to determine which calls take place. For each method entry event, the data collector should determine the thread in which this call took place, the name of the called method, the name of the class which defines this method, and, if the called method is an instance method, the unique identifier of the called object.

- Method exits, to determine which method is active. The data collector should track normal and abrupt (due to an uncaught exception) method exits.

The data collector should tag each event with a timestamp, to be able to reconstruct the time line of events.

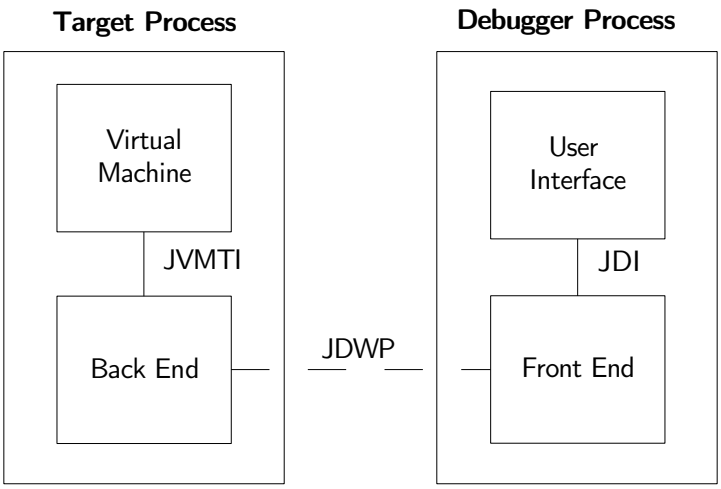


Figure 3.1: The Java Platform Debugger Architecture, as described in Section 3.2.1.

3.2 Techniques

The data described in the previous section can be collected in several ways. This section discusses the techniques which were used for this purpose.

3.2.1 The Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) [13] provides functionality which can be used to write debugger applications. Despite its name, the JPDA is not only suitable for debugging. It can also be used to monitor executions, for profiling, and to visualize program executions.

Figure 3.1 shows the structure of the JPDA. The program being debugged or monitored runs on the target virtual machine in the target process. The debugger runs in the debugger process, alongside the program being debugged.

JPDA consists of three layers:

- The Java Virtual Machine Tool Interface (JVMTI) [15].
- The Java Debug Wire Protocol (JDWP) [12].
- The Java Debug Interface (JDI) [10].

At the lowest level, the virtual machine provides JVMTI. JVMTI is a native (C/C++) programming interface. JVMTI is a two-way interface: it provides functions, which can be used to inspect the state and control the execution of programs running in the JVM, and callbacks, which notify a JVMTI client that an event, such as the creation of an object, has occurred. A JVMTI client is also called an agent. An agent runs in the same process as and communicates directly (by function calls) with the virtual machine which executes the program under study.

The back end uses JVMTI to inspect and control the execution of the program. It exposes its functionality by means of JDWP. JDWP defines the format of messages transferred

between the back end and the front end. It allows the program under study and the user interface of the debugger to run in different processes or even on different computers.

The front end communicates with the back end using JDWP. The front end implements the JDI. JDI is a Java programming interface, which allows the user interface of the debugger to be written in Java.

Sun's JVM implements JVMTI, JDWP and JDI. Other Java platforms might not implement all three interfaces, or might implement them only partially. For instance, a virtual machine might only implement JVMTI. Another platform might not implement JVMTI and JDWP, and implement JDI in its own way. If the target Java platform allows it, a programmer developing a debugger application can choose any of these three interfaces to communicate with the target JVM. Each approach has its own merits and drawbacks. These will be discussed in Section 3.2.3.

3.2.2 Bytecode Instrumentation

Bytecode Instrumentation (BCI) is a technique in which the bytecode of a program is modified in order to facilitate runtime analysis. At certain points in the bytecode, a special piece of bytecode, called instrumentation, is inserted. This instrumentation can perform different actions, depending on the type of analysis. For instance, a call to a certain method can be detected by inserting instrumentation at the start of this method's code. The instrumentation might register the call in a data structure or write information about the call to a file.

A major advantage of BCI is that its effect on the performance of a program can be low. Once inserted, the instrumentation bytecode is part of the original program's bytecode. The JVM can therefore run at full speed, optimizing not only the original code, but also the instrumentation. The actual effect on the program's performance depends on the efficiency of the instrumentation.

BCI is a powerful technique. In fact, it can be used to rewrite the whole program. If BCI is used to monitor an existing application, it would be undesirable to change the program in such a way that the program with BCI operates in a fundamentally different way than that same program without BCI. Therefore, special care should be taken such that the instrumentation does not change the functionality of the program.

BCI can be performed in three different ways. It can be done either statically, at load-time, or dynamically. Static BCI rewrites the on-disk class files. Load-time BCI redefines the class images as they are loaded. Dynamic BCI modifies an already loaded class. An advantage of static BCI is that it has less effect on the execution of the program than the alternatives, because the addition of instrumentation code happens before the execution, not during it. Also, the instrumentation has to be added only once, not every time the program runs. On the other hand, with static BCI the instrumentation code has to be added every time the program is recompiled, which is cumbersome. It also changes files, which might be an unwanted side effect. Load-time BCI and dynamic BCI are more convenient for the user. Load-time and dynamic BCI are not mutually exclusive: a class which was modified at load-time can be redefined afterwards.

3.2.3 Discussion

Initially, data collection for TraceVis was performed by using JDI. JDI is an easy to use API and allows the debugger or visualization tool to be completely written in Java. During this

project, two serious problems with JDI were encountered:

- Its performance is insufficient.
- It does not provide all the desired information, as stated in Section 3.1.

JDI is the top layer of JPDA. As such, all information collected from the JVM has to pass through all three layers of the JPDA. Each layer adds a bit of overhead. The result is that for a simple event like a method call, which normally takes little processor time, a significant amount of time is used just to transmit it to the JDI client. This situation would be acceptable if the amount of events is small, but many programs call thousands of methods each second.

In addition, JDI can retrieve some information only by suspending the target JVM, or certain threads. For instance, to determine the object which was called in the case of a method call, the current stack frame of the thread in which the call took place must be consulted. But the information in the stack of a thread is only valid if this thread is suspended. So, in this case the thread would need to be suspended on every method call. Again, this adds an unacceptably large overhead.

JDI does not provide all required data. In particular, it lacks information about garbage collection. This means that the amount of active objects cannot be determined using JDI.

To develop a traditional debugger, JDI seems to be a good choice. Traditional debuggers are often used to step through a program and inspect the program's state at each step. For this purpose relatively little data needs to be collected about the execution. Performance of the data collection is not as important, because the performance is limited by the programmer interacting with the user interface. Due to its performance problems, it is less suited to collecting large amounts of data.

Later in the project, JDI was replaced by JVMTI. It provides the data that JDI lacks and is the bottom layer of the JPDA, so it has the least overhead of the three interfaces. JVMTI is harder to use than JDI, because it is more low-level and a native API. While JVMTI provides enough information, BCI was used to speed up the most frequently occurring events, viz., method entry, method exit, and object allocation events.

The change to JVMTI and BCI was made before TraceVis was transformed from an online tool to an offline tool (see Section 2.2). As discussed previously, the performance of the data collection is important for an online visualization tool. For an offline visualization tool, it is less important. It is, however, desirable that the program under study is responsive to user input when a trace of its execution is being generated.

3.3 Data Collection Process

Figure 3.2 shows the components involved in the data collection. The program under study runs on a JVM, as usual. Data about its execution is collected by the data collector. The data collector is an agent library, a shared library which implements a JVMTI agent, as described in Section 3.2.3, and performs BCI on the executing program. The data collector writes a trace file[1]. This trace is read and visualized by the visualization tool.

[1]The format of trace files is described in Appendix B.

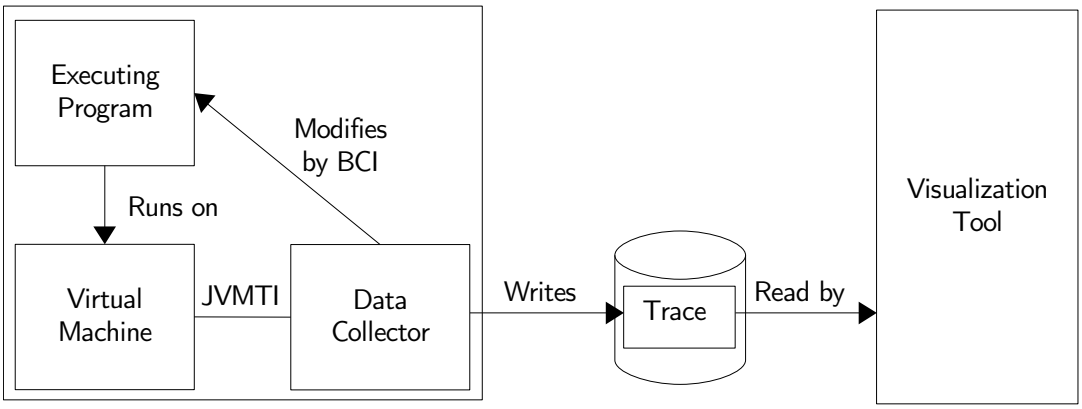


Figure 3.2: The data collection process, as described in Section 3.3.

3.4 Limitations

The current data collector works correctly and can collect all required information. Nonetheless, it has some limitations:

- With the current data collector, it may not be feasible to instrument all classes of a program. Even with the use of BCI, data collection imposes a considerable overhead on the execution of the program.
- Parts of the program (e.g., the standard library) can be excluded from instrumentation by specifying filters. The current data collector cannot detect calls from instrumented classes to uninstrumented classes, and vice versa.
- The data collector uses the `java_crw_demo` library[2] for BCI. It provides instrumentations for method entries and exits, and object allocations and is programmed in C, which simplifies integration with the data collector, which is also programmed in C. Unfortunately, extension of this library is not straightforward. For more complex BCI, more general libraries, such as BCEL [3] or ASM [2], are recommended.
- The current data collector can not be used to track calls to native methods, because this is not necessary to study programs which are completely written in Java. It should, however, be fairly straightforward to add this capability.

[2]The `java_crw_demo` library is part of the Java Development Kit [11].

Chapter 4

Visualization

The purpose of the visualization is to visualize the data model as described in Section 2.1 and to allow the user to find answers to the questions described in Section 1.1.1.

The data model has two aspects: a *structural* and a *time* aspect. The structural aspect consists of the call graph and the time aspect concerns the events the program generates. These two aspects could be combined in a single visualization. However, this might not be ideal for the following reasons. A graph can be complex. It is, therefore, desirable to have at least two dimensions to visualize it. Time can be visualized using another dimension. This would lead to a three-dimensional view. We expect that this visualization would not be useful, because multiple graphs (one for each point of time in the execution) overlaid on each other would lead to a cluttered view.

We, therefore, decided to visualize these two aspects using two separate visualizations, viz.,

- a structural view, and
- a time line view.

Section 4.1 describes the structural view and the time view is described in Section 4.2. To provide detailed information about classes and calls we decided to add a detail view, which is described in Section 4.3. Figure 4.1 provides an overview of the visualization tool.

4.1 Structural View

The purpose of the structural view is to visualize the call graph. The visualization of a graph has two aspects, viz.:

- visual representation, and
- layout.

The visual representation of a graph determines the appearance of the graph, i.e., what do its elements look like. Attributes of vertices and edges, such as weights, degrees, etc., can be encoded into their visual representations, by using different sizes, shapes or colors. Section 4.1.1 discusses the visual representation used in the structural view. A graph layout determines the positions of the vertices and the routing of the edges. The elements of the graph should be laid out in such a way that the user can quickly gain insight into the structure

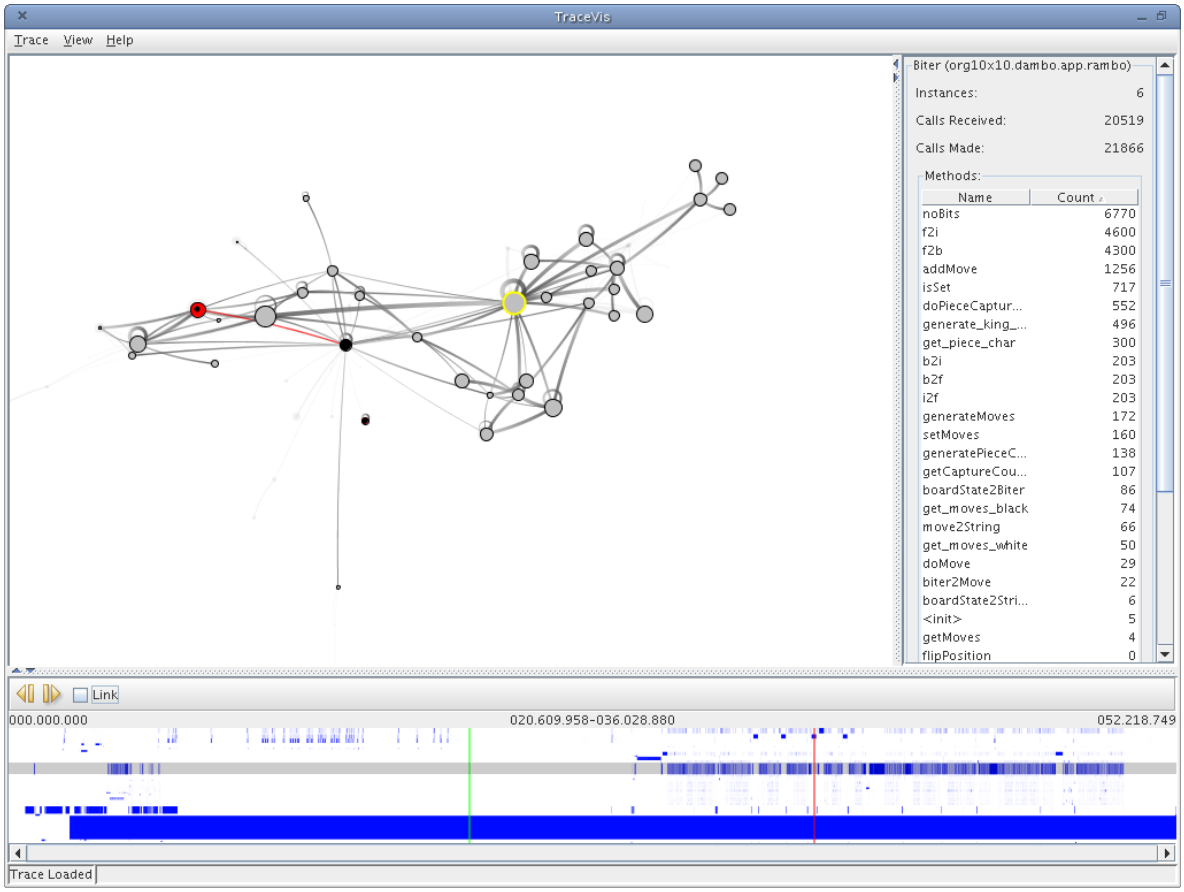


Figure 4.1: An overview of the visualization tool. The top left frame is the structural view, the top right frame is the detail view, and the bottom frame is the time view.

of the graph. A graph layout is critical to the readability of the graph. The graph layout method used in the structural view is described in Section 4.1.2.

4.1.1 Visual Representation

Figure 4.2 gives an overview of the concepts used in the visual representation of the call graph. Vertices are represented by circles. These circles can vary in two aspects to visualize attributes of the class represented by this vertex, viz.:

- size, and
- color.

The size of a vertex can visualize one of three attributes of the class represented by this vertex:

- the number of calls this class has received, or
- the number of call this class has sent, or
- the number of instances of this class.

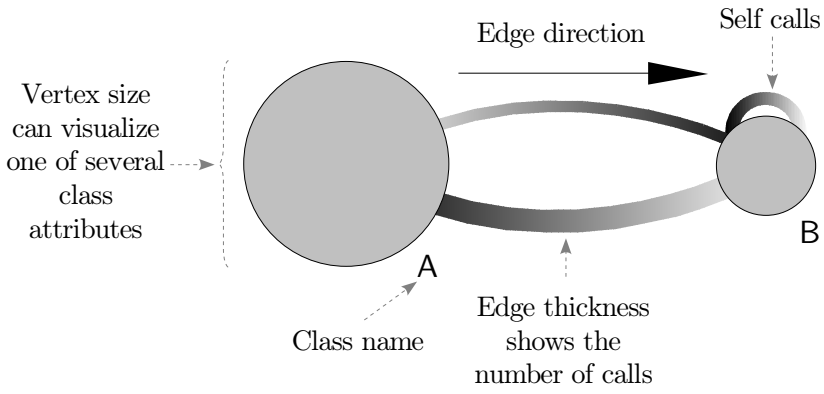


Figure 4.2: An overview of the visual representation of the call graph.

The user can specify which attribute is used. The color of a vertex can visualize different aspects, by means of two different color modes, viz.:

- a user-specified color, or
- the position of this class in one of the call stacks.

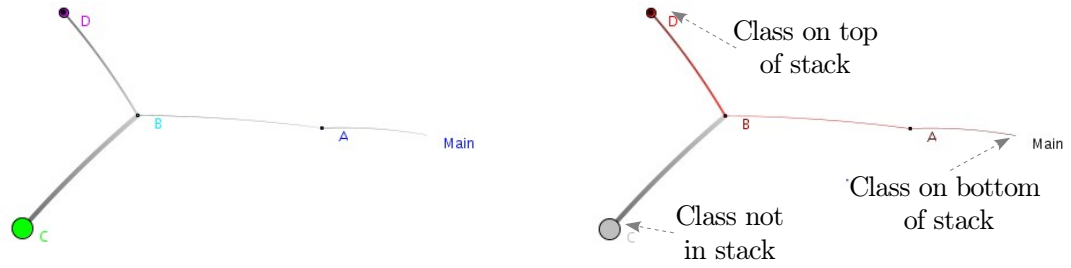
The user can specify the color of a vertex. This allows the user to mark certain classes for easier recognition, or group classes by assigning them the same color. Figure 4.3(a) shows an example of this color mode. For the stack-base color mode, if a class is not present in a call stack, its vertex is gray. If it is, its color is a shade of red. The class on top of the stack is bright red, classes which are lower on the stack are progressively a darker shade of red, until the class at the bottom is black. Furthermore, the vertex on top of the stack is marked by a filled black circle in its interior. Figure 4.3(b) shows an example of this color mode.

Edges are drawn from the center of its source to the center of its destination. In a call graph, edges are directed, and it is possible that it contains an edge from vertex A to vertex B, and vice versa. To prevent overlap, edges are slightly curved. To indicate direction, gradients are used. An edge is light at its source and turns dark gradually as it approaches its destination. The main attribute of an edge is the number of calls from its source class to its destination class. This attribute is visualized by the thickness of the curve. If vertices are colored according to their position in the stack, edges between two vertices whose classes are on the stack are colored with a gradient.

4.1.2 Layout

The layout of a graph is critical to its readability. A bad layout will be harder to interpret than a good layout. The quality of a graph layout is generally determined using *aesthetic criteria*, i.e., criteria which determine whether a graph looks good. Examples of aesthetic criteria are:

- Symmetry: if the graph contains symmetries, the layout of the graph should reflect that.
- Crossing minimization: if edges cross, it is harder to determine which vertices are connected. It might, therefore, be desirable to minimize edge crossings.



(a) Custom color mode.

(b) Stack based color mode.

Figure 4.3: The two color modes.

Because not much properties of call graphs were known in advance, it was hard to specify aesthetic criteria which were specific to call graphs. Instead, we used the following two aesthetic criteria, which are good criteria for all graphs:

1. Vertices should be uniformly distributed over the page (or screen).
2. Edges should have uniform lengths.

The first criterion reduces clutter caused by too many vertices packed too close together. The second criterion implies that adjacent vertices should be close.

As stated in the previous section, edges are drawn slightly curved. For layout purposes this is close enough to a straight line. In other words, the layout should be based on a visual representation with straight lines, a so-called straight-line embedding. For such an embedding, the routing of edges is trivial. So the layout problem reduces to the positioning of the vertices.

A popular class of graph layout methods for straight-line embeddings is the class of force-directed layout methods. Force-directed layout methods are suitable for general graphs and try to satisfy the two previously mentioned criteria. These methods generally represent the vertices and edges of a graph by a system of charged balls and springs, respectively. This system is modeled using the forces these objects exert on each other. An advantage of force-directed layouts is that it is straightforward to make adjustments to the forces in order to attain additional layout goals, i.e., force-directed layouts are adaptable.

The call graph changes during an execution. There are two approaches to deal with these changes. A static layout computes a layout which can be used for all instances of the call graph in advance. A dynamic layout updates its layout on every change of the graph.

The following sections give an introduction into force-directed layouts, a discussion of the extensions used in the structural view, and a discussion of static and dynamic layout methods, respectively.

Force-Directed Layout[1]

The family of force-directed layout methods is part of a family of graph layout methods which are based on physical analogies. Methods based on physical analogies compute a layout for a

[1]This section is based on Chapter 4 of [16], in which a more extensive introduction into force-directed layouts can be found.

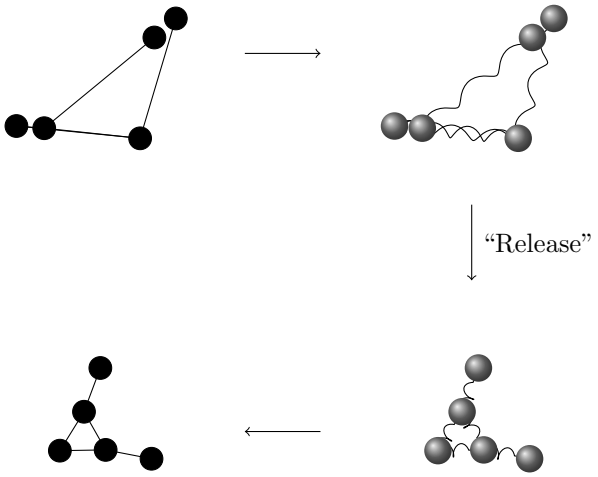


Figure 4.4: The spring analogy: a graph is modeled as a system of charged balls and springs. From a random initial configuration, this system is "released" to obtain a stable system. Translating this stable system back to a graph yields a nice graph layout.

graph by regarding it as a system of interacting objects. The assumption that underlies these methods is that configurations of these objects where the energy of the system is minimal correspond to readable layouts.

These methods generally consist of a *model* and an *algorithm*. The model describes the physical objects and the interactions between them. The algorithm computes an energy minimal configuration.

A popular method uses charged balls and springs to model vertices and edges respectively. Charged balls repel each other, which ensures that vertices are spaced evenly apart. Springs keep connected vertices close. The algorithm starts out with a random configuration of balls, and simulates the release of this system. Eventually, the system will reach a stable state, which should correspond to a nice layout. Figure 4.4 visualizes this method.

This model can be expressed either in terms of forces acting on the physical objects, or in terms of a potential energy reflecting the internal stress of the system. The first describes the family of layout methods known as force-directed layouts. The latter describes the family of energy-based layouts. In the sequel, only the force-directed layout methods, and in particular, the *spring embedder* by Eades [6], will be discussed.

Consider the undirected graph G, specified by a set of vertices V and a set of edges E. Let p be a placement (or layout), which associates a position p_v (a point in the plane[2]) with every vertex v from V. Denoted by $\|p_v - p_u\|$ is the Euclidean distance between positions p_u and p_v, and $\overrightarrow{p_u p_v}$ denotes the unit length vector $(p_v - p_u)/\|p_v - p_u\|$.

The spring embedder implements the analogy of charged balls and springs using two forces. It defines the repelling force

$$f_{rep}(p_u, p_v) = \frac{c_\rho}{\|p_v - p_u\|^2} \cdot \overrightarrow{p_u p_v}$$

between every pair of non-adjacent vertices $u, v \in V$, where c_ρ is a repulsion constant. Adjacent vertices are kept together by the springs between them. The spring embedder does

[2]Force-directed layout are not restricted to two-dimensional layouts. Adaptation of a force-directed layout method for one-dimensional, three-dimensional and higher-dimensional layouts is straightforward.

not use natural springs, which act according to Hooke's law, but logarithmic springs, which results in

$$f_{spring}(p_u, p_v) = c_\sigma \cdot \log\left(\frac{\|p_u - p_v\|}{l}\right) \cdot \overrightarrow{p_v p_u}$$

as the spring force. In this formula, l is the natural (or ideal) length of the spring, i.e., the length of the spring when no forces are applied to it. The more the actual length of the spring deviates from its natural length, the larger the force exerted on its adjacent vertices. The constant c_σ controls the strength of the spring. Stronger springs are harder to deform.

To compute a layout with this model, all vertices are assigned an initial position. These positions might be random, or derived from a heuristic. The algorithm then iteratively moves these positions to a more stable configuration. Each iteration moves each position p_v at the same time by $\delta \cdot F_v(t)$, where $F_v(t)$ is the *net force vector*, the sum of all repulsion and spring forces on v, and δ is a small constant which prevents excessive movements. This procedure is summarized in the algorithm ForceDirectedLayout.

Algorithm ForceDirectedLayout(G)
Input. An undirected graph $G = (V, E)$.
Output. A low-energy placement p of the vertices in G.

1. Create an initial placement p, by assigning all vertices v, $v \in V$, an initial position p_v.
2. **for** $t := 1$ **to** $ITERATIONS$
3. **for each** $v \in V$
4. $F_v(t) := \sum_{u:\{u,v\} \notin E} f_{rep}(p_u, p_v) \quad + \sum_{u:\{u,v\} \in E} f_{spring}(p_u, p_v)$
5. **for each** $v \in V$
6. $p_v := p_v + \delta \cdot F_v(t)$
7. **return** p

Because each iteration computes the interaction between every pair of vertices, each iteration of this algorithm takes $\mathcal{O}(|V|^2)$ time.

Extensions

As discussed previously, force-directed layout methods are flexible. It is easy to modify or add force functions in order to express different layout requirements. The structural view uses a force-directed layout model which is based on the original spring embedder.

One feature which was added to the original model is the ability to handle vertices of non-uniform sizes. As described in Section 4.1.1, the size of a vertex is used to express properties of the class it represents, such as the number of instances or the number of received calls. The spring embedder model does not handle non-uniform vertices well, because the charged balls it uses to model vertices are considered to be infinitely small points. A drawing of a graph which was laid out by a spring embedder, and drawn with varying vertex sizes, might not look nice, because larger vertices might overlap other vertices. This problem can be solved in a straightforward way by using the shortest distance between the boundaries of the vertices, and not between their centers, in the force functions. For circular vertices which do not overlap this boils down to subtracting the radii from the distance between the centers. For overlapping vertices the distance between their boundaries is taken to be zero. Or more

formally, for circular vertices u and v, with radii r_u and r_v, and center points p_u and p_v respectively, the distance $\text{dist}(p_u, p_v)$ between their boundaries is:

$$\text{dist}(p_u, p_v) = \begin{cases} 0 & \text{if } \|p_u - p_v\| \leq r_u - r_v \\ \|p_u - p_v\| - r_u - r_v & \text{if } \|p_u - p_v\| > r_u - r_v \end{cases}$$

The spring force was also modified in two other aspects. First, it was observed that high-degree vertices are subject to many different forces. This causes those vertices to jump around wildly. Therefore, to dampen these movements, the force a spring exerts on a vertex is divided by the degree of the vertex. Second, the springs are natural (linear) springs, instead of the logarithmic springs used by Eades. This yields the spring force as it was used in the structural view:

$$f_{spring}(p_u, p_v) = c_\sigma \cdot \frac{l - \text{dist}(p_u, p_v)}{\deg(v)} \cdot \overrightarrow{p_v p_u} \ , \tag{4.1}$$

where $\deg(v)$ is the degree of vertex v. In the repelling force only the notion of distance has changed:

$$f_{rep}(p_u, p_v) = \frac{c_\rho}{\max(\epsilon, \text{dist}(p_v, p_u))^2} \cdot \overrightarrow{p_u p_v} \ , \tag{4.2}$$

where ϵ is a small constant to prevent division by zero.

The algorithm is based on the algorithm ForceDirectedLayout, as described in the previous section. It was slightly modified in order to provide better convergence. As is mentioned by Harel and Koren [9], the convergence of the system becomes much slower when dealing with sized vertices instead of points. The reason is that large vertices reduce the maneuvering space which is needed to converge. It is, therefore, more likely that the system will get stuck in a local optimum. Harel and Koren propose a simple solution to this problem. The system will converge much faster if the vertices are considered to be zero-sized in the first few iterations. With every following iteration, the vertices are scaled up a bit, until, in the last few iterations, the vertices are at their intended size. Our experience is that this modified algorithm indeed provides faster convergence.

This force model could be extended in many more ways. Two extensions are particularly interesting, but not implemented, viz.,

- layout using directed edges, and
- layout using weighted edges.

Layout using directed edges means that the direction of edges (in case of a directed graph) is taken into account when creating a layout. For instance, it may be desirable to let all edges point downwards or outwards from the center. Sugiyama and Misue [20] introduced the concept of "magnetic" springs, i.e., magnetically charged springs which point in the direction of the surrounding magnetic field, like the needle of a compass. One problem is that it interferes with other layout goals. Another problem with the magnetic springs method is that it does not deal well with cycles in the graph. It is not possible to lay out a cycle with all edges pointing in the same direction. Hence, layout cannot be used as the sole cue for direction. We, therefore, decided not to use magnetic edges and to encode the direction of an edge in its visual representation, as has been described in Section 4.1.1.

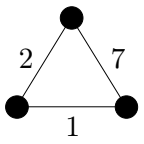


Figure 4.5: A weighted graph which does not satisfy the triangle inequality.

The call graph is an edge-weighted graph, i.e., its edges have attributes, or weights, assigned to them. An intuitive way to visualize these weights would be to lay out the graph's vertices such that the length of the edges between them is proportional to their weights. For instance, if a certain class frequently calls another class, these classes would be close together in such a layout. Unfortunately, this approach has several problems. First, if the weights change during visualization, the layout will have to be changed as well, so a dynamic graph layout is necessary. Second, not all weighted graphs can be laid out such that the edge lengths are proportional to their weights. For instance, graphs of which the weights do not satisfy the triangulation inequality, such as the graph depicted in Figure 4.5, can not be laid out such that the edge lengths are proportional to their weights. Such a layout can only be approximated. Due to these problems, we decided not to use a weighted edge layout, but to encode an edge's weight by means of its visual representation, as has been described in Section 4.1.1.

Dynamic and Static Layout

The call graph is a graph which changes during the execution of a program, i.e., there are multiple instances of the call graph, of which each corresponds to a single point in time of the execution. One way to compute a layout for these graphs is to use a dynamic layout. This means that first an initial layout is computed. This layout is then modified for every instance.

Another approach is to use a static layout: the graph layout is computed in advance, and never changed during the use of the visualization. In general, such a layout can be computed by computing a layout for a graph which contains all instances of the graph. In the case of the call graph, this graph is determined easily. During the execution, the call graph is extended due to class load and method calls. No elements are removed from the call graph. Hence, the instance of the call graph at the end of the execution contains all elements from previous instances.

A static layout does not provide an ideal layout for every situation, see for instance Figure 4.6. However, a static layout is stable, i.e., during the use of the visualization, vertices and edges do not move. This allows the user to build up a so-called "mental map", which links the identities of vertices (in the case of the call graph, their names) with their positions. This mental map can be facilitated even further by drawing a "background graph", i.e., by drawing elements which are not part of the current graph vaguely in the background. Another advantage of a static layout is that once the layout is computed, no resources are needed to continually update the layout, which improves responsiveness.

A dynamic layout can adapt itself to a changing graph. Hence, the layout may be better suited to the current graph (see Figure 4.6). However, to obtain this better layout, the vertices and edges may have to change positions. These changes are not just local, i.e., they only move neighboring vertices, but global, i.e., all vertices in the graph may move. The user, therefore, needs to continually adapt his mental map. This can be remedied somewhat by animating the

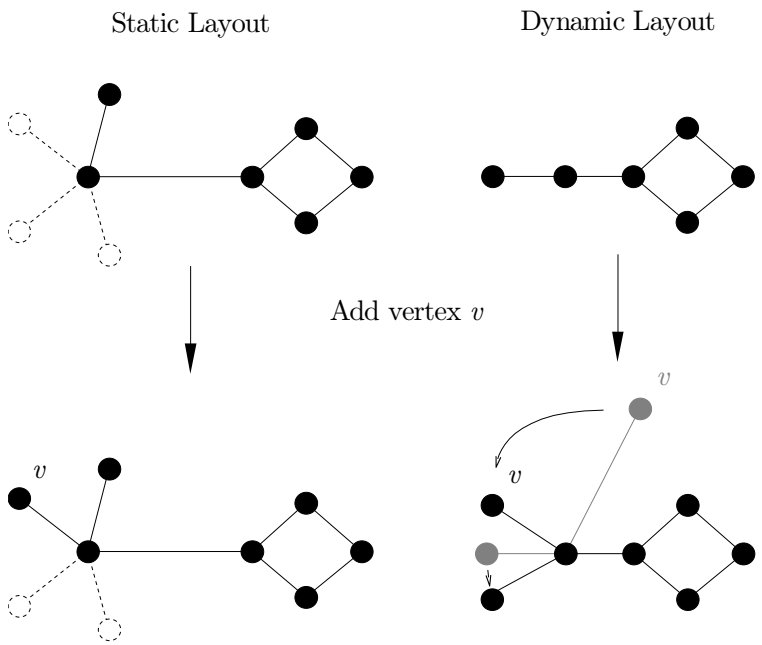


Figure 4.6: A comparison of static and dynamic layouts. The static layout is determined in advance for the whole graph (black and stippled vertices and edges). The layout for a subgraph (only the black vertices and edges) is based on this layout. When a vertex is added, the graph layout remains stable. With a dynamic layout, a new vertex is placed in a random position, and subsequently moved to a stable position. Other vertices move to accommodate this new vertex.

transition from one graph to another. However, animation may be ineffective if the changes between the two graphs are large.

We tried both approaches and found that the stable view provided by a static layout is better suited to display the call graph.

4.2 Time Line View

The time line view visualizes the time-related aspects of the trace. It can be used to pinpoint interesting points in the execution. The time line view can visualize different time-related aspects. In TraceVis, two different time line views, visualizing two different aspects, were implemented, viz.:

- the activity view, which visualizes the activity of classes, described in Section 4.2.1, and
- the instance view, which visualizes the number of instances of classes during the execution, described in Section 4.2.2.

4.2.1 Activity View

The activity view provides a compact view on the activity of classes. A class is considered to be active when it is on top of a thread's call stack, i.e., a thread is executing a method from this class.

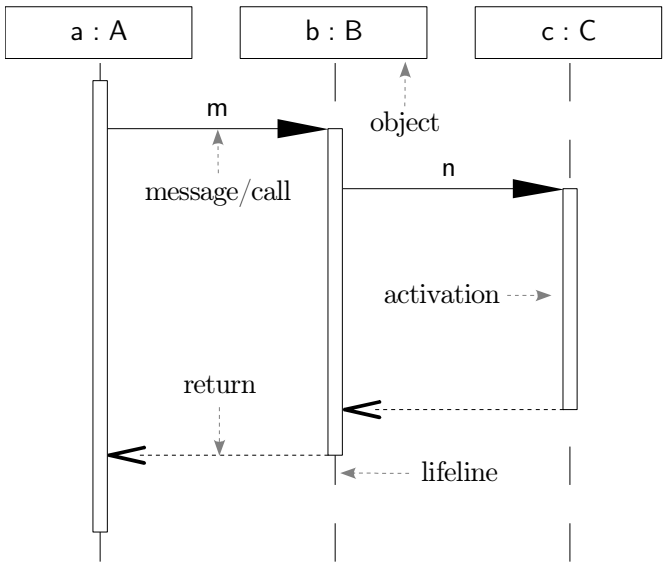


Figure 4.7: An example UML sequence diagram. In this example, an object `a` of class `A` calls the method `m` on an object `b` of class B, which in turn calls the method `n` an object c of class C. In a sequence diagram, each object has a *lifeline*, which indicates the life time of the object. On this lifeline are placed so-called *activations*, which indicate when the object is active in the interaction.

The activity view was inspired by UML's sequence diagrams. A UML sequence diagram visualizes the collaboration of objects. Figure 4.7 gives an example of a sequence diagram. A sequence diagram is a two-dimensional diagram in which both dimensions have a distinct function. The horizontal axis is used to display a row with the objects which participate in the collaboration. The vertical axis is used to visualize time. One advantage of sequence diagrams is that its notation is intuitive. It is beyond the focus of this thesis to fully explain sequence diagrams. A more complete introduction can be found in one of the numerous tutorials, such as the one by Fowler [7, Chapter 4].

Sequence diagrams were designed to model specific behavior of a small set of objects. For larger sets of objects and messages, the diagram will become cluttered. Furthermore, their visual language is meant for drawing on a whiteboard. As such, it is not very compact and does not use visual cues which are hard to draw by hand.

To make the sequence diagram more suitable to large traces, we made the following modifications:

- Instead of individual objects, the activity view displays classes. In most programs, the amount of classes is much smaller than the amount of objects.

- Not all calls and returns are shown at once, because this would lead to a cluttered view. Calls and returns are only visualized after interaction, as will be discussed in 4.4.1.

- The definition of activation was changed: in sequence diagrams, a participant is active when it is on the call stack; in the activity view, a participant is active only when it is on top of the call stack.

- The notation was made more compact.

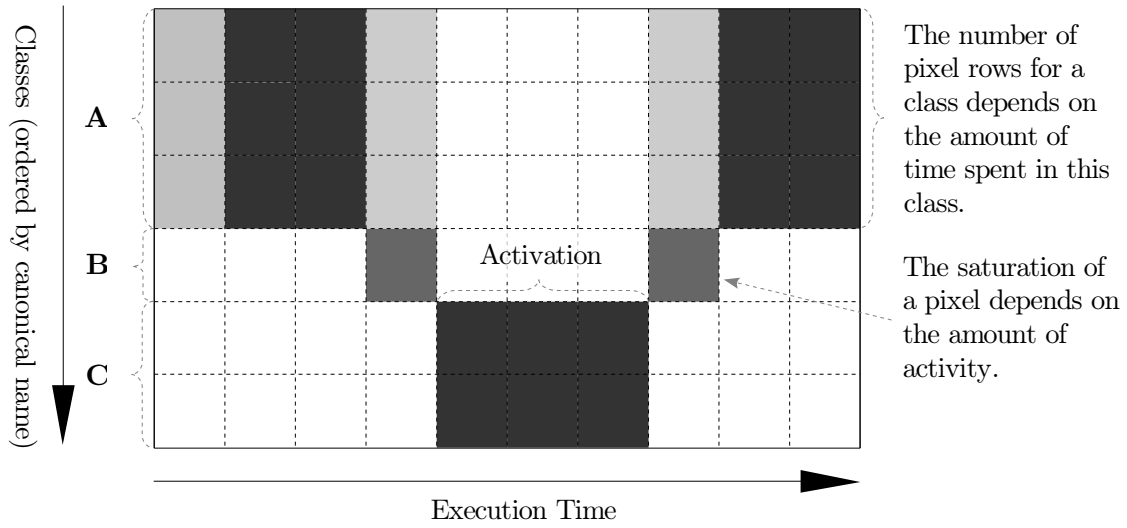


Figure 4.8: A conceptual drawing of the activity view. On the horizontal axis is the execution time. The vertical axis is a row of classes. This drawing depicts the following situation: first class `A` is active, then `B`, and then `C`. After the activation of `C`, `B` becomes active again, and finally `C` resumes activity.

In addition, the functions of the axes were interchanged, to make the activity view fit in better with other visualizations. So, in the activity view, the vertical axis is a row of classes and the horizontal axis visualizes time. Note that this change is merely a practical one, the activity view will work well in both configurations.

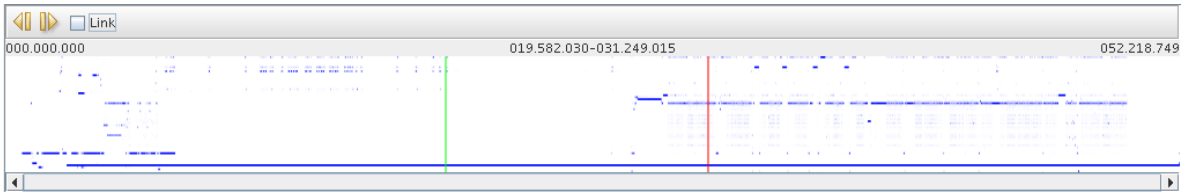
The notation was made more compact primarily by using just a few pixel rows for each class. Each row displays the activations of the corresponding class. Figure 4.8 provides a conceptual drawing of this notation. The number of activations in an execution is typically much larger than the number of activations depicted in a sequence diagram. Moreover, the length of of activations can vary from a few microseconds spent in, e.g., an accessor method, to minutes spent in a long-running computation or even longer. Therefore, it is impractical to draw each activation separately. On the horizontal axis, each pixel represents a time range. The saturation of the pixel summarizes the activity of the class in this time range. A desaturated color (close to white) means that the class was barely active during this time range. A fully saturated color means that the class has been active during the full time range. This approach allows for the drawing of both short- and long-running activations.

The exact amount of pixel rows assigned to a class, i.e., its height, is determined by its amount of activity. Classes which are more active, i.e., classes which are active for a long time, are higher than less active classes. This emphasizes the more active classes. More formally, the height assigned to a class depends on the ratio between the amount time this class was active during the execution and the total execution time, the so-called *activity ratio*. The activity ratio a_c for a class c is defined as:

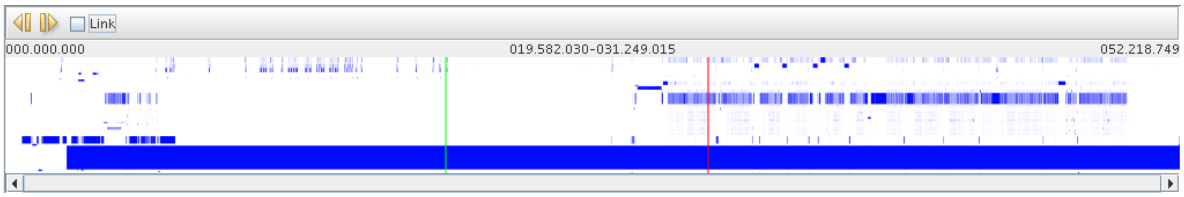
$$a_c = \frac{\text{time spent in class } c}{\text{total execution time}}$$

The height h_c assigned to a class is:

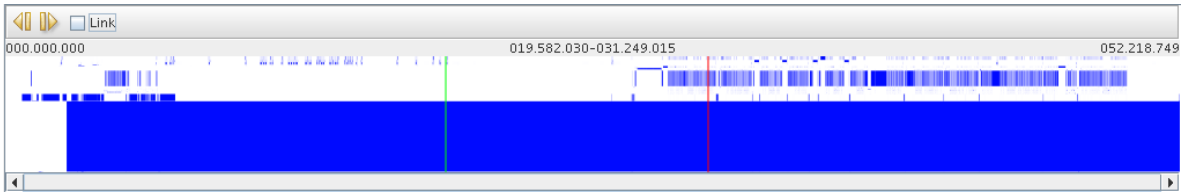
$$h_c = \frac{a_c^\beta}{\sum_{c \in C} a_c^\beta} \cdot H \ , \tag{4.3}$$



(a) $\beta = 0.0$



(b) $\beta = 0.5$ (default value)



(c) $\beta = 1.0$

Figure 4.9: The effect of the activity exponent β on the activity view. All three screen shots show the same trace.

where C is the set of classes, H is the height of the view, and β, $0 \leq \beta \leq 1$, is the activity exponent. The activity exponent determines the strength of the effect. If $\beta = 0$, all classes will be assigned the same height. If $\beta = 1$, the height of the class is linearly dependent on its activity ratio. Figure 4.9 shows the effect of the activity exponent on the activity view.

The activity view also displays the metric start time and the current time, as described in Section 2.1.2.

4.2.2 Instance View

The instance view provides a compact view on the amount of instances each class has during the execution. The instance view is similar to the activity view. The vertical axis is a row of classes and the horizontal axis visualizes time. However, the saturation of a pixel does not visualize the activity of the corresponding class in the corresponding time range, but the amount of instances of this class in this time range. Figure 4.10 gives an example of this view. Unlike the activity view, all classes have the same height in the instance view.

4.3 Detail View

The detail view is a simple textual view, which provides detailed information on specific classes or calls. The detail view can contain two kinds of views, viz.:

- class details, and

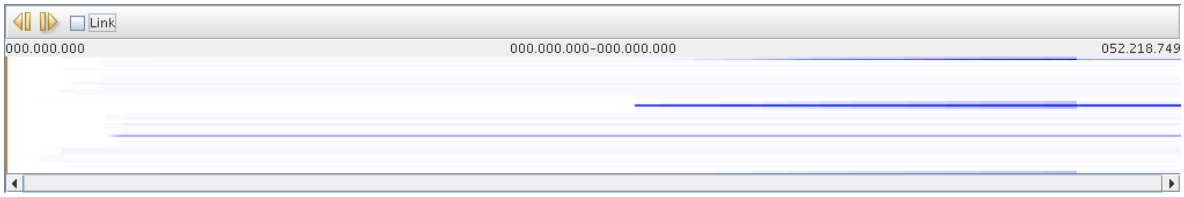


Figure 4.10: An example instance view.

- calls details.

The class details view, as shown in Figure 4.11(a), shows information about a specific class. It shows the following attributes:

- the number of calls this class has received,
- the number of calls this class has sent,
- the number of instances, and
- the number of times each of its methods have been called,

which have been defined in Section 2.1. The calls details view, as shown in Figure 4.11(b), shows information about calls from the source class to the destination class. It shows the following attributes:

- the source of the calls,
- the destination of the calls,
- the total number of calls from the source to the destination, and
- the number of times each method from the destination class has been called by the source class,

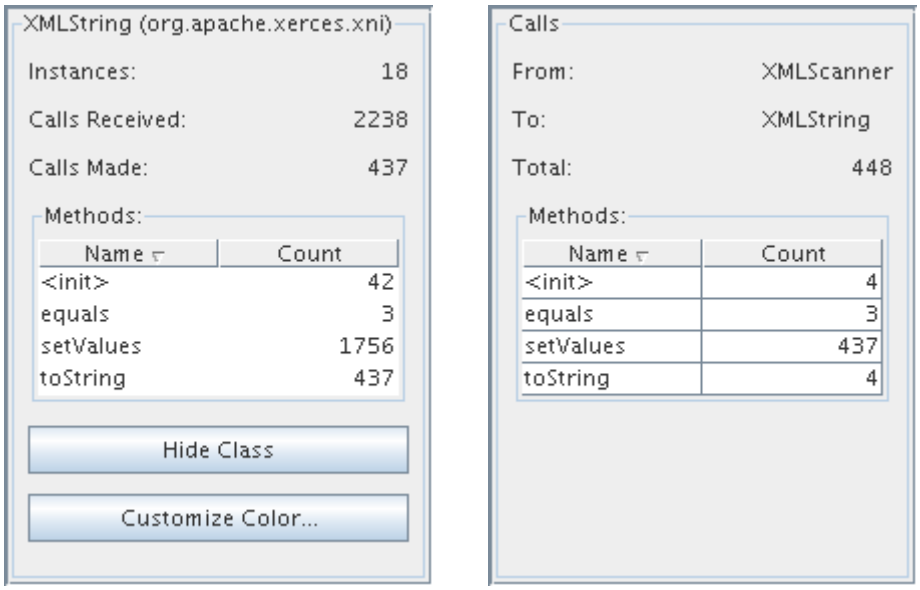
which have been defined in Section 2.1.

The values of the attributes shown in both views depend on both the subrange of the execution and the filters selected by the user, as has been discussed in Section 2.1.

4.4 Interaction

The views described above are not static entities. A user can interact with them to retrieve additional information. The views can be manipulated in the following ways:

- vertices and edges can be selected in order to show additional information about them,
- the structural and time line views can be zoomed and panned to focus on specific areas of interest,
- the metric start time and the current time can be changed,
- the colors of classes can be customized to highlight certain classes,



(a) The class details view. (b) The calls details view.

Figure 4.11: Detail views.

- the trace can be filtered to highlight certain parts of the trace, and
- the views can be configured.

The following sections will describe each of these interactions.

4.4.1 Selection

The structural view and the time line view both support selection. Both views support two kinds of selection, viz.:

- *temporary* selection, and
- *permanent* selection.

Temporary selection is initiated by moving the mouse cursor over (*brushing*) an element of the view. The selection is undone when the mouse cursor leaves the element. The result of temporary selection depends on the view.

Permanent selection is initiated by clicking on (*picking*) an element of the view. The selection is undone when another element is clicked on. The result of a permanent selection is that the selected element is highlighted and shown in the detail view.

In TraceVis, two kinds of elements can be selected, viz.:

- classes, and
- calls.

The following two sections discuss selection of these two kinds.

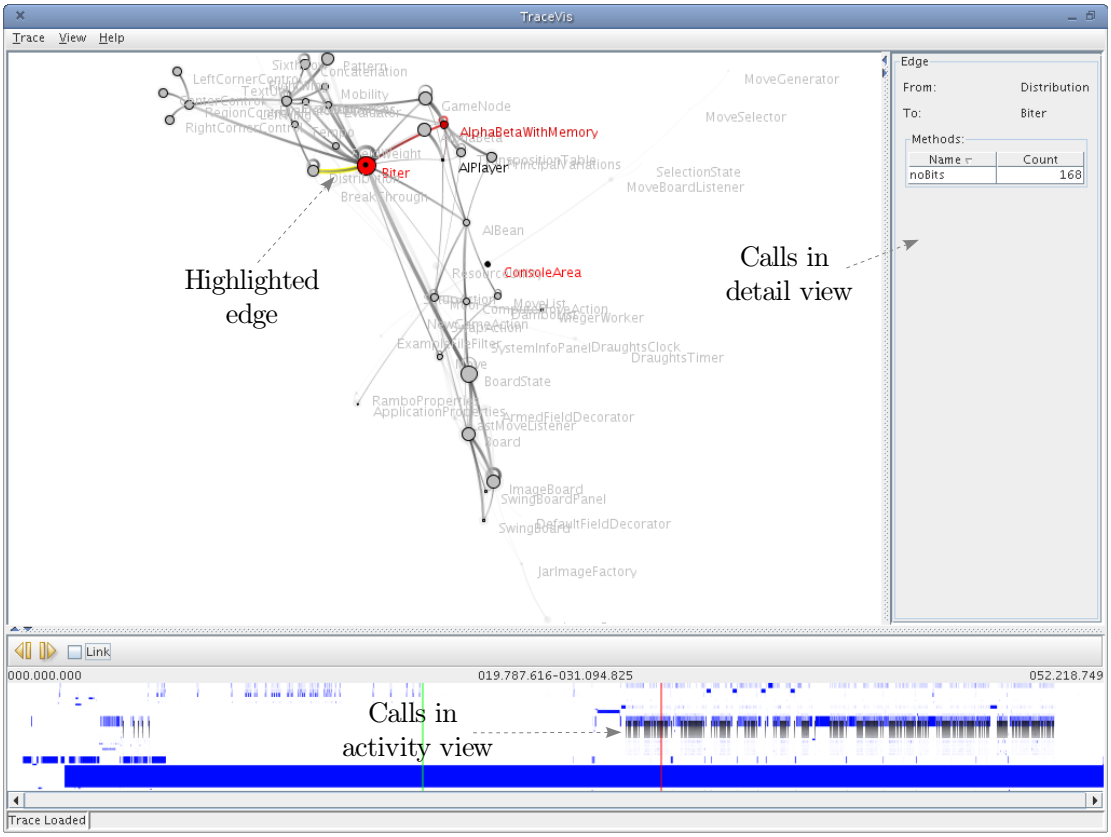


Figure 4.12: The selection of calls. Upon selection of an edge, three things happen: (1) the selected edge is highlighted, (2) detailed information about the edge, and (3) the individual calls are visualized in the activity view.

Selecting Classes

In the structural view, a class can be selected by brushing or picking the vertex which represents this class. In the time line view, a class can be selected by brushing or picking the row which represents this class.

The result of a selection is that the class is highlighted in both views. If the class was temporarily selected in one of the views, a tool tip with the full class name of the selected class, will appear near the mouse cursor. If the class was permanently selected, the detail view is updated to show information about this class.

Selecting Calls

Calls can only be selected in the structural view, by clicking on the edge which represents these calls.

As is shown in Figure 4.12, the structural view visualizes this selection by highlighting the selected edge. The activity view shows all calls and returns represented by this edge by drawing a gradient line from the calling class to the called class. The detail view shows detailed information about this edge.

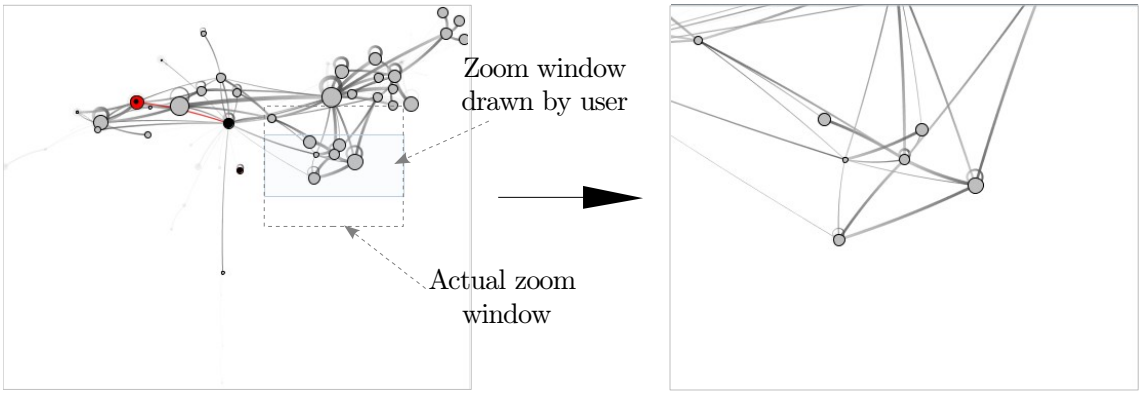


Figure 4.13: Zooming in the structural view. On the left, the user draws a zoom window around an area he is interested in. To prevent distortions in the view, the zoom window is corrected such that its aspect ratio matches the aspect ratio of the view. The resulting view is shown on the right. Note that the vertices have the same sizes in both views.

4.4.2 Zooming and Panning

The structural view and the time line view both support zooming and panning.

Zooming and panning are operations on a *world-coordinate window*. The world-coordinate window specifies which part of the "world" is shown. For TraceVis, this world is either the graph from the structural view, or the execution time from the time line view. Note that this window is not the same as a *window-manager window*. In this section, only world-coordinate windows will be discussed, so, in the sequel, world-coordinate windows will be referred to as windows. *Zooming* means that the size of the window is changed. Zooming in reduces the size of the window, and zooming out increases the size of the window. Zooming in can be used to view details, while zooming out can be used to obtain an overview. *Panning* means that the window is moved up, down, left, or right. Panning can be used to view areas which are neighbor to the current window.

Both views use the same concepts, but implement them differently enough to warrant separate discussion, as will be done in the following two sections.

Structural View

In the structural view, the world is the plane on which the graph layout lies. By zooming and panning, the user can focus on specific areas of interest. Zooming only scales the layout, not the visual representations of the vertices. This means that a specific vertex size represents the same value from all possible view points. Moreover, graphs are more readable when vertices do not take up most of the space.

The user can zoom in on the view using one of two methods, viz.:

- by drawing a window around an area of interest with the right mouse button, or
- by turning the scroll wheel forward.

The result of drawing a window around an area is that the view is constrained to this window. To prevent distortion of the view, this window is first corrected such that its aspect ratio[3]

[3]Aspect ratio: $width : height$, the ratio between the width and the height of a rectangle.

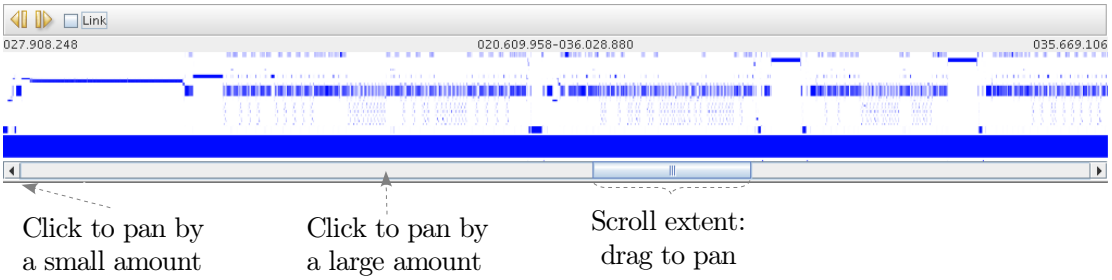


Figure 4.14: Panning in the time line view. The scroll bar at the bottom can be used to pan the time line view.

matches the aspect ratio of the view. Figure 4.13 shows how this is done. Turning the scroll wheel forwards (i.e., away from the user) zooms in the view around the current position of the mouse. The user can zoom out by turning the scroll wheel backwards (i.e., towards the user).

The view can be panned by clicking on a spot in the view which is not covered by a graph or an edge with the left mouse button, and dragging the mouse.

If the user wishes to return to the initial view (a view of the complete call graph), he can double click with the right mouse button to restore it.

Time Line View

In the time line view, the window can be adjusted in only one dimension, viz., the time dimension. This means that the view always shows all classes in the trace, but that the user can select time ranges in the trace to examine in more detail.

As with the structural view, the user can zoom in on the view using one of two methods, viz.:

- by drawing a window around an area of interest with the right mouse button, or
- by turning the scroll wheel forward.

The effects of these actions are the same as in the structural view. The view can also be zoomed out by turning the scroll wheel backward. Double clicking with the right mouse button restores the window to cover the whole trace.

In the time line view, it would be awkward to use dragging as a means to pan the view, because every pixel is occupied by a class. Therefore, panning the time line view is performed by another means, viz., a scroll bar. Figure 4.14 shows the time line with the scroll bar at the bottom. The view can be panned by dragging the extent, by clicking on one of the small arrows on the sides, or by clicking on the track. Moreover, the size and position of the extent is an indication of the position of the window in relation to the complete trace. This indication works well when the window covers a significant amount of the trace, but less so when the window is small.

4.4.3 Changing Times

Using the time line view, the user can change the metric start time and the current time. Figure 4.15 gives an overview of the relevant user interface elements. The user can perform

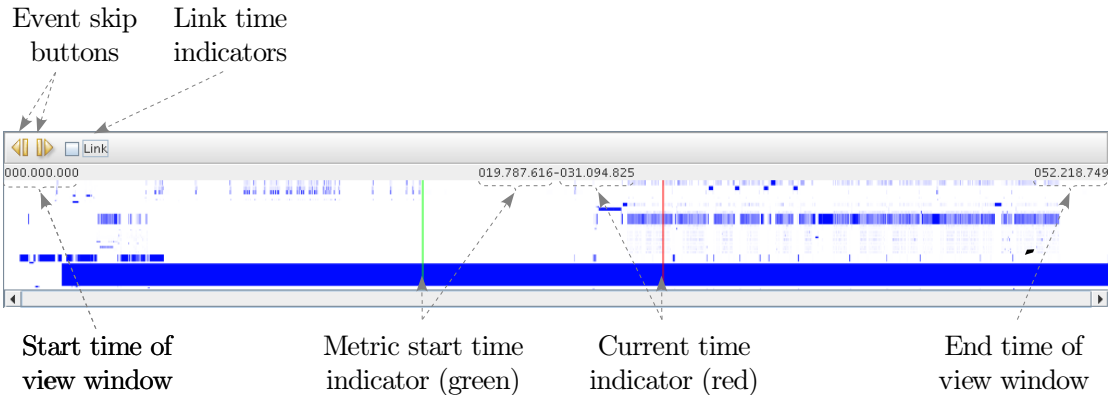


Figure 4.15: Changing times using the time line view.

these actions in one of the following ways:

- by moving the indicators for the metric start time and the current time, or
- by clicking on one of the event skip buttons.

Dragging one of the time indicators (using the left mouse button) results in a change in the corresponding time in the model. Other views are updated due to this change. By default, the time indicators move separately, that is, if the user moves the current time indicator, the metric start time indicator does not move. However, the time indicators can be "linked", by clicking on the check box labeled "Link". This way, when the current time indicator is dragged, the metric start time indicator will follow to keep the difference in time between them the same. Due to zooming and panning, as described in the previous section, the user may lose sight of the time indicators. To remedy this, the current time indicator can be brought to the position under the mouse cursor by clicking with the middle mouse button, and the metric start time indicator can be brought to the same position by clicking with the middle mouse button while holding the control button on the keyboard. In combination with the zooming capability, the time indicators allow the user to select any time range in the execution.

Clicking on one of the event skip buttons moves the current time to either the previous or the next event. This has the same effects as changing the current time using the indicator. In addition, if this event is a method entry or method exit event, the transition from the old to the new state is animated in the structural view.

4.4.4 Class Coloring

The user can highlight certain classes or packages by assigning them a custom color. For the structural view to show user selected colors, the user must select the custom color mode, which has been discussed in Section 4.1.1. How to select this color mode will be discussed in Section 4.4.6. When a class is selected, this action can be performed by clicking on the button labeled "Customize Color...", upon which the color customization dialog, as shown in Figure 4.16, will appear. This dialog allows the user to configure the color of the class itself, and of all classes belonging to the same package as the selected class.

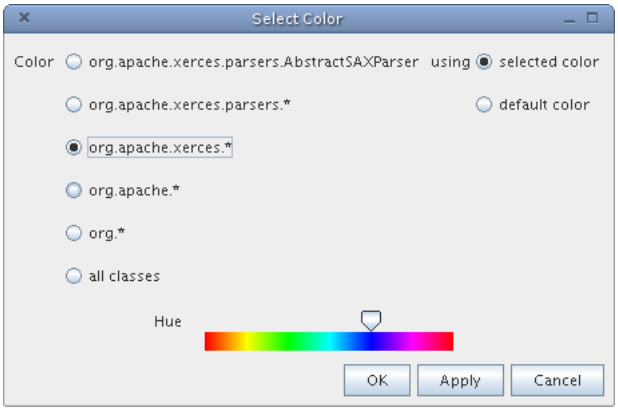
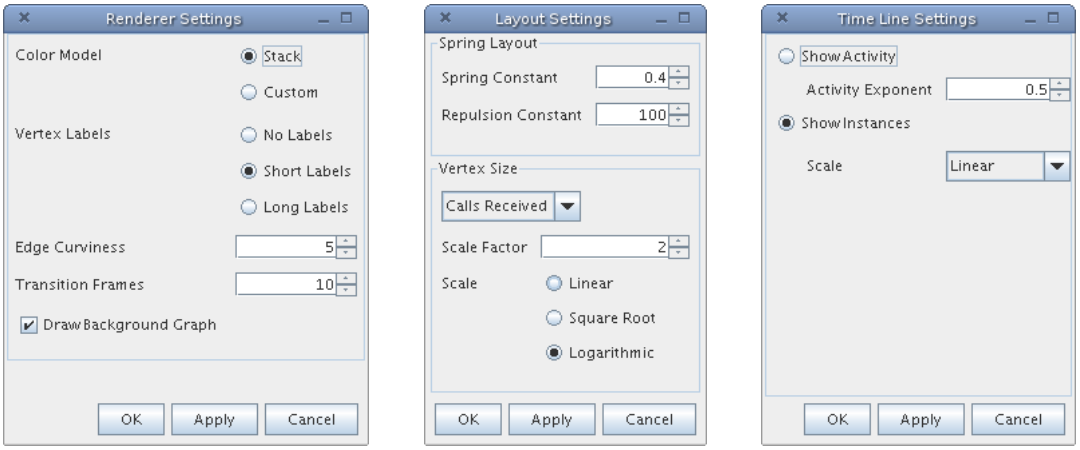


Figure 4.16: The color customization dialog allows the user to assign colors to classes. On the left, the user can choose whether to assign a color to just the selected class, to all classes of any of its enclosing packages, or to all classes. At the bottom, a hue (color) can be selected.



(a) Renderer configuration. (b) Layout configuration. (c) Time line configuration.

Figure 4.17: Configuration dialogs.

4.4.5 Filtering

By using filtering, the user can restrict his attention to interesting classes, as described in Section 2.1.2. A class can be filtered out (hidden) by selecting it, and clicking on the button labeled “Hide class” in the detail view. A class can be filtered in (shown) by selecting it, and clicking on the button labeled “Show class” in the detail view. The filter can be reset by selecting the menu item labeled “Reset Filter” under the menu labeled “Trace”. Upon each of these actions the views will be updated to reflect the new situation.

4.4.6 Configuration

The structural view and the time line view can be configured in order to better suit the user’s wishes. All configuration is performed by means of configuration dialogs, which are accessible under the menu “View”. The following two sections will discuss the configuration options for both views.

Structural View

The structural view can be configured in two aspects, viz.:

1. in its visual representation, and
2. in its layout.

The first aspect can be configured using the "Renderer Settings" dialog (shown in Figure 4.17(a)), which can be found under the aforementioned "View" menu. In this dialog the following parameters can be configured:

- Color model: the colors of vertices and edges can be based on either the stack or on custom colors configured by the user, as described in Section 4.1.1.
- Vertex labels: allows the user to choose between no vertex labels, short vertex labels, which shows just the simple name of the represented class, and long vertex names, which shows the canonical name of the represented class.
- Edge curviness: how curved the edges should be drawn, 0 yields straight edges, higher yield more curved edges.
- Transition frames: determines the amount of animation frames which is drawn for each transition. Higher values yield smoother, but slower, animation.
- Draw background graph: determines whether to draw the background graph.

The second aspect can be configured using the "Layout Settings" dialog (shown in Figure 4.17(b)), which can be found under the aforementioned "View" menu. The layout settings dialog allows the user to configure the spring constant, that is, the constant c_σ in the spring function

$$f_{spring}(p_u, p_v) = c_\sigma \cdot \frac{l - \text{dist}(p_u, p_v)}{\deg(v)} \cdot \overrightarrow{p_v p_u} \ , \tag{4.1}$$

which was first described in Section 4.1.2. The repulsion constant, i.e., the constant c_ρ in the repulsion function

$$f_{rep}(p_u, p_v) = \frac{c_\rho}{\max(\epsilon, \text{dist}(p_v, p_u))^2} \cdot \overrightarrow{p_u p_v} \ , \tag{4.2}$$

described in Section 4.1.2, can also be configured. As described in Section 4.1.1, the size of a vertex can depend on one of three class attributes, viz.

1. the number of calls this class has received,
2. the number of calls this class has sent, and
3. the number of instances this class currently has.

Which of these attributes is used can be selected in the "Vertex Size" panel. For each of these attributes two things can be configured: the scale factor and the scale. The scale factor determines how much the vertex is scaled. The scale can be a linear scale, a square root scale, or a logarithmic scale.

Time Line View

The time line view can be configured using the "Time Line Settings" dialog (shown in Figure 4.17(c)), which can be found under the aforementioned "View" menu. Using this dialog, the time line view can be switched between the activity view and instance view.

For the activity view, the activity exponent, as described in Section 4.2.1, that is the parameter β in

$$h_c = \frac{a_c^\beta}{\sum_{c \in C} a_c^\beta} \cdot H \ , \tag{4.3}$$

can be configured.

For the instance view, the user can choose between three scales: a linear scale, a square root scale, and a logarithmic scale.

Chapter 5

Tool Evaluation

To determine the usefulness of TraceVis, it was used to gain insight into the execution traces of a few programs, and it was compared to other tools which visualize execution traces.

5.1 Test Cases

For the following test cases, traces were generated from two programs,

1. Dambo, a draughts simulator, developed by Wieger Wesselink and Huub van de Wetering, and
2. Ant [1], a software build tool, developed by the Apache Software Foundation.

The following two sections will discuss the observations which were made by studying these traces.

5.1.1 Dambo

Dambo is a draughts simulator. In Dambo, two players can play a game of draughts. Either of these players can be either a human or the computer. A computer player determines its next move by evaluating possible situations ahead, up to a certain number of moves. The computer then picks the move which receives the best evaluation. The computer player's strength depends on the number of moves it evaluates ahead and the type of evaluation it performs for each of the situations created by these moves.

The execution trace discussed in this section covers the initialization of the program, and a few moves from a game between two computer players. Classes have been colored in the following way:

- The evaluator classes have been colored red.
- `Biter`, an important utility class, has been colored pink.
- Other back end classes have been colored purple.
- `Mooi`, the main class, has been colored light blue.
- User interface classes have been colored green.

The trace contains 58 active classes and nearly 380.000 events.

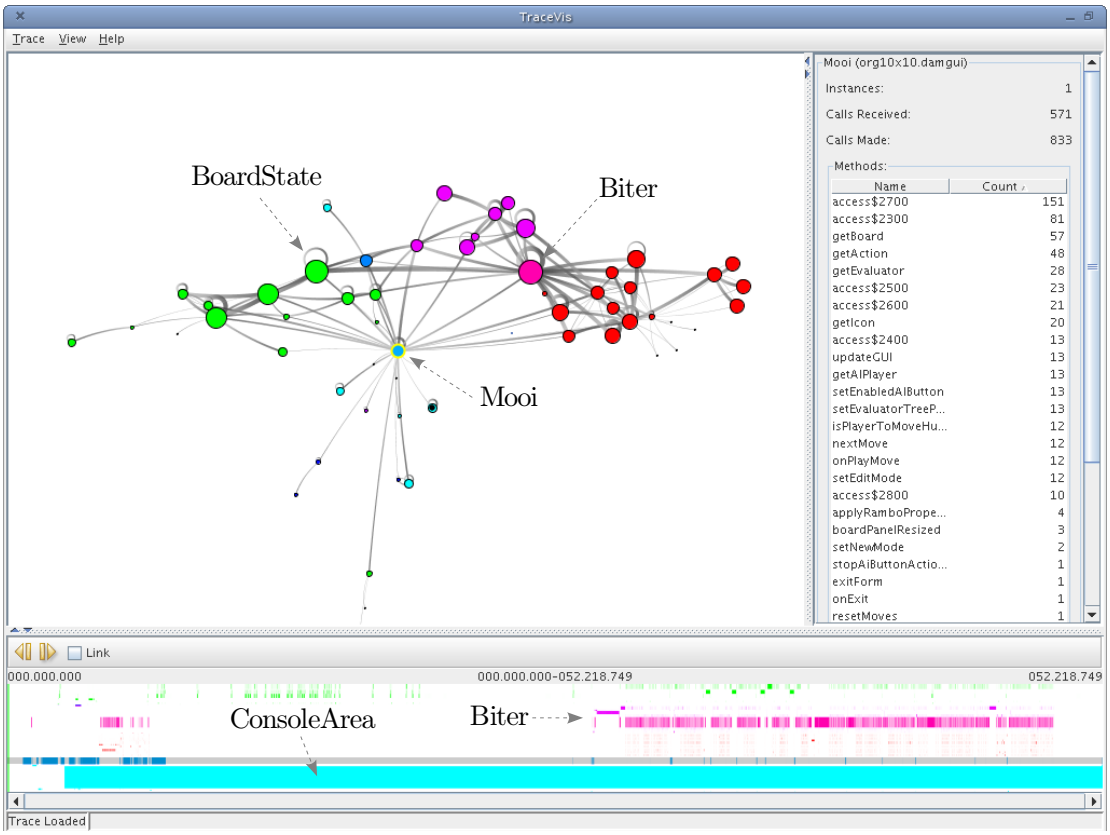


Figure 5.1: An overview of an execution trace from Dambo.

General Observations

Figure 5.1 shows an overview of the trace. From this picture, and after interaction, a few observations can be made about this trace.

- The program spends most of its time in the classes `Biter` and `ConsoleArea`.
- The class `Biter` receives many calls from the evaluator classes, and sends many calls to the class `BoardState`.
- The class `Mooi` (the main class) sends calls to many other classes, and receives very little calls.
- A clear separation is visible between the back end, which computes the best move for a given game situation (red, pink, and purple classes in the figure), and the front end, which shows the game situation on the screen (green classes).
- Two phases in the execution can be discerned, the initialization and the playing of the game.

Allocation

TraceVis can be used to gain an insight into allocation. A programmer might want to know which classes have the most instances. He might also want to know which other classes

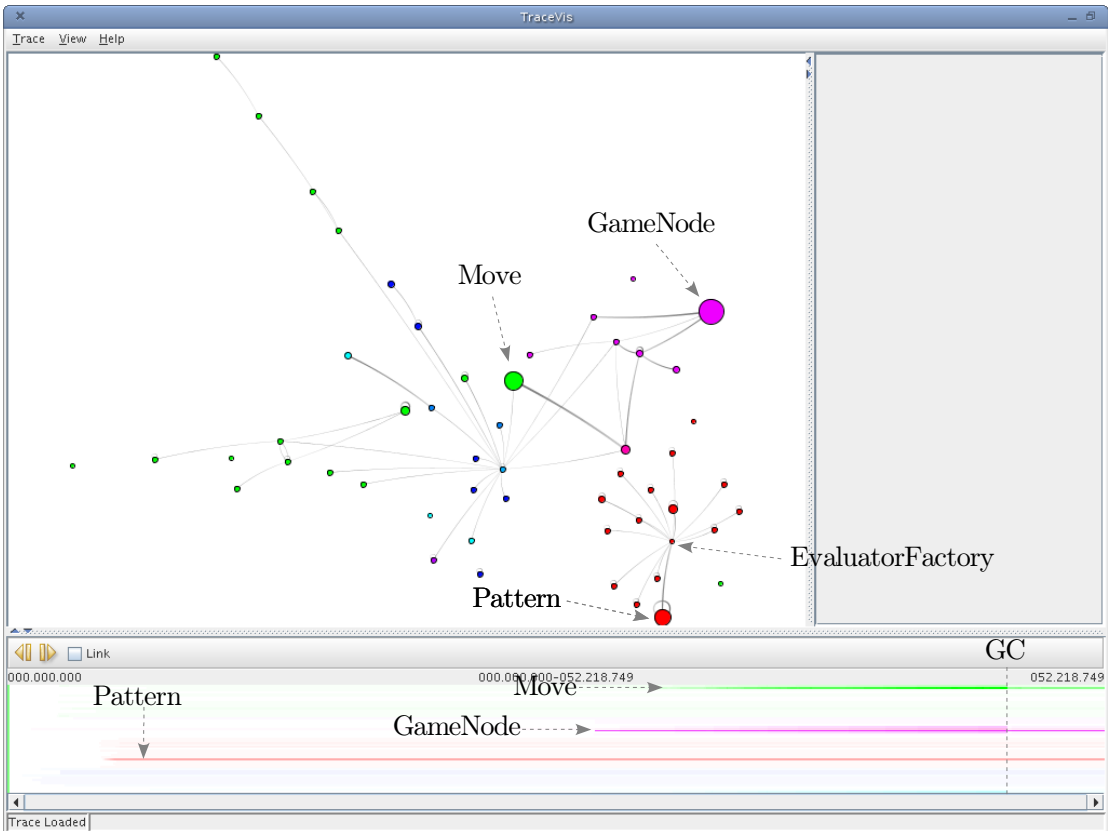


Figure 5.2: TraceVis configured to show allocations.

allocated these instances. A programmer can use TraceVis to find answers to these questions by:

- assigning calls to the object class when loading the trace,
- selecting the number of instances as the attribute which determines the vertex size,
- filtering all but the constructor calls from the event list, and
- choosing the instance view.

This results are shown in Figure 5.2.

Three classes stand out: `GameNode`, `Pattern` and `Move`. By looking at the structural view it can be determined that `GameNode` is allocated by three other classes, `Pattern` is allocated by `EvaluatorFactory`, and `Move` mainly by `Biter`. The instance view tells the user that most instances of `GameNode` are allocated at once. At one point in the execution, the number of instances of `Move` drops drastically, together with the numbers of instances of other classes. This is due to garbage collection (GC).

5.1.2 Ant

Apache Ant is a software build tool. It is mainly used to build Java software, but can be used to build other things, such as documentation, as well. Ant's main input is a so-called

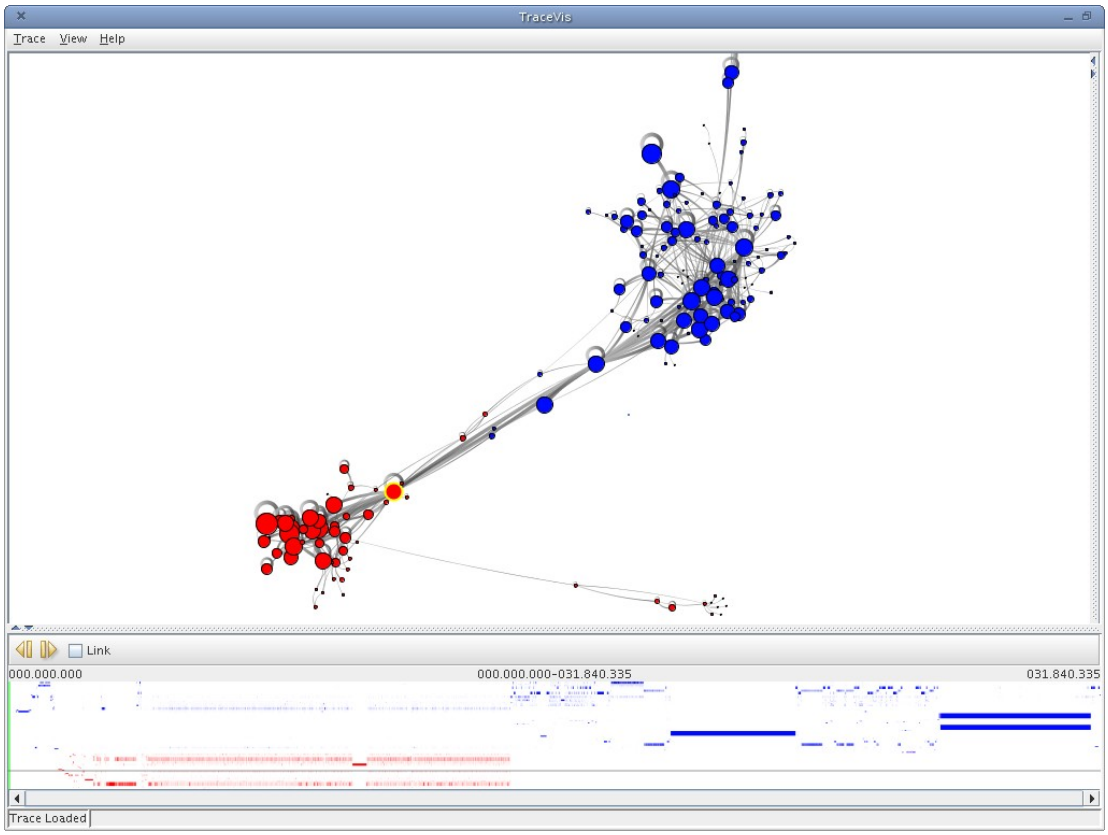


Figure 5.3: An example of an execution trace from Ant. In the structural view, two clusters are clearly visible. The left one is the Xerces XML library, which has been colored red. The right one is Ant's own code, which has been colored blue. In the time line, a large part of the activations near the start of the execution are colored red. This means that Ant is reading its buildfile, which is an XML file.

buildfile, which specifies which actions have to be undertaken to build the software. This buildfile is written in XML.

The trace covered in this section covers the building of the class files and documentation of a small Java project. It contains classes from both the Xerces XML Library [22] and Ant itself. Classes have been colored in the following way:

- The Xerces classes have been colored red.
- The Ant classes have been colored blue.

The trace contains 175 active classes and more 250.000 events.

General Observations

Figure 5.3 shows an overview of the trace. A few observations can be made about the trace.

- The separation between Xerces and Ant itself is strict, as can be determined by the presence of two clusters. A conclusion which can be drawn from this is that both Xerces and Ant are cohesive, and are loosely coupled.

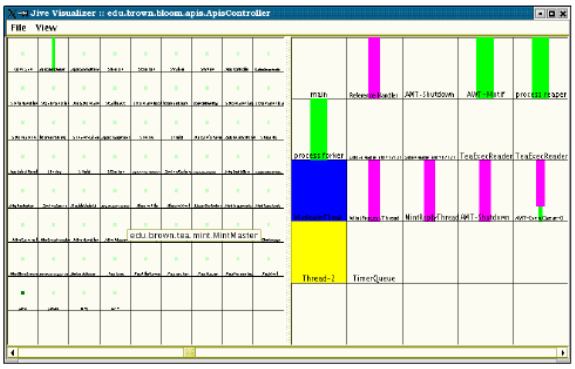


Figure 5.4: A typical view of Jive, from [19]. On the left, the class and package view. On the right, the thread view.

- The XML library is active in the first half of the execution. This corresponds to the reading of the XML buildfile.

- Long activations can be seen in the second half of the execution. These correspond to the running of external tools, such as the Java compiler, or the Javadoc tool.

Another observation is that with 175 classes, the structural view becomes cluttered.

5.2 Related Research

TraceVis is not the only tool which tries to visualize program executions. In fact, there are many tools which try to do that. It would not be feasible to compare TraceVis to all these tools. Instead we selected a few other tools to compare to our tool.

5.2.1 Jive

Jive [19] is an online visualization tool, i.e., it runs simultaneously with the program under study. Figure 5.4 shows a typical view of Jive. It provides two views: a class and package view, and a thread view. The class and package view is divided into a grid in which each cell visualizes runtime aspects about a class or package. These runtime aspects include the number of calls a class has received, the number of instances, or how often a thread has synchronized on instances of a class. These aspects can be mapped to various visual properties of a cell, such as width, height, brightness, or hue.

Jive is equipped with a “rewind” function, that is, the user can temporarily suspend the program and scroll to a previous state of the program. This function is accessed by a simple scroll bar, so the resolution the user can select a point in the execution with is limited.

An interesting aspect of Jive is that its data collection is fairly efficient. Therefore, it can be run alongside the program under study without too much slow down. However, as described in [19], the authors had to restrict the amount of data which is collected, which limits its usefulness.

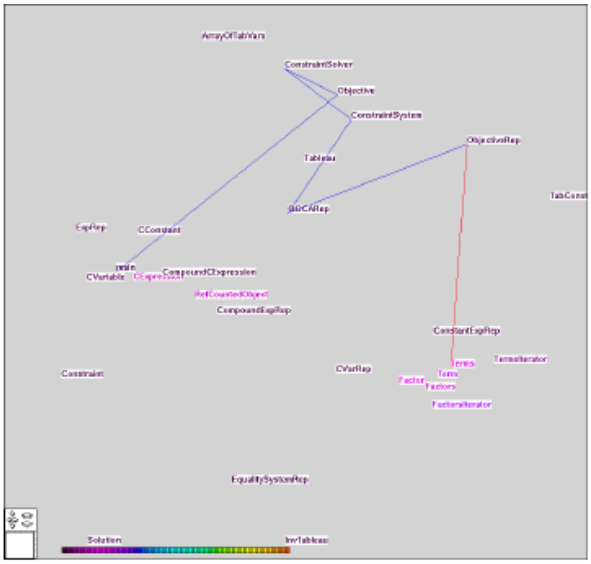


Figure 5.5: An inter-class call cluster, from [18].

5.2.2 Inter-Class Call Clusters

Inter-class call clusters were designed by De Pauw et al. [18] to provide a dynamic overview of communication patterns between classes. Figure 5.5 shows a snapshot of this view. The tool described in this paper is an online visualization tool, which continually updates its view to reflect the current status of the program.

The view draws classes as floating labels containing their names. The amount of communication between classes determines the distance between them. As the execution progresses, classes which communicate more often will be placed closer together. The view does not show edges between classes, but it does show the stack.

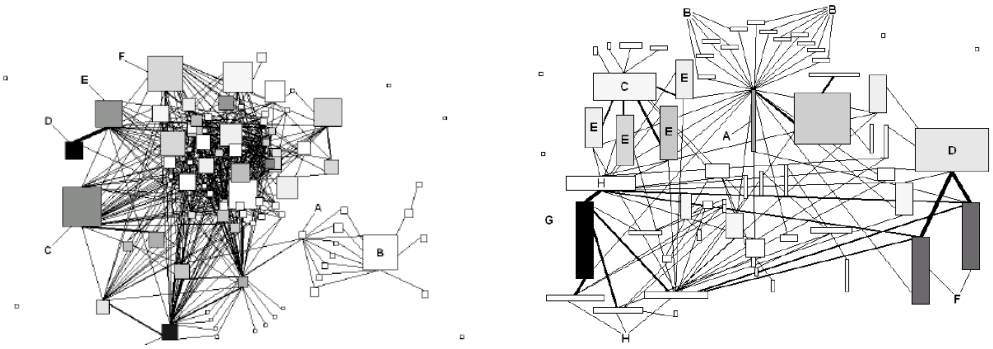
5.2.3 Communication and Creation Interaction Views

The communication interaction view is one of the views presented by Bertuli et al. in [4]. It is an offline visualization. During the execution of the program under study, certain statistics about classes are gathered, such as the number of method invocations, and the number of called methods. After the execution these statistics are displayed in the communication interaction view, as shown in Figure 5.6(a).

The creation interaction view is similar to the communication interaction view, but instead of statistics related to calls, statistics related to creation are displayed. Figure 5.6(b) shows an example of the creation interaction view.

5.3 Comparison

Unfortunately, a direct comparison, where all tools are used to solve the same use cases, is not possible. The system underlying the inter-class call clusters only supports C++ programs. The tool which provides communication and creation interaction views seems to support only SmallTalk programs. Moreover, both tools do not seem to be publicly available. Only Jive



(a) A communication interaction view. (b) A creation interaction view.

Figure 5.6: Two class interaction views from [4].

supports Java programs and is available in an executable form.

While Jive visualizes about the same aspects of classes as TraceVis, it does not show anything about their interactions. Therefore, Jive can not be used gain insight in these interactions. It can, however, be used to find out which classes are called the most frequently, or have the most instances. It is probably more convenient to do this with Jive, because Jive can be run alongside the program under study.

In contrast, inter-class call clusters show very little information about classes, but focus on call relationships between them. They can not be used for much more than determining communication.

Communication and creation interaction views are much like the structural view of TraceVis. They show about the same information. However, this information covers the whole execution. It is not possible to select subranges of the execution. It also does not provide any insight into the time aspects of the execution.

To conclude, TraceVis offers several improvements over similar tools.

Chapter 6

Project Evaluation and Conclusions

6.1 Project Evaluation

This master's project had three main activities:

- Visualization research. In the beginning of the project, we tried to gain insight into the ways dynamic program aspects can be visualized. To this end, many prototypes were created. We also studied several papers in the field of software visualization to become familiar with past work in this field. This initial research led us to focus on class interaction. During the project, many things were experimented with to arrive at the current solution.

- Data collection. In the beginning of the project, simultaneously with the initial visualization research, we developed several prototypes using JDI to get an idea about which data is available using this API. Later on, a significant amount of work was put into creating a definitive data collector using JVMTI and BCI.

- Tool development. After the prototyping phase, development on a visualization tool was started. This tool was used to experiment with visualizations for class interaction.

During these activities, the following lessons were learned:

- The importance of keeping design and code manageable. During the development of the visualization and the tool, requirements changed frequently. This sometimes meant that code designed carefully for a given feature had to be restructured to accommodate new features. New features do not always warrant such elaborate design, especially when they are meant to try something out. Only after they have proved their worth is it worthwhile to clean up the code which implements them. By undertaking frequent efforts to refactor the code, it was kept manageable, while still allowing changes to be made quickly.

- The importance of initial research. The resulting tool is useful, but it might have been more useful if we had conducted more research into what programmers really want to know about executions.

- Data collection is not trivial. It is a problem which has many different solutions. Unfortunately, they could not all be investigated due to time constraints. Data collection is

an interesting topic in itself, where one has to deal with performance problems and large data sets, and worth much more time than could be spent on it during this project.

6.2 Conclusions

In Chapter 1 the goal of this project has been defined to be the development of a tool through which the user can gain insight into the dynamic interaction between classes. Chapters 2, 3, and 4 discussed the development of the tool TraceVis. In Chapter 5, the tool was evaluated by using it in a few test cases and by comparing it to similar tools. The conclusion is that TraceVis can indeed be used to gain insight into class interaction, and that it provides improvements over similar tools. For large programs, however, some improvements are needed.

Although the code of TraceVis has a few rough edges due to continual experimentation, it is well structured. It should be relatively straightforward to improve the existing views and new views can be added with relatively little effort. In addition, this project has delivered a data collector which can create execution traces for programs. The trace format the data collector creates is easy to comprehend, parse, and extend.

6.3 Future Work

Some suggestions for future work are:

- Visualize profiling data. By extending the data collector to collect profiling information, such as very accurate timings or the actual size of objects, TraceVis can be used as a profiling tool. For instance, the size of a class can then depend on the time spent in this class.

- Improve the data collector. The data collector might be improved to repair the limitations listed in Section 3.4.

- Make use of hierarchies to increase the scalability of the visualization. The current tool does not make use of hierarchies present in the data, such as the package hierarchy. For instance, classes of relatively uninteresting packages might be collapsed to a single vertex in the structural view, or into a single row in the time line view. Conversely, a class might be so interesting that it is desirable to break it down in separate methods or objects.

- Provide additional filters. Currently, only a few filters are implemented. An interesting filter would be to show only those calls which are either indirectly calling a selected class, or have been indirectly called by the same class.

- Improve the layout. For traces with many classes, the layout method does not perform well. One could explore possible improvements to the layout method which make its output more readable for large graphs. Another interesting layout would be a matrix-based layout, such as the one described by Van Ham [21].

- Improve the height scaling of the activity view. The height of a class is based on its activity over the whole of the execution. This might be improved by computing the class height on a per-pixel basis. The height of a class might then replace the saturation

as the cue which indicates activity. The activity view will also make better use of screen space this way, and color can be used for other purposes.

- Add height scaling to the instance view. TraceVis does not implement height scaling for the instance view as it does for the activity view.
- Skip events in larger steps. The user can jump to the next or previous event one at a time. This is fine-grained and is not well suited to explore large traces.
- Add support for dynamically typed languages such as Python or Perl. For a statically typed languages such as Java, a fair amount of static analysis can be performed. For dynamically typed languages, this is not the case. Hence, programmers using dynamic languages might have more benefit from a dynamic visualization than programmers using a static language.

Appendix A

Java

Java is an object-oriented, general-purpose programming language, created by Sun Microsystems. Its syntax is similar to C and C++, but it is designed to be safer and easier to program in than those languages. The Java programming language is defined in the Java Language Specification [8]. The following sections describe those aspects of Java which are relevant for this project.

A.1 Language Concepts

As an object-oriented language, Java focuses on constructs called *objects*. An object consists of data and operations (called *methods*). Objects of the same kind are instances of the same *class*. A class defines which data and methods its instances have. The following sections describe these, and other, concepts.

A.1.1 Classes

A *class* defines a data type. A class has a name, called an *identifier*, and it contains two types of elements (called *members*):

- *methods*, which define the operations the class supports, and
- *fields*, which define the data each instance of the class stores.

A.1.2 Methods

A *method* defines an operation. This operation can be parameterized by passing *parameters* to the method. The *signature* of a method consists of its name, and the number and types of its parameters. Every method belongs to a class. Each class can have multiple methods with the same name, which is called *overloading*, but only one method for each signature.

The functionality defined in a method can be used by means of a *method call*.

A *method declaration* introduces a method as a member, without providing an implementation. A *method definition* introduces a method as a member and provides an implementation.

An *instance method* is a method which can only be called with respect to an object. A *static method* (also called a *class method*) is a method which can be called without reference to a specific object.

A.1.3 Objects

An *object* is an instance of a class. Each object has its own copy of the fields specified by the class. The fields of an object are initialized by a special method, called the *constructor*. A class can define a constructor explicitly, or omit it, in which case the Java language adds a default constructor. A class can define multiple constructors, provided that they have different signatures.

A.1.4 Inheritance

Inheritance describes a relation between two classes, a *derived class*, or *subclass*, and a *parent class*, or *superclass*. The derived class inherits the methods and fields of the parent class, which allows for shared functionality. In addition to inheriting the methods from its parent class, a derived class can add new methods and/or redefine the parent's methods. Redefining a method is also called *overriding*, and the new method is then called an *overridden method*.

Java supports only *single inheritance*, i.e., every class has at most one superclass. In Java, every class is directly or indirectly a subclass of the class `Object`.

An *abstract class* is a class that can not be instantiated. In addition to normal methods, an abstract class can contain *abstract methods*. An abstract method is a method which is just declared, i.e., it has no implementation. Non-abstract subclasses of an abstract class need to implement all abstract methods of the superclass and can be instantiated.

An *interface* is a construct related to an abstract class. Its definition is similar to a class definition, except that an interface can only contain abstract methods. An interface may *extend* another interface, which means that the first class, also called the *subinterface*, inherits the methods declared in the latter class, also called the *superinterface*, and adds new method declarations. A class may *implement* an interface, which means that it must implement all methods declared in this interface. A class can implement multiple interfaces. Interfaces thus provide a kind of multiple inheritance.

A.1.5 Exceptions

The Java language defines a mechanism for the handling of errors, called *exception handling*. An *exception* signifies that an erroneous condition has occurred. It will cause a transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be *thrown* from the point where it occurred and is said to be *caught* at the point to which control is transferred. When an exception is caught, an *exception handler* is executed.

An exception can be thrown by the JVM, to signify that the program violates the semantics of the language, such as division by zero, has exceeded a resource limit, such as using too much memory, or that an internal error has occurred. A program can also throw exceptions explicitly.

An exception is represented by an instance of the class `Throwable` or one of its subclasses. These subclasses represent different kinds of exception. For example, Java defines the exception classes `ArithmeticExcepton` and `OutOfMemoryError`, which are both subclasses of `Throwable`. They, respectively, signify that an exceptional arithmetic condition, such as division by zero, has occurred or that the JVM has exceeded its memory limit. A programmer can specify that an exception handler handles all kinds of exceptions, or only specific kinds.

If the statement which caused an exception to be thrown is enclosed by a suitable exception handler (i.e., it can handle the kind of exception which is thrown), control is transferred to this exception handler. If this is not the case, control is transferred to the method which called the current method. This method can either handle the exception, or transfer control to the method which called *this* method. This procedure continues until a suitable exception handler is found.

A.2 Program Structure

All Java programs are organized as sets of *packages*. A package is a set of classes, interfaces and subpackages. A subpackage is a package itself, which can in turn contain classes, interfaces and subpackages. The programmer can define a package hierarchy in this way.

A class can contain fields, methods and other classes. A class which is contained in another class or interface is called a *nested class*. A *top level class* is a class that is not a nested class. The class a nested class is contained in is called the *enclosing class*.

Classes, interfaces and packages have two kinds of names: a *simple name* and a *fully qualified name*. The simple name of a class, interface or package is its identifier. The fully qualified name of a package, which is not a subpackage of another package is its simple name. If `P` is the fully qualified name of a package, and `Q` is the simple name of a package contained in `P`, the fully qualified name of `Q` is `P.Q`. If `P` is the fully qualified name of a package, and `R` is the simple name of a top level class or interface contained in `P`, the fully qualified name of `R` is `P.R`.

A.3 Compilation

A Java programmer generates executable code (commonly called *bytecode*) by invoking the Java compiler. The Java compiler produces a *class file* for each class or interface in the program. Multiple class files can be combined in a single *JAR file*. The Java compiler locates the class files necessary for the compilation (e.g., external libraries) using the *class path*, a list of directories and/or JAR files which contain class files.

A.4 Runtime

Java programs are executed on an abstract computing machine, the *Java Virtual Machine (JVM)* [17].

A.4.1 Class Loading

The bytecode for a program is stored in one or more class files. Each class file contains the code for a single class or interface. The JVM loads a class file when the code it contains is needed. Similar to compilation, class files are located using the class path.

A.4.2 Threads

The JVM can support multiple *threads of execution* at once. A thread executes code, independently of other threads. Threads share memory, i.e., they have access to the same set of

objects.

A new thread can be created by creating an object of the class `java.lang.Thread` and calling the method `start` on this object. This method will create a thread and call the method `run` of the `Thread` object. A thread stops execution when its `run` method terminates.

A thread can be marked as a *daemon thread*. The JVM starts up with a single non-daemon thread, which typically calls the (static) `main` method of a class which is specified on the command line. The virtual machine exits when all non-daemon threads have stopped.

A.4.3 Call Stack

Each thread has a *call stack*. A call stack is a *last-in/first-out* data structure, which stores *stack frames*. Among other things, the call stack is used to keep track of method calls and returns. A method call transfers control from the calling method to the called method, and a new frame is created and put on top of (*pushed on*) the call stack of the thread. A frame is removed from the top of the thread's call stack (*popped*) when its method call completes. A method call can complete either *normally* or *abruptly* (due to an uncaught exception).

The frame on top of a call stack in called the *current frame*, and its method is the *current method*. The class in which this method is defined is called the *current class*.

If the method of a frame is an instance method, this frame stores a reference to the instance this method was invoked on. In the Java programming language, this reference is called `this`.

A.4.4 Objects

Objects can be created during the execution of a program. When a new instance of a class `C` is created, the JVM reserves sufficient memory to store the fields defined in `C` and its superclasses. To initialize these fields, a constructor of class `C` is called[1], which does the following[2]:

1. If `C` is not `java.lang.Object`, a constructor of the superclass of `C` is called[1]. This call is processed recursively using these same two steps.

2. The constructor of class `C` is executed.

In essence, this procedure initializes the fields defined by `java.lang.Object` first, followed by the next-derived class, and so on, until, finally, the fields defined by `C` are initialized.

Java has no support for explicit deallocation of memory. Java relies on a so-called *garbage collector* to clean up unused objects. A garbage collector checks periodically which objects are unused and deallocates them. If the class of the object defines the method `finalize`, it is called when the object is deallocated.

[1] Which constructor is called depends on the parameters passed to the constructor.

[2] This procedure is a simplification of the procedure described in [17, Section 2.17.6].

Appendix B

Trace File Format

An execution trace is stored in a ZIP file. This ZIP file contains a single file, called `trace`, which stores the actual trace data in a textual, line-oriented format. This format is described further on.

By using a ZIP file and a textual, line-oriented format, a trace can be opened and read by a programmer without using specialized tools. A ZIP utility and a text editor suffice. This has simplified the development of the trace generator and parser. Furthermore, a ZIP file is easily extensible, because additional information about the trace can be stored by adding files to the ZIP file.

A disadvantage of this approach is that, without compression, the file `trace` is very large. This situation is improved by using the compression offered by a ZIP file. Another disadvantage is that, due to the combination of ZIP decompression and the parsing of a large file, loading a trace takes a long time.

Nowadays, it is a common choice to base a textual format on XML. XML is a good format to store complex hierarchical data. It is well known in the industry and many tools exist to deal with it. XML formats add a lot of structural information (tags). This structural information is not strictly necessary for the `trace` file, because the structure of the data it stores (a list of events) is fairly simple. It would only increase the size the `trace` file, without significantly improving readability. Furthermore, the format described in the next section is simpler than an XML format would be. For this project, it was considered best to use the simplest format that could store the needed data. Hence, the `trace` file uses a custom format, which is not based on XML.

B.1 The `trace` File

The file `trace` is a text file (i.e. it is human readable), encoded as UTF-8. It stores a list of events, separated by newlines. All events have the following header:

```
<event type>:<time stamp>
```

where

- `<event type>` is the event type, encoded as two capital letters, and
- `<time stamp>` is the event's time stamp, the value of the system timer at the time of the event.

Information specific to an event type can be included after the header. The event types are described in the following sections.

VM Start Event

A VM Start Event indicates the start of the Java Virtual Machine. It is always the first event in a trace. Its syntax is:

```
VS:<time stamp>
```

VM Init Event

A VM Init Event indicates that the Java Virtual Machine is initialized and ready to execute the program. It is always the second event in a trace. Its syntax is:

```
VI:<time stamp>
```

VM Death Event

A VM Death Event indicates that the Java Virtual Machine has terminated. It is always the last event in a trace. Its syntax is:

```
VD:<time stamp>
```

Class Load Event

A Class Load Event indicates that the Java Virtual Machine has loaded a class. Its syntax is:

```
CL:<time stamp>:<class name>
```

where `<class name>` is the fully qualified name of the loaded class. The class name is stored as it is in a class file, i.e., the identifiers that make up the fully qualified class name are separated by an ASCII forward slash ('/') (See [17, Section 4.2]). For instance, the class `java.lang.Object` (as used in the Java programming language) is stored as `java/lang/Object`.

Object Allocation Event

An Object Allocation Event indicates that an object has been created. Its syntax is:

```
OA:<time stamp>:<class name>:<object id>
```

where

- `<class name>` is the name of the created object's class, and
- `<object id>` is a unique number identifying the object.

Object Free Event

An Object Free Event indicates that an object has been freed by the garbage collector. Its syntax is:

```
OF:<time stamp>:<class name>:<object id>
```

where

- `<class name>` is the name of the created object's class, and
- `<object id>` is a unique number identifying the object.

Thread Start Event

A Thread Start Event indicates that a thread has been started. Its syntax is:

```
TB:<time stamp>:<thread id>
```

where `<thread id>` is a unique number identifying the thread.

Thread Stop Event

A Thread Stop Event indicates that a thread has been stopped. Its syntax is:

```
TE:<time stamp>:<thread id>
```

where `<thread id>` is a unique number identifying the thread.

Method Entry Event

A Method Entry Event indicates that a thread has entered a method. Its syntax is:

```
MN:<time stamp>:<thread id>:<class name>:<method name>:<object id>
```

where

- `<thread id>` is the identifier of the thread entering this method,
- `<class name>` is the name of the class defining the entered method,
- `<method name>` is the name of the entered method, and
- `<object id>` is the identifier of the object referenced to by `this` in the current frame. It is 0 if the entered method is static.

Method Exit Event

A Method Exit Event indicates that a thread has left a method. Its syntax is:

```
MX:<time stamp>:<thread id>:<class name>:<method name>
```

where

- `<thread id>` is the identifier of the thread leaving this method,
- `<class name>` is the name of the class defining the left method,
- `<method name>` is the name of the left method, and

Frame Pop Event

A Frame Pop Event indicates that a frame has been popped due to an exception. Its syntax is:

```
FP:<time stamp>:<thread id>:<class name>:<method name>
```

where

- `<thread id>` is the identifier of the thread in which the frame was popped,
- `<class name>` is the name of the class defining the popped method, and
- `<method name>` is the name of the popped method.

Bibliography

[1] Apache Ant [online, cited 10 May 2006]. Available from: `http://ant.apache.org/`.

[2] ASM [online, cited 13 February 2006]. Available from: `http://asm.objectweb.org/`.

[3] BCEL (Byte Code Engineering Library) [online, cited 13 February 2006]. Available from: `http://jakarta.apache.org/bcel/`.

[4] Roland Bertuli, Stéphane Ducasse, and Michele Lanza. Run-time information visualization for understanding object-oriented systems. In *Proceedings of WOOR 2003 (4th International Workshop on Object-Oriented Reengineering)*, pages 10 – 19, 2003.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Chicester, UK, 1996.

[6] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[7] Martin Fowler. *UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, Mass., 2004.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Boston, Mass., 2005. Available from: `http://java.sun.com/docs/books/jls/`.

[9] David Harel and Yehuda Koren. Drawing graphs with non-uniform vertices. In *Proceedings of Working Conference on Advanced Visual Interfaces (AVI'02)*, pages 157–166. ACM Press, 2002. Available from: `citeseer.ist.psu.edu/harel02drawing.html`.

[10] JDI (Java Debug Interface) specification [online, cited 8 February 2006]. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/index.html`.

[11] JDK (Java Development Kit) [online, cited 13 February 2006]. Available from: `http://java.sun.com`.

[12] JDWP (Java Debug Wire Protocol) specification [online, cited 8 February 2006]. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdwp-spec.html`.

[13] JPDA (Java Platform Debug Architecture) documentation [online, cited 8 February 2006]. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/`.

[14] JUNG (Java Universal Network/Graph Framework) [online, cited 6 February 2006]. Available from: `http://jung.sourceforge.net`.

[15] JVMTI (Java Virtual Machine Tool Interface) specification [online, cited 8 February 2006]. Available from: `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html`.

[16] Michael Kaufmann and Dorothea Wagner, editors. *Drawing graphs: methods and models*. Springer-Verlag, London, UK, 2001.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, Boston, Mass., 1999. Available from: `http://java.sun.com/docs/books/vmspec/`.

[18] Wim De Pauw, Doug Kimelman, and John M. Vlissides. Modeling object-oriented program execution. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 163–182, London, UK, 1994. Springer-Verlag.

[19] Steven P. Reiss. Visualizing java in action. In Stephan Diehl, John T. Stasko, and Stephen N. Spencer, editors, *SOFTVIS*, pages 57–65, 210. ACM, 2003.

[20] Kozo Sugiyama and Kazuo Misue. A simple and unified method for drawing graphs: Magnetic-spring algorithm. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 364–375, London, UK, 1995. Springer-Verlag.

[21] Frank van Ham. Using multilevel call matrices in large software projects. In *INFOVIS*. IEEE Computer Society, 2003.

[22] Xerces XML parser [online, cited 19 May 2006]. Available from: `http://xerces.apache.org`.