UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENȚĂ

3D Software Visualization Tool for Enhanced Code Comprehension

Conducător ştiinţific
Boian Rareş

Absolvent
Balázs Béla

2012

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

ENGLISH COMPUTER SCIENCE SPECIALIZATION

LICENSE THESIS

3D Software Visualization Tool for Enhanced Code Comprehension

Supervisor
Boian Rareş

Author
Balázs Béla

2012

Table of Contents

1. Introduction

Contemporary Software Systems keep growing in size and complexity and their development and maintenance requires the collaboration of many people, often from different companies and different teams. This increases the difficulty of programming, understanding and modifying the source code, especially when this is done based on other people's code. Therefore many tools have been developed to aid this process; most of them are integrated directly into the development environment and are specific to the platform and the programming language and assume that development is done on similar environments.

One of the biggest challenges in software engineering consists in maintaining complex software systems. Adaptive, corrective or perfective maintenance needs to be done throughout the period in which the system is deployed in a production environment.
Maintenance is often done by third-parties, who were not involved in the development phase of the product and change requests may occur even years after the system was launched into production, when the contractual obligations of the original developers have long since expired and the code is categorized as legacy code.

An interesting example about how these processes are dealt with can be studied observing them in the open-source world. In the open-source model of development, the software product is developed and maintained by the community and this product is available to anyone, free of charge, together with the source code and anyone is free to modify it, submit patches and start a different project based on existing code. These decisions are motivated by the statement: *"Given enough eyeballs, all bugs are shallow* [15] *."*
These teams consist of programmers with different levels of experience and technical background using various development environments and operating systems.
As the number of modules, packages and classes increases so does the time needed to find the implementation of a specific behavior or feature within a system.
There is a clear need for code comprehension tools, because having an overview over the entire codebase contributes to writing quality code. It is important for every developer to be aware of the conventions used in a project and to keep a consistent style, because it is desirable to keep similar functionality in the same place and to have a clear distribution of responsibilities for each class. Duplicating existing functionality adds redundant complexity to the code-base.

The most important source of information when dealing with an unfamiliar codebase is the documentation. This can consist of various documents containing the technical design of the application, UML diagrams describing structure, state and interaction between components, method and class descriptions, various guides and how-to-s.
Writing this documentation and more importantly keeping it up to date in order to reflect all the changes in implementation, requirements, changes after refactoring, major architecture changes is definitely not a trivial task. Software companies try their best to ensure that this happens, but since deadlines are tight or the budget is limited, documentation often gets neglected. Even more so in the open source world, where everyone is a volunteer and other tasks have bigger priorities.

When a programmer enters a new project or just simply has to modify existing code with the purpose of implementing new features or fixing critical bugs the first challenge is finding out where to look for specific functionality. The more complex the source tree, the more time it takes to familiarize the structure and to mentally bind functionality to locations in source code.
In an experiment done at the Microsoft research department, seven developers with an average of 18 years of experience and high familiarity of their development environment were asked to make corrections and add features to an unfamiliar codebase.
Despite their expertise the participants had the following problems:

- *“They wanted more documentation to help identify the important parts of the code and to describe how the parts are related.*
- *They got lost as they navigated around the code base, particularly as the number of open documents increased.*
- *They relied heavily on textual search to find relevant parts of the code, lost time separating good search results from bad, and became distracted by results that seemed relevant but were not [14].”*

Software visualization is the static or animated 2D or 3D visual representation of information about software systems based on their structure, size, history or behavior [6].
The objective of software visualization is to aid the understanding of software systems; this can refer to their structure, the applied algorithms as well as the analysis of applications to discover faults and anomalies (circular dependencies, high coupling, etc).

These tools aid the analysis, modeling, debugging, testing and maintenance of software systems and anecdotal evidence exists that appropriate visualizations can significantly reduce the time and effort involved in these activities.

Harel, for instance, states that "*using appropriate visual formalisms can have a spectacular effect on engineers and programmers* [10]."

Software visualization tools can gather data statically by analyzing the source code of the software system or dynamically by analyzing runtime behavior.

The Symbion project aims to build a software analysis and visualization tool that uses profiling techniques to analyze runtime behavior and visualizes this data in 3D using a game engine, in order to aid the understanding of the structure of the source code and to ease the process of mentally mapping runtime behavior to source code locations.

The Symbion profiling agent attaches itself to the host program using the java.lang.instrument API and intercepts the method calls that match a given set of rules, in this way recording the call-graph of the given execution. This is done by binding a TCP server to a port and connecting to it using the Symbion console, which enables the user to interactively control and fine-tune the profiling process. When the profiling is stopped, the gathered data is sent to the client side and stored. The Symbion visualizer reads the gathered data, constructs a visual model to represent it and allows the navigation of the execution, displaying the relevant source code that can be viewed in parallel. This visual model aids the comprehension of the source code, by focusing only on the portions that were executed and presenting it in a logical and intuitive way.

This name was chosen because the profiler attaches itself to the host program in a way that resembles symbiotic relationships in nature and also because the visualizer uses different interconnected nodes that form a mistletoe-like structure.

To explain the motivation behind this project I will present a simple example. A developer is appointed the task of implementing a new feature or fixing a bug in a very complex system, with hundreds, even thousands of classes and packages. This application enables the developer to attach the profiler to the application, give some simple rules to exclude third-party code or other code that is beyond the scope of his task, run the application, enable the profiling process at any point in the programs execution, bring the host application in a state where the bug is reproduced, stop profiling and visualize this execution, while directly studying the source code of the executed methods. This provides a fast way of finding the source code portions which are related to his task and also provides a greater overview upon the interactions, relationships and dependencies between various classes, components and modules.

Further we will explore the process of implementing this idea, we will give justifications to various design decisions, we will try to find the best methods to present this data in an easy-to-navigate and intuitive manner and we will try to evaluate the effectiveness of this approach.
In **Chapter 2** we will present the history of this field and the current state of the art, related works that are similar to this project, we will compare them and justify the differences.
In **Chapter 3** we will describe the technologies that were used, why they were chosen and how they compare to alternatives.
In **Chapter 4** we will describe in detail the process of implementing the application, the design, the underlying algorithms, different layers of abstraction, communication protocols and the complete description of all the features.
In **Chapter 5** we will try to evaluate our approach, test its limits and see how it performs in terms of usability and effectiveness.
Chapter 6 contains the future plans and ideas that would improve our current application. **Chapter 7** presents the conclusions and summarizes our findings.

A journal article about this project with the same title was presented during the conference named: **Sesiunea de Comunicări Științifice ale Studenților**, 21 June, 2012.

2. Motivation and background

Data visualization is the study of the visual representation of data, meaning "*information that has been abstracted in some schematic form, including attributes or variables for the units of information* [8]".
Software visualization is a sub-domain of data visualization and aims to provide representations of various aspects of software systems that are more effective in conveying information.

2.1 Short history

Data visualization tools have been available since the early days of computing and have grown more and more sophisticated as the graphical capabilities increased.
In the period between 1950-1975 data visualization has experienced a real renaissance.
“By the end of this period significant intersections and collaborations would begin:
(a) computer science research (software tools, C language, UNIX, etc.) at Bell Laboratories (Becker,1994) and elsewhere would combine forces with
(b) developments in data analysis (EDA, psychometrics, etc.) and
(c) display and input technology (pen plotters, graphic terminals, digitizer tablets, the mouse, etc.). These developments would provide new paradigms, languages and software packages for expressing statistical ideas and implementing data graphics. In turn, they would lead to an explosive growth in new visualization methods and techniques. [7]“
The themes that emerged during this period include: various novel visual representations of multivariate data, Fourier function plots, multidimensional scaling, correspondence analysis, providing visualization of multidimensional data in 2D approximation, animations of statistical processes and the first exemplars of modern GIS and interactive systems for 2D and 3D statistical graphics.

“During the last quarter of the 20-th century data visualization has blossomed into a mature, vibrant and multi-disciplinary research area, as may be seen in this Handbook, and software tools for a wide range of visualization methods and data types are available for every desktop computer [7].”
During the 90-s the appearance and popularization of video games has boosted the demand for high performance graphics capable hardware that is capable of rendering data in real time.
Together with the development of the world-wide-web and the appearance of Java applets, technology that enables various data visualization applications became widely accessible.

A few relevant milestones in the history of data visualization:

- *"1988 Interactive graphics for multiple time series with direct manipulation (zoom, rescale, overlaying,etc.)— Antony Unwin and Graham Wills, UK [303]*
- *1991 Treemaps, for space-constrained visualization of hierarchies, using nested rectangles (size proportional to some numerical measure of the node)— Ben Shneiderman (1947–), USA [265, 153].*
- *1991–1996 A spate of development and public distribution of highly interactive systems for data analysis and visualization, e.g., XGobi, ViSta— Deborah Swayne, Di Cook, Andreas Buja [276, 37, 277], Forrest Young (1940–2006) [329], USA.*
- *1994 Table lens: Focus and context technique for viewing large tables; user can expand rows or columns to see the details, while keeping surrounding context— Ramana Rao and Stuart K. Card, Xerox Parc, USA [247].*
- *1996 Cartographic Data Visualiser: a map visualization toolkit with graphical tools for viewing data, including a wide range of mapping options for exploratory spatial data analysis— Jason Dykes, UK*
- *1999 Grammar of Graphics: A comprehensive systematization of grammatical rules for data and graphs and graph algebras within an object-oriented, computational framework— Leland Wilkinson (1944–), USA [322, 323].*
- *2004 Sparklines: "data-intense, design-simple, word-sized graphics," designed to show graphic information inline with text and tables — Edward Tufte (1942–), USA [290].* [8]*"*

Software visualization with the scope of understanding computer programs also has a long history. Some important early milestones [4]:

- 1947 Goldstein and Von Neumann demonstration of usefulness of flowcharts.
- 1959 Haibt developed a system that could draw flowcharts automatically from Fortran or Assembly language programs.
- 1963 Knuth produced a similar system which integrated documentation with the source code and could automatically generate flowcharts.
- 1968 Abrams wrote a review of these sort of systems.
- 1973 The Nassi-Shneiderman diagrams were developed to counter the unstructured nature of standard flowcharts.
- 1983 Myers Incense system was a more ambitious system for the display of data structures.
- 1984 One of the earliest attempts to build a debugging system to aid visualization was the work done in Lisp by Lieberman.

- 1985 Martin and McClure survey a variety of diagrammatic methods for the representation and display of program structure and behavior.
- 1990 Glinert explosion of interest in visual programming. Licklider also did early experiments on the use of computer graphics to illustrate how the memory was changing while the computer was running. This was the first use of animation in visualizing program behavior.

2.2 State of the art

There are many examples of modern software visualization, which have proven their effectiveness. I will present a few examples that are not necessarily code comprehension tools, but are popular due to their novel approaches.

Logstalgia is an open source real-time website traffic visualizer, which uses the apache access log stream to represent the traffic as a never-ending ping-pong like battle between the web server and the requests. Requests appear as colored balls (the same color as the host) which travel across the screen to arrive at the requested location. Successful requests are hit by the paddle while unsuccessful ones (ex. 404 - file not found) are missed and passed through. The paths of requests are summarized within the available space by identifying common path prefixes. Related paths are grouped together under headings. For instance, by default paths ending in png, gif or jpg are grouped under the heading Images. Paths that don't match any of the specified groups are lumped together under a miscellaneous section.

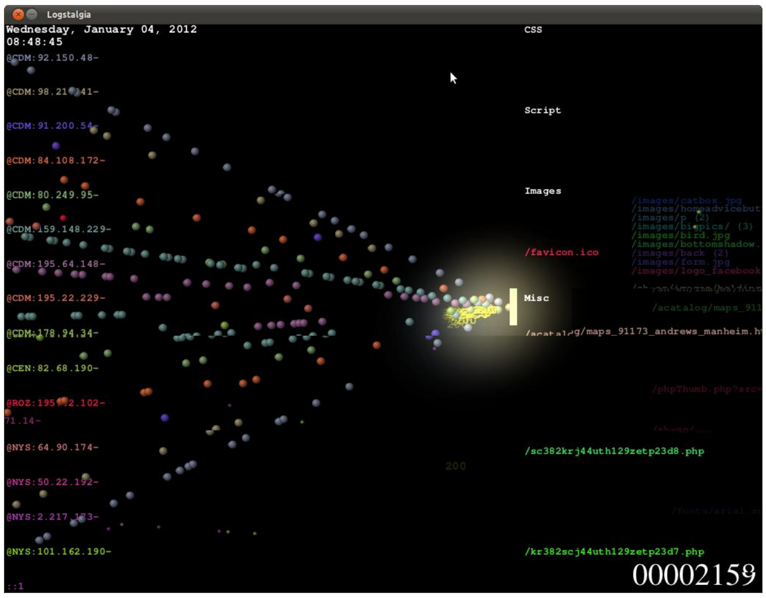

Logstalgia
Wednesday, January 04, 2012
08:48:45
CSS
Script
Images
/favicon.ico
Misc
/sc382krj44uth129zetp23d8.php
/kr382scj44uth129zetp23d7.php
00002159

Figure 1 DDoS attack in Logstalgia

Gource is an open-source software version control visualization tool which displays software projects as an animated tree with the root directory of the project at its center. Directories appear as branches with files as leaves. Developers can be seen working on the tree at the times they contributed to the project. Currently Gource includes built-in log generation support for Git, Mercurial and Bazaar and SVN. Gource can also parse logs produced by several third party tools for CVS repositories.

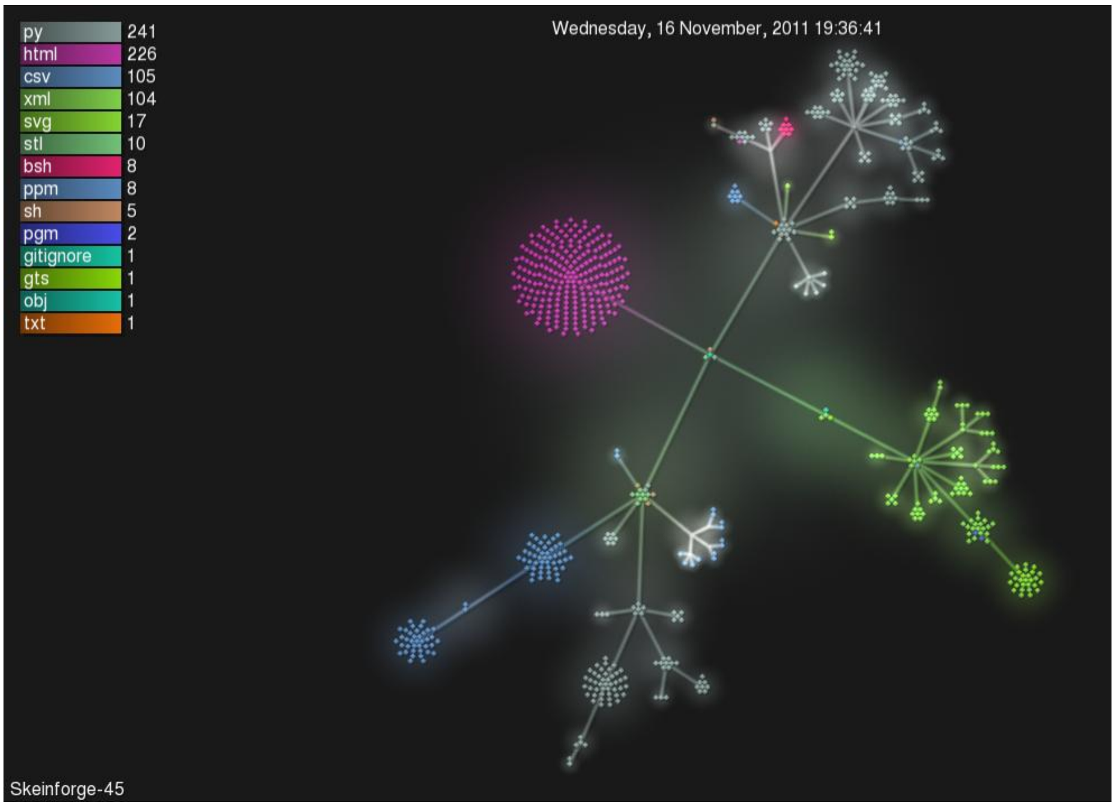
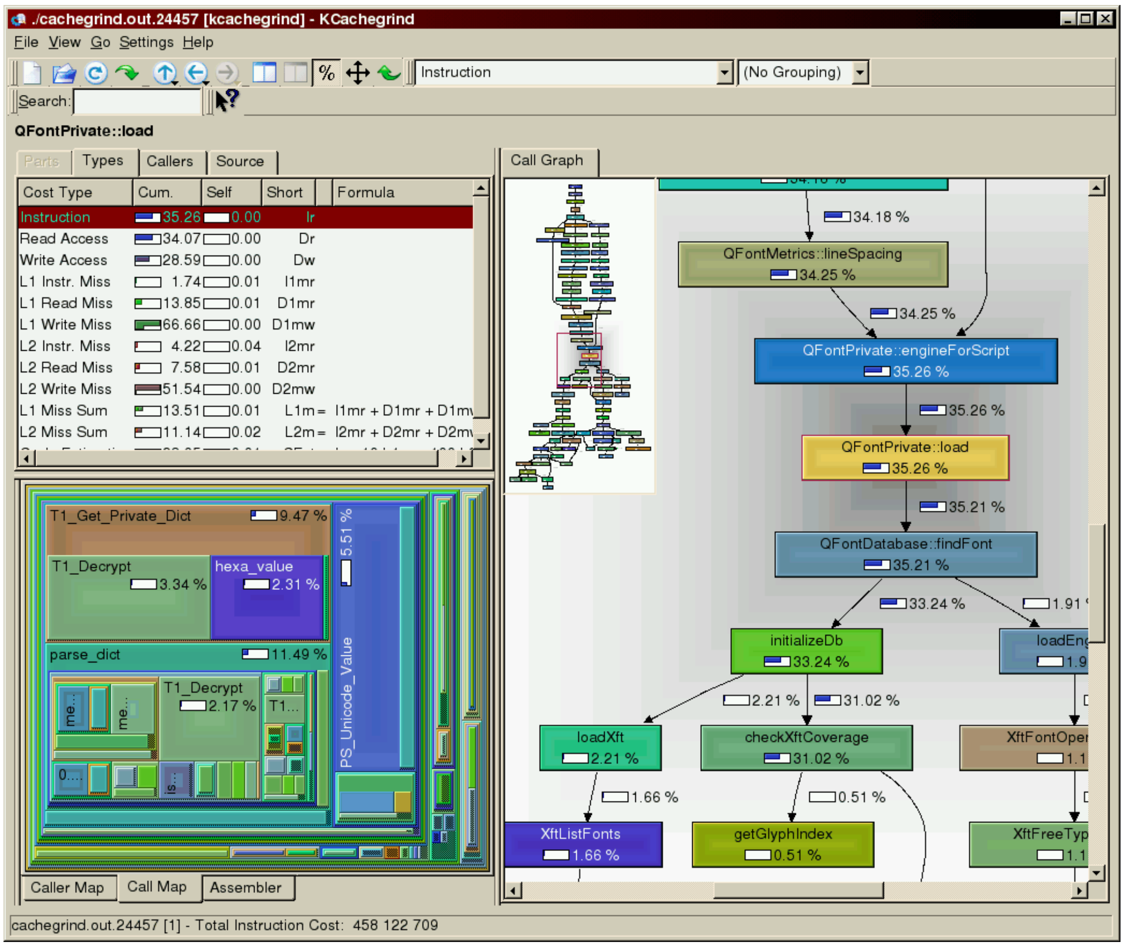


Figure 2 Gource displaying a repository

The following tools are more relevant because they have features that extract and display call-graphs or have other facilities that make understanding the behavior of a program easier.
However few have this goal as the sole purpose, most focus on profiling in order to find performance bottlenecks in applications or to find the cause of excessive resource consumption.

KCacheGrind is a visualization frontend for the Callgrind profiling tool, which is part of the well known Valgrind tool suite and is a part of the KDE project. This tool visualizes all function calls reported by Callgrind in 2D. In order to extract meaningful data the binaries must be built with debug symbols. It’s available only for linux systems.

It’s more of a profiling tool than a source code comprehension aider. It’s designed for C/C++ programs, but converters exist for PHP, Perl and Python applications.
The only thing it has in common with the Symbion project is that it uses profiling and runtime analysis to extract call-graphs.



Cost Type	Cum.	Self	Short	Formula
Instruction	35.26	0.00	Ir	
Read Access	34.07	0.00	Dr	
Write Access	28.59	0.00	Dw	
L1 Instr. Miss	1.74	0.01	I1mr	
L1 Read Miss	13.85	0.01	D1mr	
L1 Write Miss	66.66	0.00	D1mw	
L2 Instr. Miss	4.22	0.04	I2mr	
L2 Read Miss	7.58	0.01	D2mr	
L2 Write Miss	51.54	0.00	D2mw	
L1 Miss Sum	13.51	0.01	L1m =	I1mr + D1mr + D1mw
L2 Miss Sum	11.14	0.02	L2m =	I2mr + D2mr + D2mw

Figure 3 KCachegrind displaying a 2D callgraph

jSonde is a Java profiler which can attach to running processes and profile method calls, processor and memory usage, show .jar dependencies and builds reports. It can also generate UML Sequence Diagrams on the fly, during execution. Features include:

- Attach to running Java processes without restarting them.
- Generate sequence diagrams.
- Profile method calls.
- Profile memory usage.
- Show .jar files dependencies.

It’s relevant since it generates sequence diagrams which aid in understanding the interactions between objects, while this is standard UML and it’s a really nice feature to be able to generate these diagrams; they are far too detailed for our goal of aiding source code comprehension. UML Sequence Diagrams are a necessary part of the documentation and fit nicely into the general architecture of the application, they complement existing diagrams, but my opinion is that they are not enough on their own. In complex systems with thousands of classes, even if we reduce the diagrams only to the portion of the system that we are profiling, they can become too complex, therefore our application uses a different approach to source code comprehension instead of generating UML. Many CASE tools have features to generate various UML diagrams from static source code analysis and these are widely adopted and used in the field of software engineering. Still the novelty of this tool is that it enables the user to narrow down the number of generated diagrams and to focus on specific behavior.

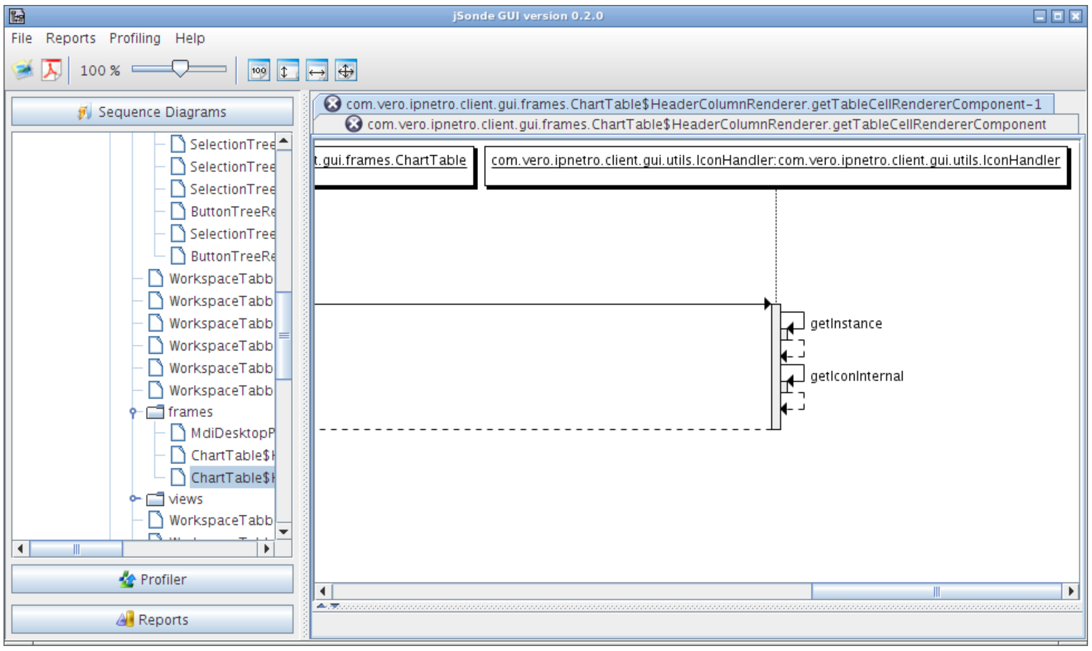
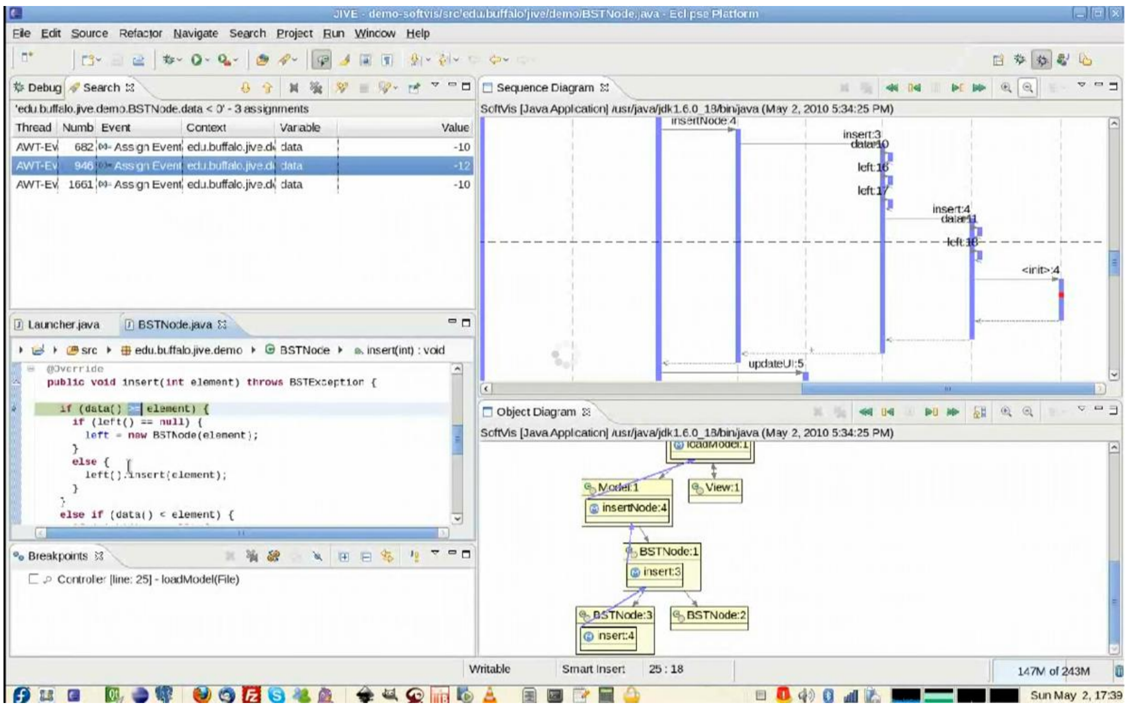

jSonde GUI version 0.2.0
File Reports Profiling Help
100 %
Sequence Diagrams
com.vero.ipnetro.client.gui.frames.ChartTable$HeaderColumnRenderer.getTableCellRendererComponent-1
com.vero.ipnetro.client.gui.frames.ChartTable$HeaderColumnRenderer.getTableCellRendererComponent
.gui.frames.ChartTable
com.vero.ipnetro.client.gui.utils.IconHandler:com.vero.ipnetro.client.gui.utils.IconHandler
getInstance
getIconInternal
frames
views
Profiler
Reports

Figure 4 jSonde displaying UML Sequence Diagrams

JIVE is an interactive execution environment for Eclipse that provides a rich visualization of the execution of Java programs. JIVE can be used both as a debugger for incorrect programs as well as for gaining a deeper insight into the behavior of correct programs. It also serves as a pedagogic tool for teaching object-oriented programming, and has been successfully used in undergraduate and graduate courses. JIVE extends the Java debugging features provided by Eclipse with interactive visualizations, query-based debugging, and reverse stepping. It depicts both the runtime state and call history of a program in a visual manner.

The runtime state is visualized as an enhanced object diagram, showing object structure as well as method activations in their proper object contexts.
The call history is depicted as an extended sequence diagram, clarifying the object interactions that occur at runtime. The diagrams are scalable and can be filtered.



'edu.buffalo.jive.demo.BSTNode.data < 0' - 3 assignments

Thread	Numb	Event	Context	Variable	Value
AWT-Ev	682	Assign Event	edu.buffalo.jive.d	data	-10
AWT-Ev	946	Assign Event	edu.buffalo.jive.d	data	-12
AWT-Ev	1661	Assign Event	edu.buffalo.jive.d	data	-10

SoftVis [Java Application] /usr/java/jdk1.6.0_18/bin/java (May 2, 2010 5:34:25 PM)

src ▸ edu.buffalo.jive.demo ▸ BSTNode ▸ insert(int) : void

```
@Override
public void insert(int element) throws BSTException {

  if (data() >= element) {
    if (left() == null) {
      left = new BSTNode(element);
    }
    else {
      left().insert(element);
    }
  }
  else if (data() < element) {
```

Controler [line: 25] - loadModel(File)

Figure 5 Jive displaying diagrams

TraceCrawler is a Java runtime visualization tool which visualizes the behavior of a java program in 3D, based on object instances and relationships [13].
It proposes an interactive "*3D visualization technique based on a combination of static and dynamic analysis which enables the software developer to step through visual representations of execution traces. We visualize dynamic behaviors of execution traces in terms of object creations and interactions and represent this in the context of a static class hierarchy view of a system* [13]."
It is similar to Symbion in the way it focuses on usage scenarios and captures execution traces. *"We establish the relationship between the features and software entities by exercising the features or usage scenarios and capturing their execution traces, which we refer to as feature-traces. A feature-trace is a sequence of runtime events (e.g., object creation/deletion, method invocation) that describes the dynamic behavior of a feature* [13]. "
It uses the TraceScraper dynamic analysis tool which builds on method wrappers to instrument the code. Symbion uses AOP techniques to insert bytecode and invoke callback

functions. The paper does not describe any interaction techniques with the profiler at runtime, it seems to record the entire execution.

Filtering mechanisms are not implemented since the article describes it as a to-do feature. It also uses an entirely different visualization model which focuses on the number of instances and on the relationships between their classes.

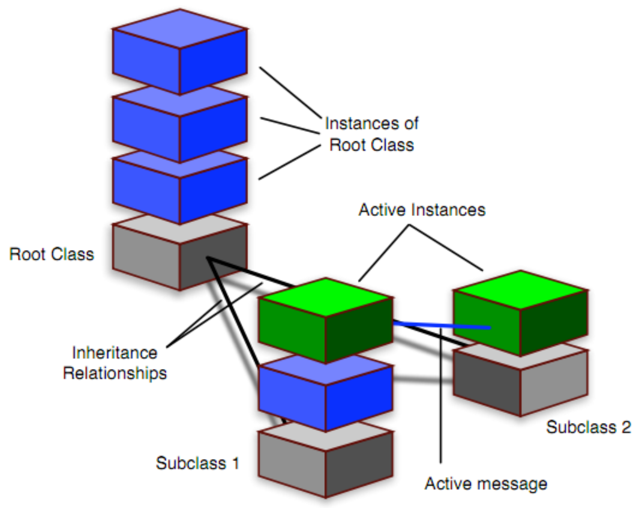


Figure 6 the visualization model used by TraceCrawler

Apart from the published article, TraceCrawler seems to be a private project, no downloads could be found, so further inspection with the purpose of comparison was not possible.

VisMOOS is a Java source code visualization tool, which statically analyzes java source code and generates a structural 3D model. It's implemented as an Eclipse plugin it also offers suitable filtering, it visualizes packages, class associations, hierarchy and package affiliations. It is described specifically as a code comprehension tool to aid maintenance projects. *"Our contribution to this task is to use relation-specific geometric arrangements in three-dimensional space. 3D space allows to display a large amount of data with optimal visual ingress, and using specific layout algorithms optimises space utilization with respect to both exploration and interpretation of the code structures displayed* [1]*."*

It is similar to Symbion because it does provide filtering mechanisms. It does perform well for large codebases, in the cited article the Eclipse source code is presented as a case study. It focuses on structure and visualizes the source code at the level of packages and class hierarchy, much like TraceCrawler, but unlike Symbion, which focuses specifically on understanding the code at a method level. This tool was also unavailable for further inspection.

Application of Helix Cone Tree Visualizations to Dynamic Call Graph Illustration [11] is an article published in 2004 about an unnamed tool which bares the most resemblance to the Symbion project and shares most of its goals and focuses on the same visualization model.

Much like the Symbion profiler and the Symbion-visualizer this application is composed of two parts the Java Trace Generator and the Call Graph Viewer.

The abstract of the article presents this application in the following way:

"We describe a tool that enables users to record and visualise runtime behaviour of software applications developed in Java. The execution trace, stored in the form of an XML File is visualized using 3D call graphs that are an extension of the Cone Tree information visualisation technique. This tool gives the user the ability to create several call graph views from a program's execution trace, providing additional representations of the program execution to both novice and expert programmers for the purposes of program execution analysis [11]*."*

Another similarity is that it uses XML to store the call graph. It also uses event driven monitoring to record the execution trace, but it uses its own debugger which has its own JVM and launches another JVM for the "debugee" application. In Symbion this principle is inverted, there is only one JVM, the one which contains the host application and byte code is inserted directly into the classes of the host application. The Java Trace Generator uses the Java Platform Debugger Architecture, which provides introspective access to a running virtual machine's state, class, array, interface, and primitive types, and instances of those types.

This approach seems slower than the one chosen in Symbion, where the instrumentation API provides access directly to the context of the host application and the profiler is running from within the profiled application, but the java.lang.instrument API was not available at the time of the development of the Java Trace Generator.

This project also delegates the task of visualizing the data to a separate application, the java Call Graph Viewer and also builds dynamic call-graphs from the gathered data.

This project justifies the usage of the Helix Cone tree in the following way:

"2-Dimensional call graph representations, while being an extremely useful and intuitive graph for the representation of small execution traces; are highly restricted in their ability to efficiently represent large data sets in finite display space.

The Cone Tree (Robertson et al., 2003) graph visualization technique uses a 3-Dimensional rendering space to display hierarchies of information. The interactive nature of Cone Trees and this expansion of the traditional tree layout into the third dimension provide a significant *improvement in the quantity of data displayable in a finite display space. Cone Trees however are not without their problems (Cockburn and McKenzie, 2000) specifically the occlusion of nodes and text labels within sub graphs. The Helix Cone Tree (Figure 3) in an effort to alleviate these problems cuts and stretches the base of a cone along the y axis into a constrained helix thus reducing the amount of occlusion of nodes and text labels in a particular sub graph. This altering of the vertical positioning of nodes within a sub graph in a predictable order introduces an additional dimension of information to the traditional Cone Tree. This additional dimension can be mapped to various attributes of a dataset but in the CGV is used to represent the implicit chronological element present in program execution data [11].”*

Symbion uses a very similar visualization model; it also focuses on the chronological order of functions and also had to deal with the same problems, but it has chosen different solutions which will be detailed in Chapter IV.
Unfortunately the article does not describe any kind of searching or filtering mechanisms to ease the navigation in 3D space or to reduce the amount of information displayed at a time. Symbion has mechanisms to deal with these issues.
As the other two applications before it, this application is also inaccessible.

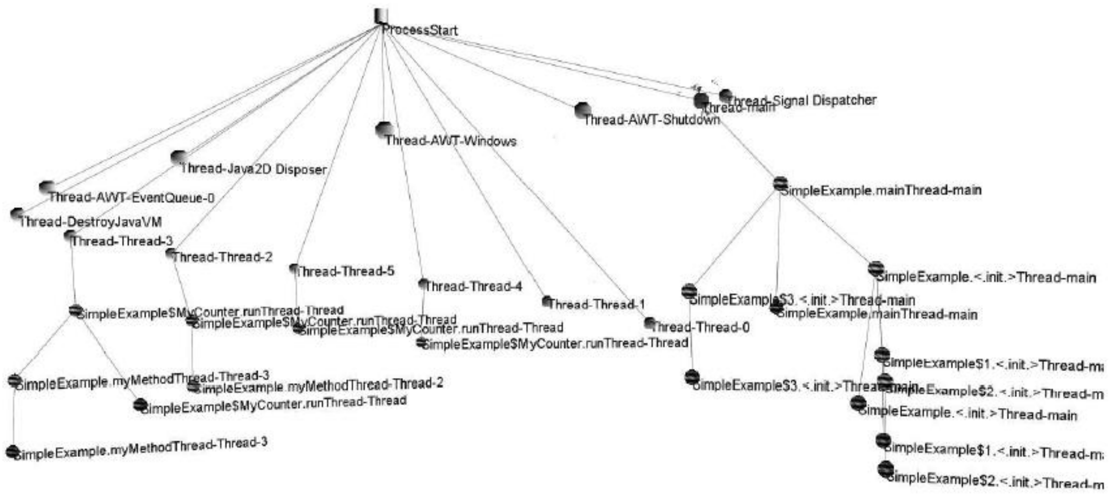

ProcessStart
Thread-Signal Dispatcher
Thread-main
Thread-AWT-Shutdown
Thread-AWT-Windows
Thread-Java2D Disposer
Thread-AWT-EventQueue-0
Thread-DestroyJavaVM
Thread-Thread-3
Thread-Thread-2
Thread-Thread-5
Thread-Thread-4
Thread-Thread-1
Thread-Thread-0
SimpleExample.mainThread-main
SimpleExample.<.init.>Thread-main
SimpleExample$3.<.init.>Thread-main
SimpleExample.mainThread-main
SimpleExample$MyCounter.runThread-Thread
SimpleExample.myMethodThread-Thread-3
SimpleExample.myMethodThread-Thread-2
SimpleExample$1.<.init.>Thread-main
SimpleExample$2.<.init.>Thread-main

Figure 7 a call-graph shown in a Helix Cone Tree

Prior art does exist, research has been done in this domain, yet no usable, accessible and functional tool has been found so we could directly compare it to Symbion in terms of performance and usability. One main difference that Symbion focuses on and no other project listed above implemented is access to the source code directly from the 3D environment.
Symbion lets you view the source of any selected function, lets you browse all the displayed classes, provides syntax highlighting and allows you to highlight search terms within the source code. The visual metaphors are directly bound to their respective source files, therefore allowing the user to deduct all the aspects of the software that the visualization failed to cover.

Another very important aspect about Symbion is the concept of focusing only on the relevant parts. This is the recommended mindset.
Only by using runtime analysis we reduce the dataset significantly, since we only record the function calls that executed. Moreover the profiling can be started and stopped at any time in a program’s execution. The user can also give rules to either accept or reject classes from certain packages; these rules can be given in the form of regular expressions. This reduces the dataset even more and allows granular control of the profiling process, filtering out classes and methods that are out of scope for the current analysis session. The classes that do not match rules or those that do match rules but also match rules that exclude them don’t get profiled. This reduces the amount of gathered data and the performance penalty imposed on the host application significantly. This allows the exclusion of the method calls from or to third party libraries or to other code that is out of the scope of the analysis session. This filtering and exclusion principle does not only apply to the profiler, it governs the entire visualization process. All function calls are expanded from an initial start node, which when clicked expands into the roots of the separate call trees, each node expands further into its invoked methods. By making the user expand only the sub-tree that interests him, we significantly reduce the amount of information that is displayed at a time and offer the freedom to choose the areas of interest. The expanding of the nodes also follows the chronological order of the host programs execution. A node for a method is displayed, every time the method is called.

Windowing is implemented to handle the case when a node expands into too many child nodes. The user can always set a range for the nodes he’s interested in (i.e. he can expand the first 10, the second 10 or the third 5, etc) he is in total control of the expanding process. The nodes do expand in random directions but ray querying algorithms are used to ensure that two nodes do not expand over each other; furthermore the expansion algorithm favors directions that expand outwards from the parent node.

The nodes and the arrows between them are color coded. Each class in the application is assigned a color; a legend is presented which also allows the user to view the source code of the class. Profiling and visualization can be repeated without closing the console or the host application, therefore allowing a gradual, visual exploration of the analyzed codebase through multiple execution scenarios.
We can search nodes by name and the camera will focus on them. Function nodes can also be highlighted and their source code is displayed in a 2D transparent overlay, the source code of the whole class is displayed but we automatically scroll to the definition of the selected function, so we can instantly view the part of the source code that caused this behavior, syntax highlighting is provided to increase readability. All functionality and navigation controls is presented to the user in 2D overlays containing text fields, buttons and other control elements, similar to a head-up display (HUD).

I have chosen to present the most relevant projects based on similarity to this project and based on shared goals. Other studies deal with the comparison of similar tools in greater detail [17].

3. Technologies used

During the implementation phase of the application we were presented with many solutions that satisfy our needs and exploring them is recommended, in order to choose the best fit. We will present each of them in the following sections and compare them, while justifying our design decisions and focusing on the needs of such an application.

One of the biggest architectural decisions was to present the gathered data in 3-D space. Visualizing in 3D allows the representation of more data and provides better methods of organization than 2D, which becomes very hard to navigate for a large dataset. 3D is only confusing when we lack reference. Open world games represent huge maps without the player ever getting lost. This is usually accomplished by providing a 2D map on which the user can locate items in 3D space. If proper navigation is implemented and movement is constrained according to the surroundings 3D engines provide a superior way to view and interact with a great number of elements, a number that 2D simply can't handle efficiently.
Therefore we have a clear need for a good 3D engine, so we chose the jMonkey Engine, which basically is a full-featured game engine. This may seem as an unusual application of a game engine, but others have explored similar ideas [3].

jMonkey is written purely in Java and uses LWJGL as its default renderer, OpenGL 2 through OpenGL 4 is fully supported. JME3 has been gaining popularity since the Beta SDK was released in the October of 2011; it supports shaders, particle effects, post-processing filters, and the majority of popular formats, has the Bullet physics engine integrated and provides 2D Graphical User interface components through the Nifty GUI library. It also has limited support for the Android mobile platform.
jMonkeyEngine is a community-centric open source project released under the new BSD license. It is used by several commercial game studios and educational institutions.
By itself, it's a collection of libraries, making it a low-level game development tool. It has an integrated SDK based on Netbeans, but we chose not to use it, because we want to develop all our components in the Eclipse environment, so we just use the libraries.

Advantages:

- Full featured game engine, also provides physics library in the event that we decide to incorporate physics in our visualization.

- It’s entirely written in Java and integrates well with the rest of the project; it allows us to use it with Maven, therefore always keeping it up to date. Maven provides very comfortable dependency management for all components; it has plugins like the maven-assembly-plugin which allows us to build one jar with all the dependencies. It eases the process of building, testing, deploying and distributing the project. We can build the entire project with all the modules in a single command.
- Provides acceptable performance in our case.
- Cross platform, without needing native libraries or different versions for different operating systems. It’s written in Java, therefore it runs anywhere.
- Nifty GUIs XML based approach to building layouts for 2D overlays obeys the MVC principle and allows us to keep our logic separate from the presentation.

Disadvantages:

- Scarce documentation and outdated examples, lacking tutorials.
- Slower than native libraries.
- Nifty GUI severely lacks features, not only compared to desktop windowing toolkits but to other widget libraries for 3D engines. It lacks basic widgets like the textarea so most features need to be implemented from scratch.

To address these issues other engines well also considered, like the OGRE 3D engine.
The **Object-Oriented Graphics Rendering Engine (OGRE 3D)** is written in C++ and is also cross platform. Cross platform in this context means that the libraries are available for more platforms but different builds have to be made on each platform and they have to be maintained separately.
OGRE has a very active community, and was SourceForge's project of the month in March 2005. It has been used in some commercial games such as Ankh, Torchlight and Garshasp.

Advantages

- Better documentation, active community, more mature code.
- Better rendering capabilities

- Increased performance, due mostly to the fact that it consists of native libraries.
- It uses CEGUI for 2D overlay widgets, which is also more mature and has more features than nifty.

Disadvantages

- Relies on different third party libraries (CEGUI, Bullet, etc) and lacks an automated build system on Windows (this is available for Linux in the form of CMAKE, but CMAKE has no dependency management while Maven does). Since the profiler is bound to Java technologies and building a C++ native profiler is both out of scope for the project and very difficult, especially if we want it to be cross platform this poses a clear development and maintenance overhead that is very undesirable. The advantages of using Maven outweigh the advantages OGRE 3D would bring to this project.

jMonkeyEngine is up to the task of visualizing our limited number of elements and integrates very well with all the different components of the project.

Building the profiler also posed a set of challenges. The java.lang.instrument API was chosen because it was specifically designed and implemented with such applications in mind.
"Available since the Java version 1.5.0, the package java.lang.instrument [9] defines a standard API for instrumenting the bytecode of classes before they're loaded by the virtual machine, as well as redefining the bytecode of classes that have already been loaded by the JVM [2]."
Bytecode instrumentation is defined as the process of adding new functionality to a program by modifying the bytecode of a set of classes before they are loaded by the virtual machine.
Bytecode instrumentation is often not about adding new functionality per se, but changing a program temporarily with the goal of measuring or recording some aspects of its execution.
A clear example of the application of bytecode instrumentation is the bytecode functionality of a debugger. Bytecode instrumentation is also often used to implement AOP functionalities.

Aspect-oriented programming (AOP) is a programming paradigm whose name was coined by the AspectJ research team at the Palo Alto Research Center around 1995.
The problem that AOP tries to address is that modern object-oriented systems often have code that handles multiple concerns on the same level of code. It defines the concept of cross cutting concerns, concerns that cut through multiple tiers of the application and mix with core

concerns (business logic). This problem hurts encapsulation and is a cause of high coupling and low cohesion. Examples of these cross cutting concerns include security, logging, and transaction management. AOP tries to extract and separate these functionalities into so-called reusable aspects that are implemented in one place and applied upon multiple cutpoints throughout the application. This is usually done by intercepting function calls and inserting bytecode.
The same AOP concepts apply to our profiler, profiling is a cross cutting concern, the rules given by the users to match functions are cutpoints and our data gathering functions are aspects applied to them.

"This type of problem is a perfect candidate for an aspect-oriented solution. The agent interface introduced in Java 5 is ideal for building an aspect-oriented profiler because it gives you an easy way to hook into the classloader and modify classes as they're loaded. [18] *"*

To ease our work, we use a Bytecode Engineering Library, since it provides the necessary abstraction levels to minimize the need to write Java bytecode directly.
After some investigation both Javassist and Apache BCEL (Byte Code Engineering Library) were considered, but **ObjectWeb ASM** was chosen because of its performance and simplicity.
The main focus when choosing a bytecode engineering library is speed.

Table 1 Class construction times in different bytecode engineering libraries.

Framework	**First time**	**Later times**
Javassist	257	5.2
BCEL	473	5.5
ASM	62.4	1.1

The conclusions of a comparative study between ASM, Javassist and BCEL were:
"Matching ASM up against the other classworking frameworks shows that it's several times faster than the others (at least for this one fairly typical test case). It's also much more compact, with the run-time JAR weighing in at only 33K (versus 310K for Javassist and a whopping 504K for BCEL). Ease of use is trickier to determine, but the interface seems significantly cleaner than BCEL's while offering almost as much flexibility (lacking only some unique features of BCEL, such as the ability to construct code in segments rather than

linearly). ASM is still not as easy to use as Javassist with its Java-like source code interface, but if you want to work at the bytecode level, I'd recommend you look into using ASM. [16]"
These benchmarks were done with ASM 2.0, and ASM has since improved, so these results still remain relevant.
Another good reason to choose ASM is the ASM Bytecode Outline Eclipse plugin which displays the bytecode of java classes and source code portions; this eases the construction of the injected bytecode significantly. ASM also supports modern Java 7 specific instructions like invokedynamic. It is also the bytecode manipulation framework used by the AspectWerkz 2 AOP framework. Also it is very well documented and available under a BSD license. The conclusion is that there are many advantages to choosing ObjectWeb ASM.

ObjectWeb ASM uses the visitor design pattern to expose the internal aggregate components of a given Java Class. It provide two APIs, one having an event based model, representing a class as a sequence of events, each event representing the an element of a class, and one having an object based model, where a class is represented as a tree of objects, each object representing the part of a class.
"These two APIs can be compared to the Simple API for XML (SAX) and Document Object Model (DOM) APIs for XML documents: the event based API is similar to SAX, while the object based API is similar to DOM. The object based API is built on top of the event based one, like DOM can be provided on top of SAX. ASM provides both APIs because there is no best API. Indeed each API has its own advantages and drawbacks: The event based API is faster and requires less memory than the object based API, since there is no need to create and store in memory a tree of objects representing the class (the same difference also exists between SAX and DOM). However implementing class transformations can be more difficult with the event based API, since only one element of the class is available at any given time (the element that corresponds to the current event), while the whole class is available in memory with the object based API. [5]"
We use the SAX-like API which has a noticeably reduced memory footprint.

For implementing the user interface, the **SWT** (Standard Widget Toolkit) was chosen. The main reason for this is to ease an eventual integration as an Eclipse plugin, since it is easier for developers to access these kinds of tools directly from within their development environment, but we still would like to provide the application in a standalone form to aid developers who use other popular development environments like InteliJ IDEA or Netbeans.
SWT was originally developed by Stephen Northover at IBM and is now maintained by the Eclipse Foundation in tandem with the Eclipse IDE. It is an alternative to the Abstract

Window Toolkit (AWT) and Swing Java GUI toolkits provided by Sun Microsystems as part of the Java Platform, Standard Edition. It's an essential part of the Eclipse Platform.
The main advantage it has over Swing is that it uses native libraries using the JNI (Java Native Interface) and that it looks native under all operating systems.
Although SWT does not follow MVC principles like Swing does, we use the **JFace** library, developed as part of the Eclipse project and implements a higher-level Model-View-Controller abstraction on top of SWT.

The **Spring Framework** is also used mainly because of their Jaxb2Marshaller implementation which is part of Spring OXM (Object XML Mappers) and because of the Spring RMI implementation. The Jaxb2Marshaller is used to define the format of the XML document containing the gathered data, this is desirable, since the gathered data can be quite large and therefore we can't use XMLEncoder/XMLDecoder.
Spring RMI is needed for communication between the host application and the server; there are reasons for this, which will be detailed in the next chapter. Spring RMI provides nice abstractions over traditional RMI and makes it a lot easier to use.

Apache Maven is a build automation tool typically used for Java projects. Maven serves a similar purpose to the Apache Ant tool, but it is based on different concepts and works in a profoundly different manner. Maven uses an XML file (pom.xml, from Project Object Model) to describe the structure of the project, the dependencies, metadata, build order and required plugins. It automatically compiles and packages the project and the maven-assembly-plugin can be used to include all the dependencies into a single jar.
It keeps all the artifacts in a local repository. These are downloaded automatically from central repositories.

Blender was used to create the models of the nodes and the arrows and **Log4j** was used for logging.

4. Implementation

We will present the implementation process separately for all the modules of the application. These modules are all separate maven projects.

4.1 Architecture

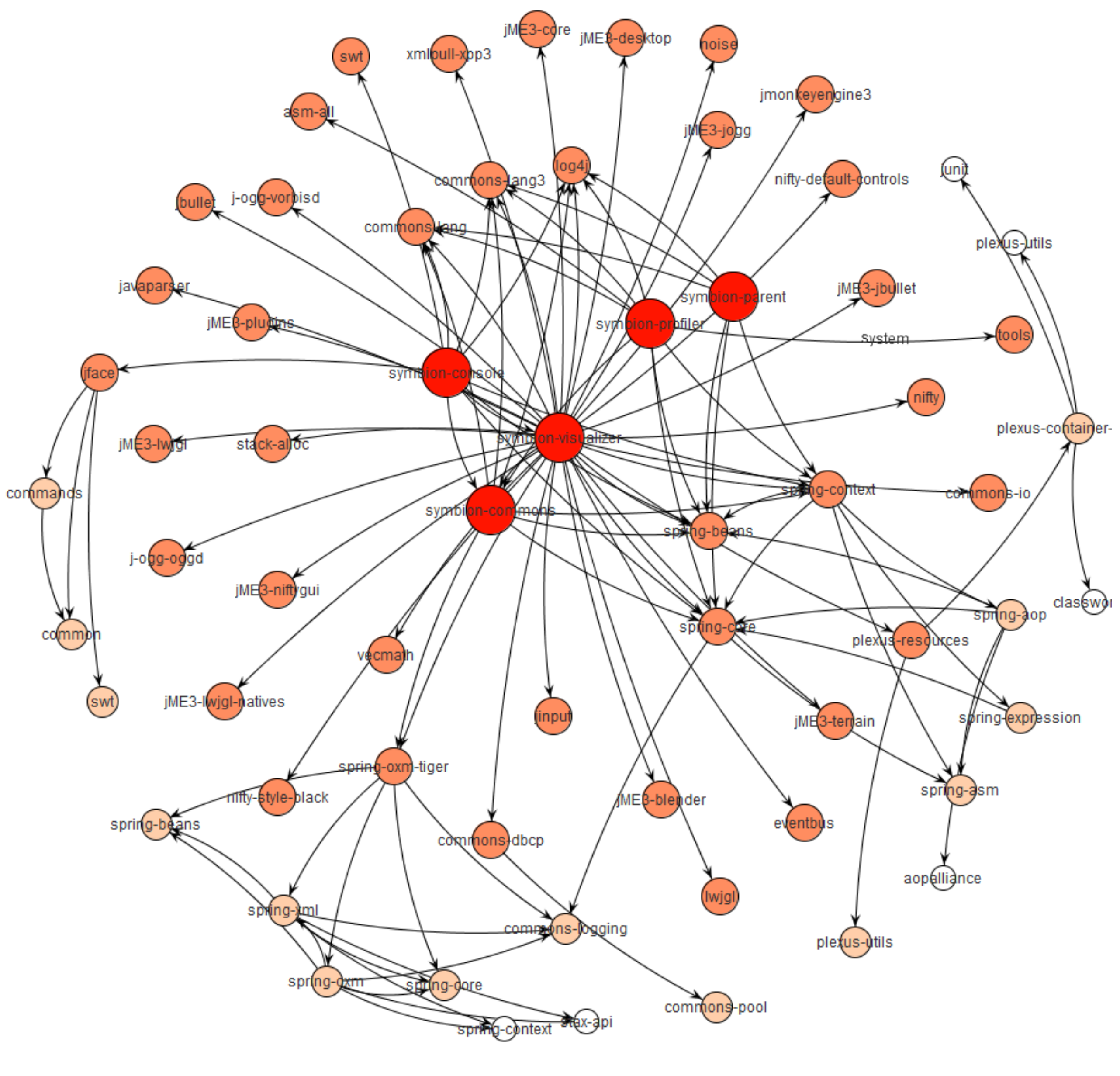


Figure 8 Overview of the dependency graph of all the modules

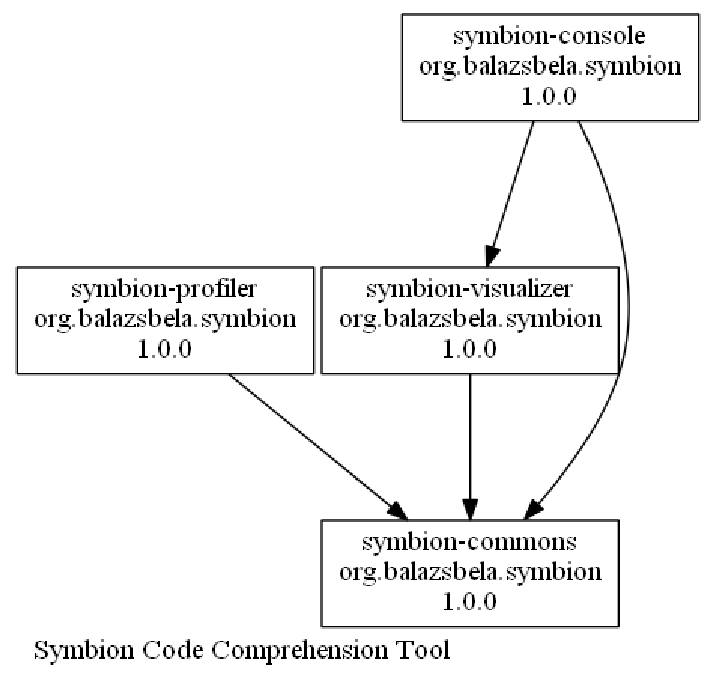


Figure 9 Dependency graph between the modules of the project

The project is structured into the following sub-modules:

symbion-commons : containing common elements, used by all the modules.

symbion-console: containing the client application that connects the the profilers TCP server and manages the interaction with the profiler, also starts the visualizer.

symbion-visualizer: the visualizer application.

symbion-profiler: the module containing the profiling agent and the TCP server which is used to communicate with it.

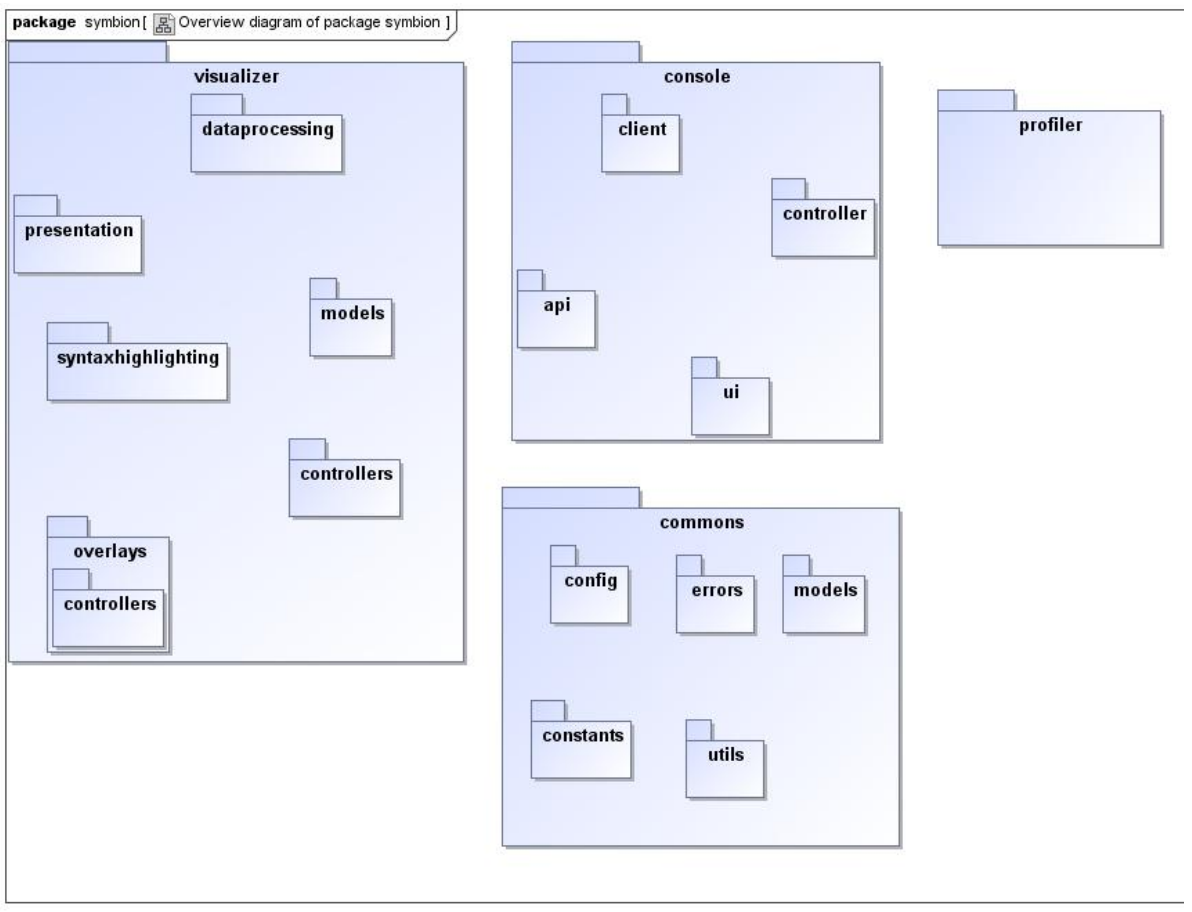


Figure 10 Package overview

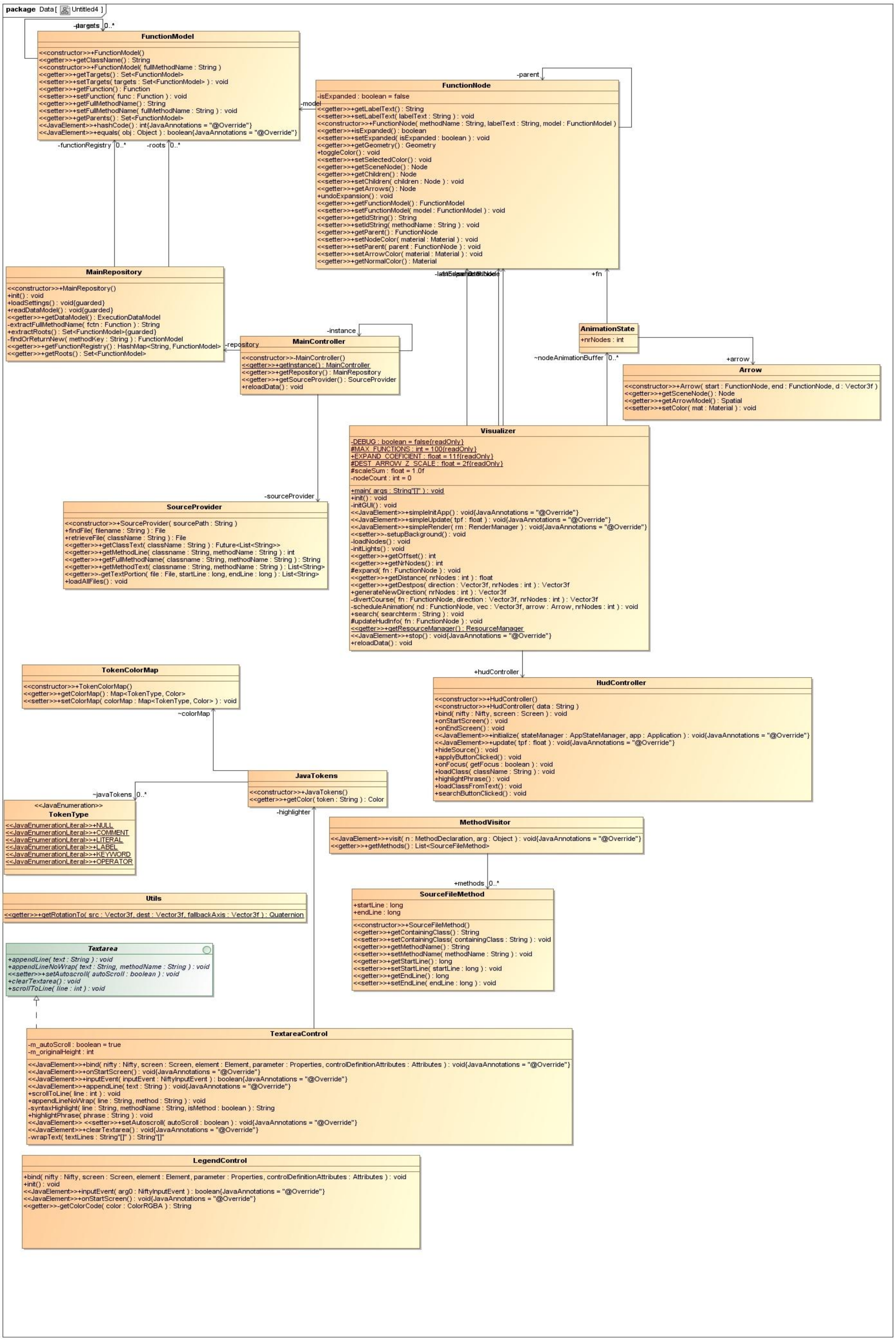

package Data[Untitled4]
-targets 0..*
-parents
FunctionModel
<<constructor>>+FunctionModel()
<<getter>>+getClassName() : String
<<constructor>>+FunctionModel(fullMethodName : String)
<<getter>>+getTargets() : Set<FunctionModel>
<<setter>>+setTargets(targets : Set<FunctionModel>) : void
<<getter>>+getFunction() : Function
<<setter>>+setFunction(func : Function) : void
<<getter>>+getFullMethodName() : String
<<setter>>+setFullMethodName(fullMethodName : String) : void
<<getter>>+getParents() : Set<FunctionModel>
<<JavaElement>>+hashCode() : int{JavaAnnotations = "@Override"}
<<JavaElement>>+equals(obj : Object) : boolean{JavaAnnotations = "@Override"}
-functionRegistry 0..*
-roots 0..*
-model
-parent
FunctionNode
-isExpanded : boolean = false
<<getter>>+getLabelText() : String
<<setter>>+setLabelText(labelText : String) : void
<<constructor>>+FunctionNode(methodName : String, labelText : String, model : FunctionModel)
<<getter>>+isExpanded() : boolean
<<setter>>+setExpanded(isExpanded : boolean) : void
<<getter>>+getGeometry() : Geometry
+toggleColor() : void
<<setter>>+setSelectedColor() : void
<<getter>>+getSceneNode() : Node
<<getter>>+getChildren() : Node
<<setter>>+setChildren(children : Node) : void
<<getter>>+getArrows() : Node
+undoExpansion() : void
<<getter>>+getFunctionModel() : FunctionModel
<<setter>>+setFunctionModel(model : FunctionModel) : void
<<getter>>+getIdString() : String
<<setter>>+setIdString(methodName : String) : void
<<getter>>+getParent() : FunctionNode
<<setter>>+setNodeColor(material : Material) : void
<<setter>>+setParent(parent : FunctionNode) : void
<<setter>>+setArrowColor(material : Material) : void
<<getter>>+getNormalColor() : Material
+fn
MainRepository
<<constructor>>+MainRepository()
+init() : void
+loadSettings() : void{guarded}
+readDataModel() : void{guarded}
<<getter>>+getDataModel() : ExecutionDataModel
-extractFullMethodName(fctn : Function) : String
+extractRoots() : Set<FunctionModel>{guarded}
-findOrReturnNew(methodKey : String) : FunctionModel
<<getter>>+getFunctionRegistry() : HashMap<String, FunctionModel>
<<getter>>+getRoots() : Set<FunctionModel>
-instance
-repository
MainController
<<constructor>>-MainController()
<<getter>>+getInstance() : MainController
<<getter>>+getRepository() : MainRepository
<<getter>>+getSourceProvider() : SourceProvider
+reloadData() : void
AnimationState
+nrNodes : int
~nodeAnimationBuffer 0..*
+arrow
Arrow
<<constructor>>+Arrow(start : FunctionNode, end : FunctionNode, d : Vector3f)
<<getter>>+getSceneNode() : Node
<<getter>>+getArrowModel() : Spatial
<<setter>>+setColor(mat : Material) : void
-sourceProvider
SourceProvider
<<constructor>>+SourceProvider(sourcePath : String)
+findFile(filename : String) : File
+retrieveFile(className : String) : File
<<getter>>+getClassText(className : String) : Future<List<String>>
<<getter>>+getMethodLine(classname : String, methodName : String) : int
<<getter>>+getFullMethodName(classname : String, methodName : String) : String
<<getter>>+getMethodText(classname : String, methodName : String) : List<String>
<<getter>>-getTextPortion(file : File, startLine : long, endLine : long) : List<String>
+loadAllFiles() : void
Visualizer
-DEBUG : boolean = false{readOnly}
#MAX_FUNCTIONS : int = 100{readOnly}
+EXPAND_COEFICIENT : float = 11f{readOnly}
#DEST_ARROW_Z_SCALE : float = 2f{readOnly}
#scaleSum : float = 1.0f
-nodeCount : int = 0
+main(args : String"[]") : void
+init() : void
-initGUI() : void
<<JavaElement>>+simpleInitApp() : void{JavaAnnotations = "@Override"}
<<JavaElement>>+simpleUpdate(tpf : float) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+simpleRender(rm : RenderManager) : void{JavaAnnotations = "@Override"}
<<setter>>-setupBackground() : void
-loadNodes() : void
-initLights() : void
<<getter>>+getOffset() : int
<<getter>>+getNrNodes() : int
#expand(fn : FunctionNode) : void
<<getter>>+getDistance(nrNodes : int) : float
<<getter>>+getDestpos(direction : Vector3f, nrNodes : int) : Vector3f
+generateNewDirection(nrNodes : int) : Vector3f
-divertCourse(fn : FunctionNode, direction : Vector3f, nrNodes : int) : Vector3f
-scheduleAnimation(nd : FunctionNode, vec : Vector3f, arrow : Arrow, nrNodes : int) : void
+search(searchterm : String) : void
#updateHudInfo(fn : FunctionNode) : void
<<getter>>+getResourceManager() : ResourceManager
<<JavaElement>>+stop() : void{JavaAnnotations = "@Override"}
+reloadData() : void
+hudController
HudController
<<constructor>>+HudController()
<<constructor>>+HudController(data : String)
+bind(nifty : Nifty, screen : Screen) : void
+onStartScreen() : void
+onEndScreen() : void
<<JavaElement>>+initialize(stateManager : AppStateManager, app : Application) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+update(tpf : float) : void{JavaAnnotations = "@Override"}
+hideSource() : void
+applyButtonClicked() : void
+onFocus(getFocus : boolean) : void
+loadClass(className : String) : void
+highlightPhrase() : void
+loadClassFromText() : void
+searchButtonClicked() : void
TokenColorMap
<<constructor>>+TokenColorMap()
<<getter>>+getColorMap() : Map<TokenType, Color>
<<setter>>+setColorMap(colorMap : Map<TokenType, Color>) : void
~colorMap
JavaTokens
<<constructor>>+JavaTokens()
<<getter>>+getColor(token : String) : Color
~javaTokens 0..*
-highlighter
<<JavaEnumeration>>
TokenType
<<JavaEnumerationLiteral>>+NULL
<<JavaEnumerationLiteral>>+COMMENT
<<JavaEnumerationLiteral>>+LITERAL
<<JavaEnumerationLiteral>>+LABEL
<<JavaEnumerationLiteral>>+KEYWORD
<<JavaEnumerationLiteral>>+OPERATOR
MethodVisitor
<<JavaElement>>+visit(n : MethodDeclaration, arg : Object) : void{JavaAnnotations = "@Override"}
<<getter>>+getMethods() : List<SourceFileMethod>
+methods 0..*
SourceFileMethod
+startLine : long
+endLine : long
<<constructor>>+SourceFileMethod()
<<getter>>+getContainingClass() : String
<<setter>>+setContainingClass(containingClass : String) : void
<<getter>>+getMethodName() : String
<<setter>>+setMethodName(methodName : String) : void
<<getter>>+getStartLine() : long
<<setter>>+setStartLine(startLine : long) : void
<<getter>>+getEndLine() : long
<<setter>>+setEndLine(endLine : long) : void
Utils
<<getter>>+getRotationTo(src : Vector3f, dest : Vector3f, fallbackAxis : Vector3f) : Quaternion
Textarea
+appendLine(text : String) : void
+appendLineNoWrap(text : String, methodName : String) : void
<<setter>>+setAutoscroll(autoScroll : boolean) : void
+clearTextarea() : void
+scrollToLine(line : int) : void
TextareaControl
-m_autoScroll : boolean = true
-m_originalHeight : int
<<JavaElement>>+bind(nifty : Nifty, screen : Screen, element : Element, parameter : Properties, controlDefinitionAttributes : Attributes) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+onStartScreen() : void{JavaAnnotations = "@Override"}
<<JavaElement>>+inputEvent(inputEvent : NiftyInputEvent) : boolean{JavaAnnotations = "@Override"}
<<JavaElement>>+appendLine(text : String) : void{JavaAnnotations = "@Override"}
+scrollToLine(line : int) : void
+appendLineNoWrap(line : String, method : String) : void
-syntaxHighlight(line : String, methodName : String, isMethod : boolean) : String
+highlightPhrase(phrase : String) : void
<<JavaElement>> <<setter>>+setAutoscroll(autoScroll : boolean) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+clearTextarea() : void{JavaAnnotations = "@Override"}
-wrapText(textLines : String"[]") : String"[]"
LegendControl
+bind(nifty : Nifty, screen : Screen, element : Element, parameter : Properties, controlDefinitionAttributes : Attributes) : void
+init() : void
<<JavaElement>>+inputEvent(arg0 : NiftyInputEvent) : boolean{JavaAnnotations = "@Override"}
<<JavaElement>>+onStartScreen() : void{JavaAnnotations = "@Override"}
<<getter>>-getColorCode(color : ColorRGBA) : String

Figure 11 Class diagram for the Visualizer module

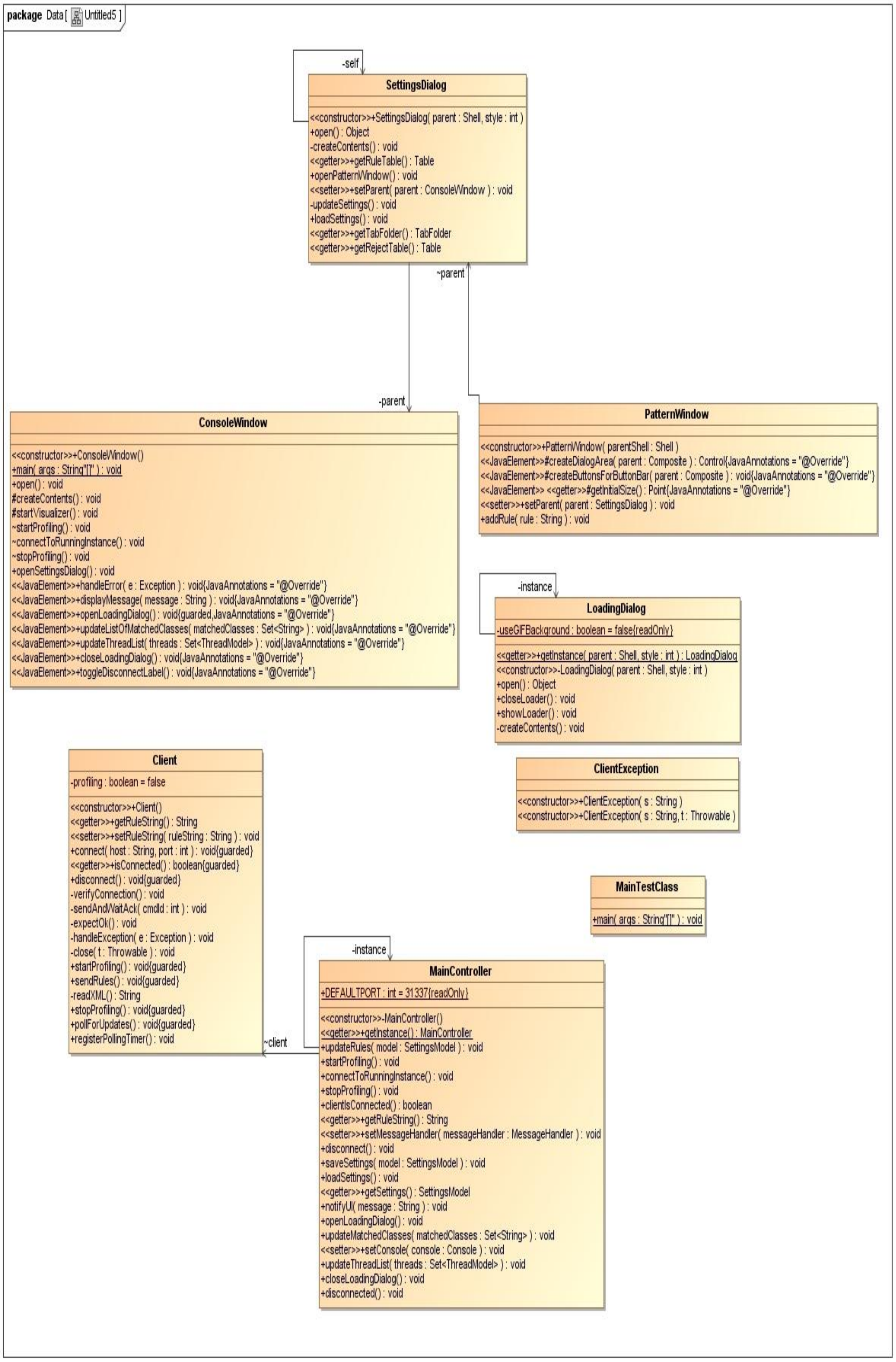

package Data[Untitled5]
-self
SettingsDialog
<<constructor>>+SettingsDialog(parent : Shell, style : int)
+open() : Object
-createContents() : void
<<getter>>+getRuleTable() : Table
+openPatternWindow() : void
<<setter>>+setParent(parent : ConsoleWindow) : void
-updateSettings() : void
+loadSettings() : void
<<getter>>+getTabFolder() : TabFolder
<<getter>>+getRejectTable() : Table
~parent
-parent
ConsoleWindow
<<constructor>>+ConsoleWindow()
+main(args : String"[]") : void
+open() : void
#createContents() : void
#startVisualizer() : void
~startProfiling() : void
~connectToRunningInstance() : void
~stopProfiling() : void
+openSettingsDialog() : void
<<JavaElement>>+handleError(e : Exception) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+displayMessage(message : String) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+openLoadingDialog() : void{guarded,JavaAnnotations = "@Override"}
<<JavaElement>>+updateListOfMatchedClasses(matchedClasses : Set<String>) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+updateThreadList(threads : Set<ThreadModel>) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+closeLoadingDialog() : void{JavaAnnotations = "@Override"}
<<JavaElement>>+toggleDisconnectLabel() : void{JavaAnnotations = "@Override"}
PatternWindow
<<constructor>>+PatternWindow(parentShell : Shell)
<<JavaElement>>#createDialogArea(parent : Composite) : Control{JavaAnnotations = "@Override"}
<<JavaElement>>#createButtonsForButtonBar(parent : Composite) : void{JavaAnnotations = "@Override"}
<<JavaElement>> <<getter>>#getInitialSize() : Point{JavaAnnotations = "@Override"}
<<setter>>+setParent(parent : SettingsDialog) : void
+addRule(rule : String) : void
-instance
LoadingDialog
-useGIFBackground : boolean = false{readOnly}
<<getter>>+getInstance(parent : Shell, style : int) : LoadingDialog
<<constructor>>-LoadingDialog(parent : Shell, style : int)
+open() : Object
+closeLoader() : void
+showLoader() : void
-createContents() : void
Client
-profiling : boolean = false
<<constructor>>+Client()
<<getter>>+getRuleString() : String
<<setter>>+setRuleString(ruleString : String) : void
+connect(host : String, port : int) : void{guarded}
<<getter>>+isConnected() : boolean{guarded}
+disconnect() : void{guarded}
-verifyConnection() : void
-sendAndWaitAck(cmdId : int) : void
-expectOk() : void
-handleException(e : Exception) : void
-close(t : Throwable) : void
+startProfiling() : void{guarded}
+sendRules() : void{guarded}
-readXML() : String
+stopProfiling() : void{guarded}
+pollForUpdates() : void{guarded}
+registerPollingTimer() : void
ClientException
<<constructor>>+ClientException(s : String)
<<constructor>>+ClientException(s : String, t : Throwable)
MainTestClass
+main(args : String"[]") : void
-instance
~client
MainController
+DEFAULTPORT : int = 31337{readOnly}
<<constructor>>-MainController()
<<getter>>+getInstance() : MainController
+updateRules(model : SettingsModel) : void
+startProfiling() : void
+connectToRunningInstance() : void
+stopProfiling() : void
+clientIsConnected() : boolean
<<getter>>+getRuleString() : String
<<setter>>+setMessageHandler(messageHandler : MessageHandler) : void
+disconnect() : void
+saveSettings(model : SettingsModel) : void
+loadSettings() : void
<<getter>>+getSettings() : SettingsModel
+notifyUI(message : String) : void
+openLoadingDialog() : void
+updateMatchedClasses(matchedClasses : Set<String>) : void
<<setter>>+setConsole(console : Console) : void
+updateThreadList(threads : Set<ThreadModel>) : void
+closeLoadingDialog() : void
+disconnected() : void

Figure 12 UML Class Diagram of the Console module

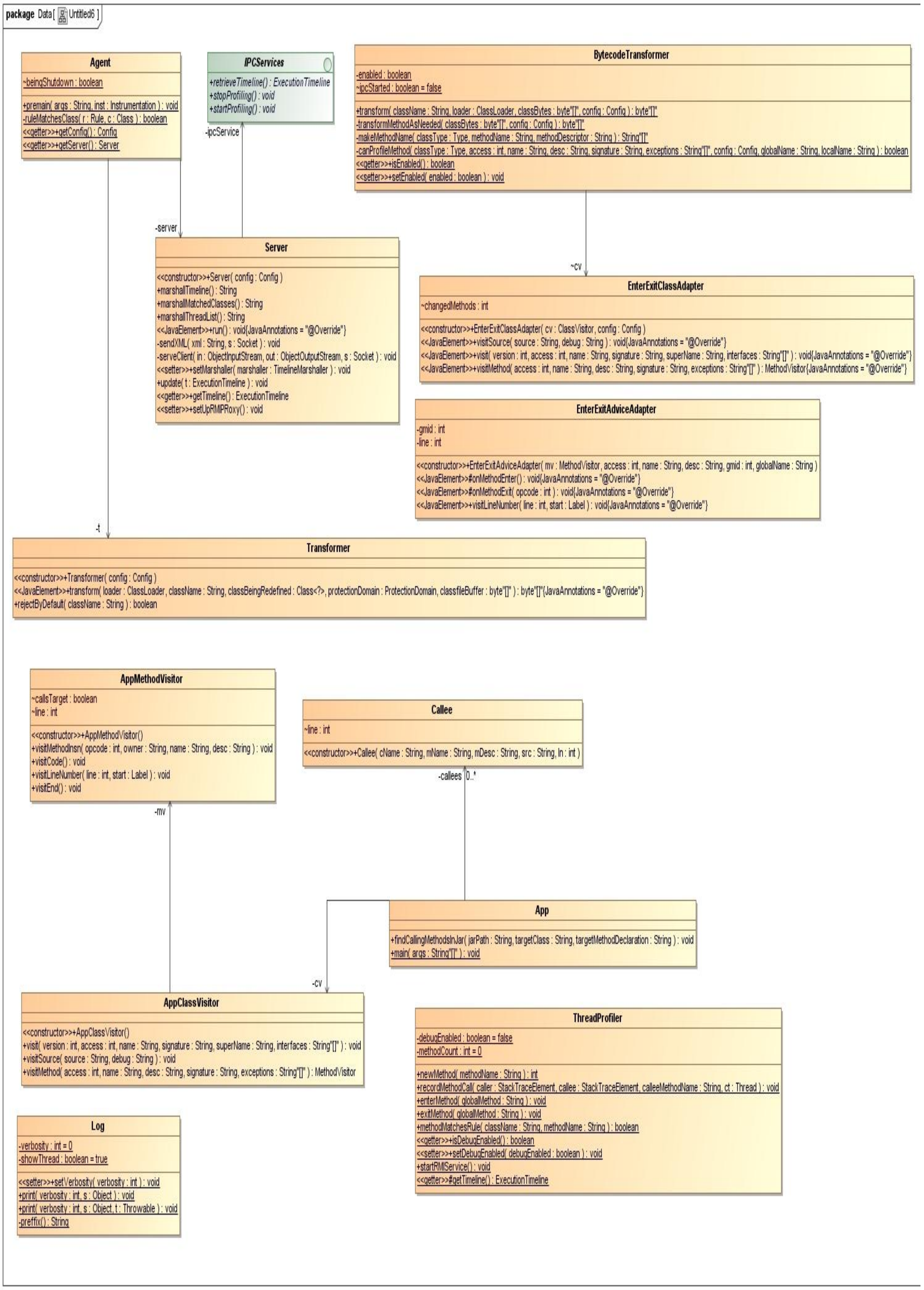

package Data[Untitled6]
Agent
~beingShutdown : boolean
+premain(args : String, inst : Instrumentation) : void
-ruleMatchesClass(r : Rule, c : Class) : boolean
<<getter>>+getConfig() : Config
<<getter>>+getServer() : Server
IPCServices
+retrieveTimeline() : ExecutionTimeline
+stopProfiling() : void
+startProfiling() : void
-ipcService
-server
Server
<<constructor>>+Server(config : Config)
+marshallTimeline() : String
+marshallMatchedClasses() : String
+marshallThreadList() : String
<<JavaElement>>+run() : void{JavaAnnotations = "@Override"}
-sendXML(xml : String, s : Socket) : void
-serveClient(in : ObjectInputStream, out : ObjectOutputStream, s : Socket) : void
<<setter>>+setMarshaller(marshaller : TimelineMarshaller) : void
+update(t : ExecutionTimeline) : void
<<getter>>+getTimeline() : ExecutionTimeline
<<setter>>+setUpRMIProxy() : void
BytecodeTransformer
-enabled : boolean
~ipcStarted : boolean = false
+transform(className : String, loader : ClassLoader, classBytes : byte"[]", config : Config) : byte"[]"
-transformMethodAsNeeded(classBytes : byte"[]", config : Config) : byte"[]"
-makeMethodName(classType : Type, methodName : String, methodDescriptor : String) : String"[]"
-canProfileMethod(classType : Type, access : int, name : String, desc : String, signature : String, exceptions : String"[]", config : Config, globalName : String, localName : String) : boolean
<<getter>>+isEnabled() : boolean
<<setter>>+setEnabled(enabled : boolean) : void
~cv
EnterExitClassAdapter
~changedMethods : int
<<constructor>>+EnterExitClassAdapter(cv : ClassVisitor, config : Config)
<<JavaElement>>+visitSource(source : String, debug : String) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+visit(version : int, access : int, name : String, signature : String, superName : String, interfaces : String"[]") : void{JavaAnnotations = "@Override"}
<<JavaElement>>+visitMethod(access : int, name : String, desc : String, signature : String, exceptions : String"[]") : MethodVisitor{JavaAnnotations = "@Override"}
EnterExitAdviceAdapter
-gmid : int
-line : int
<<constructor>>+EnterExitAdviceAdapter(mv : MethodVisitor, access : int, name : String, desc : String, gmid : int, globalName : String)
<<JavaElement>>#onMethodEnter() : void{JavaAnnotations = "@Override"}
<<JavaElement>>#onMethodExit(opcode : int) : void{JavaAnnotations = "@Override"}
<<JavaElement>>+visitLineNumber(line : int, start : Label) : void{JavaAnnotations = "@Override"}
-t
Transformer
<<constructor>>+Transformer(config : Config)
<<JavaElement>>+transform(loader : ClassLoader, className : String, classBeingRedefined : Class<?>, protectionDomain : ProtectionDomain, classfileBuffer : byte"[]") : byte"[]"{JavaAnnotations = "@Override"}
+rejectByDefault(className : String) : boolean
AppMethodVisitor
~callsTarget : boolean
~line : int
<<constructor>>+AppMethodVisitor()
+visitMethodInsn(opcode : int, owner : String, name : String, desc : String) : void
+visitCode() : void
+visitLineNumber(line : int, start : Label) : void
+visitEnd() : void
-mv
Callee
~line : int
<<constructor>>+Callee(cName : String, mName : String, mDesc : String, src : String, ln : int)
-callees
0..*
App
+findCallingMethodsInJar(jarPath : String, targetClass : String, targetMethodDeclaration : String) : void
+main(args : String"[]") : void
-cv
AppClassVisitor
<<constructor>>+AppClassVisitor()
+visit(version : int, access : int, name : String, signature : String, superName : String, interfaces : String"[]") : void
+visitSource(source : String, debug : String) : void
+visitMethod(access : int, name : String, desc : String, signature : String, exceptions : String"[]") : MethodVisitor
ThreadProfiler
-debugEnabled : boolean = false
-methodCount : int = 0
+newMethod(methodName : String) : int
+recordMethodCall(caller : StackTraceElement, callee : StackTraceElement, calleeMethodName : String, ct : Thread) : void
+enterMethod(globalMethod : String) : void
+exitMethod(globalMethod : String) : void
+methodMatchesRule(className : String, methodName : String) : boolean
<<getter>>+isDebugEnabled() : boolean
<<setter>>+setDebugEnabled(debugEnabled : boolean) : void
+startRMIService() : void
<<getter>>#getTimeline() : ExecutionTimeline
Log
-verbosity : int = 0
-showThread : boolean = true
<<setter>>+setVerbosity(verbosity : int) : void
+print(verbosity : int, s : Object) : void
+print(verbosity : int, s : Object, t : Throwable) : void
-preffix() : String

Figure 13 UML Class Diagram for the Profiler Module

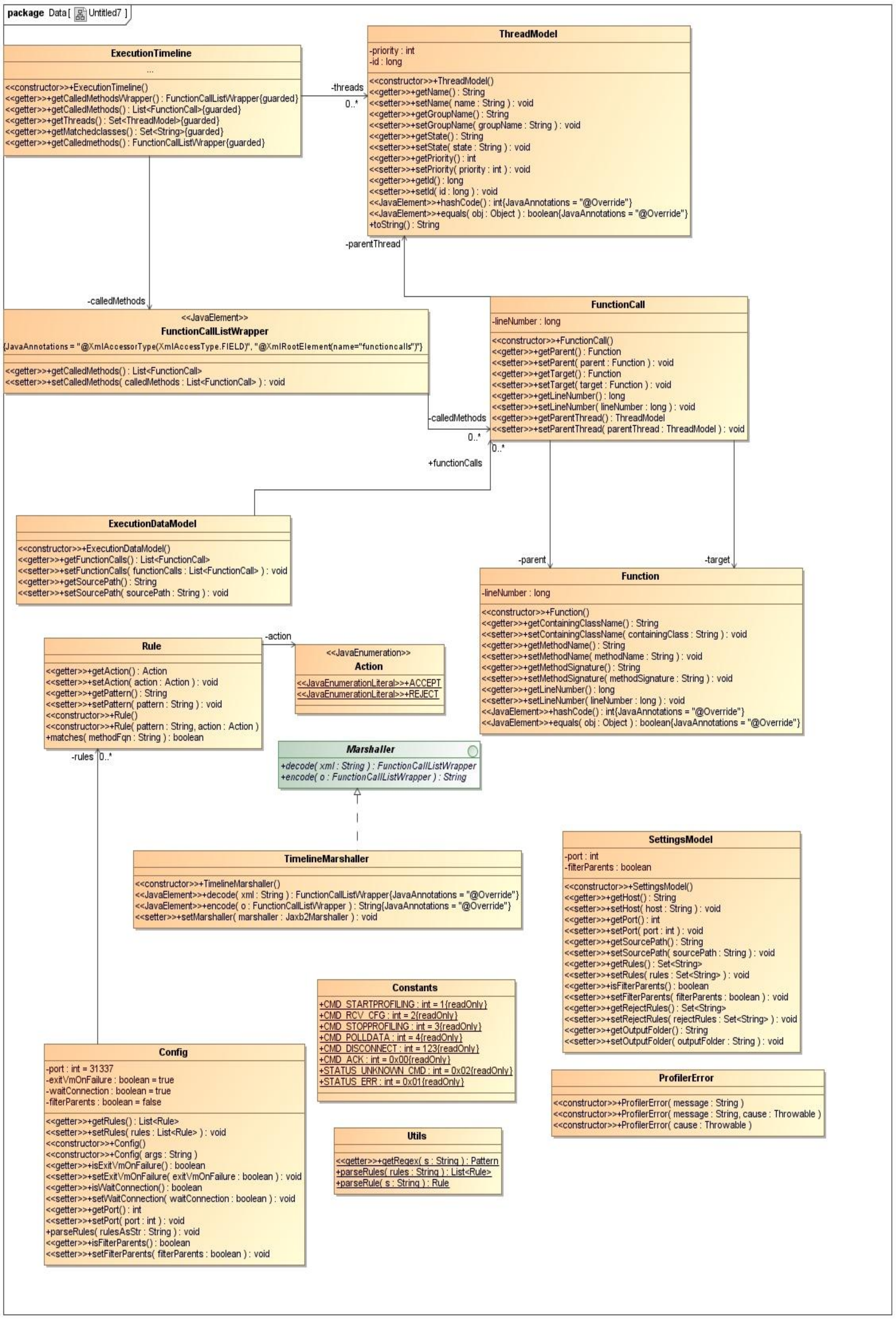

package Data[Untitled7]
ExecutionTimeline
...
<<constructor>>+ExecutionTimeline()
<<getter>>+getCalledMethodsWrapper() : FunctionCallListWrapper{guarded}
<<getter>>+getCalledMethods() : List<FunctionCall>{guarded}
<<getter>>+getThreads() : Set<ThreadModel>{guarded}
<<getter>>+getMatchedclasses() : Set<String>{guarded}
<<getter>>+getCalledmethods() : FunctionCallListWrapper{guarded}
-threads
0..*
ThreadModel
-priority : int
-id : long
<<constructor>>+ThreadModel()
<<getter>>+getName() : String
<<setter>>+setName(name : String) : void
<<getter>>+getGroupName() : String
<<setter>>+setGroupName(groupName : String) : void
<<getter>>+getState() : String
<<setter>>+setState(state : String) : void
<<getter>>+getPriority() : int
<<setter>>+setPriority(priority : int) : void
<<getter>>+getId() : long
<<setter>>+setId(id : long) : void
<<JavaElement>>+hashCode() : int{JavaAnnotations = "@Override"}
<<JavaElement>>+equals(obj : Object) : boolean{JavaAnnotations = "@Override"}
+toString() : String
-parentThread
-calledMethods
<<JavaElement>>
FunctionCallListWrapper
{JavaAnnotations = "@XmlAccessorType(XmlAccessType.FIELD)", "@XmlRootElement(name="functioncalls")"}
<<getter>>+getCalledMethods() : List<FunctionCall>
<<setter>>+setCalledMethods(calledMethods : List<FunctionCall>) : void
FunctionCall
-lineNumber : long
<<constructor>>+FunctionCall()
<<getter>>+getParent() : Function
<<setter>>+setParent(parent : Function) : void
<<getter>>+getTarget() : Function
<<setter>>+setTarget(target : Function) : void
<<getter>>+getLineNumber() : long
<<setter>>+setLineNumber(lineNumber : long) : void
<<getter>>+getParentThread() : ThreadModel
<<setter>>+setParentThread(parentThread : ThreadModel) : void
-calledMethods
0..*
0..*
+functionCalls
ExecutionDataModel
<<constructor>>+ExecutionDataModel()
<<getter>>+getFunctionCalls() : List<FunctionCall>
<<setter>>+setFunctionCalls(functionCalls : List<FunctionCall>) : void
<<getter>>+getSourcePath() : String
<<setter>>+setSourcePath(sourcePath : String) : void
-parent
-target
Function
-lineNumber : long
<<constructor>>+Function()
<<getter>>+getContainingClassName() : String
<<setter>>+setContainingClassName(containingClass : String) : void
<<getter>>+getMethodName() : String
<<setter>>+setMethodName(methodName : String) : void
<<getter>>+getMethodSignature() : String
<<setter>>+setMethodSignature(methodSignature : String) : void
<<getter>>+getLineNumber() : long
<<setter>>+setLineNumber(lineNumber : long) : void
<<JavaElement>>+hashCode() : int{JavaAnnotations = "@Override"}
<<JavaElement>>+equals(obj : Object) : boolean{JavaAnnotations = "@Override"}
Rule
<<getter>>+getAction() : Action
<<setter>>+setAction(action : Action) : void
<<getter>>+getPattern() : String
<<setter>>+setPattern(pattern : String) : void
<<constructor>>+Rule()
<<constructor>>+Rule(pattern : String, action : Action)
+matches(methodFqn : String) : boolean
-action
<<JavaEnumeration>>
Action
<<JavaEnumerationLiteral>>+ACCEPT
<<JavaEnumerationLiteral>>+REJECT
Marshaller
+decode(xml : String) : FunctionCallListWrapper
+encode(o : FunctionCallListWrapper) : String
-rules
0..*
TimelineMarshaller
<<constructor>>+TimelineMarshaller()
<<JavaElement>>+decode(xml : String) : FunctionCallListWrapper{JavaAnnotations = "@Override"}
<<JavaElement>>+encode(o : FunctionCallListWrapper) : String{JavaAnnotations = "@Override"}
<<setter>>+setMarshaller(marshaller : Jaxb2Marshaller) : void
SettingsModel
-port : int
-filterParents : boolean
<<constructor>>+SettingsModel()
<<getter>>+getHost() : String
<<setter>>+setHost(host : String) : void
<<getter>>+getPort() : int
<<setter>>+setPort(port : int) : void
<<getter>>+getSourcePath() : String
<<setter>>+setSourcePath(sourcePath : String) : void
<<getter>>+getRules() : Set<String>
<<setter>>+setRules(rules : Set<String>) : void
<<getter>>+isFilterParents() : boolean
<<setter>>+setFilterParents(filterParents : boolean) : void
<<getter>>+getRejectRules() : Set<String>
<<setter>>+setRejectRules(rejectRules : Set<String>) : void
<<getter>>+getOutputFolder() : String
<<setter>>+setOutputFolder(outputFolder : String) : void
Constants
+CMD_STARTPROFILING : int = 1{readOnly}
+CMD_RCV_CFG : int = 2{readOnly}
+CMD_STOPPROFILING : int = 3{readOnly}
+CMD_POLLDATA : int = 4{readOnly}
+CMD_DISCONNECT : int = 123{readOnly}
+CMD_ACK : int = 0x00{readOnly}
+STATUS_UNKNOWN_CMD : int = 0x02{readOnly}
+STATUS_ERR : int = 0x01{readOnly}
Config
-port : int = 31337
-exitVmOnFailure : boolean = true
-waitConnection : boolean = true
-filterParents : boolean = false
<<getter>>+getRules() : List<Rule>
<<setter>>+setRules(rules : List<Rule>) : void
<<constructor>>+Config()
<<constructor>>+Config(args : String)
<<getter>>+isExitVmOnFailure() : boolean
<<setter>>+setExitVmOnFailure(exitVmOnFailure : boolean) : void
<<getter>>+isWaitConnection() : boolean
<<setter>>+setWaitConnection(waitConnection : boolean) : void
<<getter>>+getPort() : int
<<setter>>+setPort(port : int) : void
+parseRules(rulesAsStr : String) : void
<<getter>>+isFilterParents() : boolean
<<setter>>+setFilterParents(filterParents : boolean) : void
ProfilerError
<<constructor>>+ProfilerError(message : String)
<<constructor>>+ProfilerError(message : String, cause : Throwable)
<<constructor>>+ProfilerError(cause : Throwable)
Utils
<<getter>>+getRegex(s : String) : Pattern
+parseRules(rules : String) : List<Rule>
+parseRule(s : String) : Rule

Figure 14 Class diagram for the Commons module

Each module follows MVC. The dataflow between the different modules is described below. The host application is launched with the profiling agent attached; this binds a TCP server to a port and suspends the execution of the host application. The console application is started, the rules are introduces, the source path of the host application is given. The console application connects to the profiling agent by TCP and issues commands. When profiling is started the console sends a command code to the agent, sends all the configuration data and the agent resumes the execution of the host program, enabling the BytecodeTransformer, all method calls are intercepted and if they match a rule BytecodeTransformer inserts a static call to the ThreadProfiler which is responsible for gathering the necessary data and recording the method call. When the first method is matched, the BytecodeTransformer also inserts a static call to the start the IPC (Inter Process Communication) service of the ThreadProfiler. This is used by the TCP server to communicate with the host application in case that they do not share the same static context (the host application spawns different processes). The client periodically retrieves the list of active threads and the list of matched classes from the host application using the server as a proxy. When the profiling is stopped, the server retrieves the ExecutionTimeline through IPC, marshalls all the data into XML and sends it to the client through the socket, it also disables the BytecodeTransformer, methods continue to be intercepted but they are not recorded. The client receives it and saves it into an xml file stored in the configured output folder. When profiling is started, the same process repeats, all previous data is cleared and the xml file is overwritten. When the user starts visualizing, the visualizer reads the settings file of the console application to retrieve the source-path of the host application and to retrieve the path to the execution-timeline.xml file. The visualizer marshalls the execution timeline and constructs the corresponding graph in memory, when nodes are expanded and functions or classes are selected, the source code is retrieved from the source path and it's displayed in the source viewer.

In the following we will present the implementation details, challenges faced and decisions made separately for each major component of the system. The major algorithms and the applied design patterns will also be presented, along with features, functionalities and communication protocols of the system.

4.2 Presentation of components

4.2.1 The profiling agent

The profiling agent uses the java.lang.instrument API to attach itself to the execution of the host program and to instrument its classes before they are loaded.
Two interfaces in the package are of special interest:

- *"**Instrumentation** provides a set of methods that allow custom class transformers to be registered with the JVM. It can also be used to inspect the set of classes loaded by the JVM and to measure the approximate storage space required for any specific object in the memory. Redefinition of already loaded classes is possible thru the method redefineClasses(...). The implementation of this interface is provided by the Java runtime itself.*

- ***ClassFileTransformer** is the interface that's implemented by classes that perform the actual instrumentation of bytecode. It specifies a single method, transform(...), that's called on each of the registered class transformers whenever a class is being loaded by the JVM. Implementations of this interface are provided by the users of the API* [2]*."*

In order to use the API the agent class must be implemented and registered with the JVM. The agent class must specify a static method with the following signature:

public static void premain(String agentArgs, Instrumentation inst);

The second parameter is the JVM-provided Instrumentation instance through which we can register our implementation of ClassFileTransformer called Transformer.
The compiled agent must be packaged into a jar archive and we have to specify the Premain-Class attribute in the MANIFEST.
To enable the agent, we must specify a special command line switch when invoking the java executable.

Example: java –javaagent:agent.jar –jar hostapplication.jar

Our Transformer class is registered with the Instrumentation instance and it's transform method checks if we should reject the class by default, if not it invokes the transform method of the BytecodeTransformer class.

We have to reject classes by default because some classes should be ignored by the profiler every single time, the list of these classes includes the current profiler, which should not instrument itself, dynamic proxies, system classes, classes from ObjectWeb ASM and other classes which cause JVM crashes like java.lang.reflect, java.lang.Thread, sun.security and so on. The BytecodeTransformer class is responsible for transforming classes and keeping track of the modified classes. This is where ObjectWeb ASM is used.
ObjectWeb ASM uses the visitor design pattern to add new operations to class elements.

The visitor design pattern provides a way of separating an algorithm from the object on which it operates, therefore enabling us to add new operations to existing objects without modifying their structure; instead we create visitor classes that implement the new operations. The visitor takes the instance reference as input and implements this goal through double dispatch.

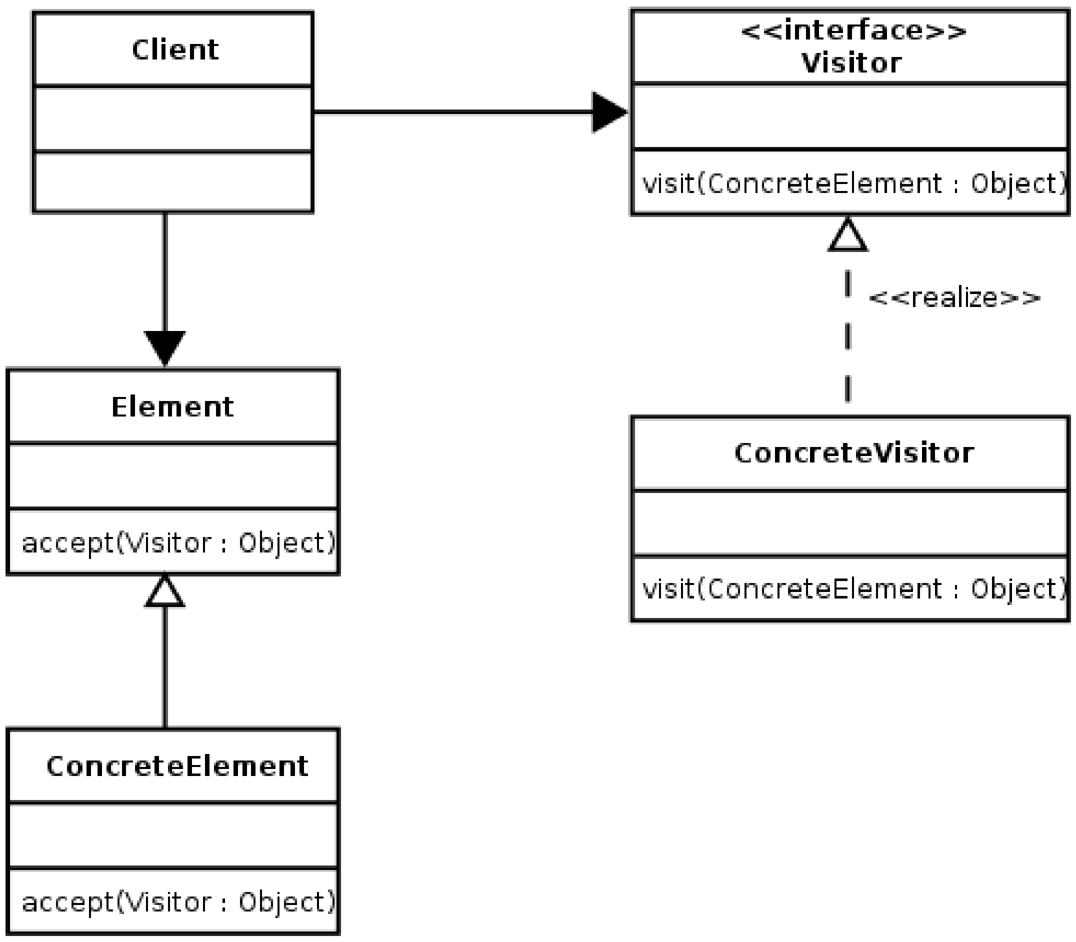


Figure 15 The Visitor Design Pattern in UML

The ClassReader and ClassWriter instances provides ways of reading and writing the bytecode of classes. We define a ClassAdapter subclass, which is basically a visitor on the class object, our subclass is called EnterExitClassAdapter and is responsible for modifying the methods of the class by around advice. It visits all the methods of the class [12], determines if we should profile the method, by matching their names to our regular expression rules, given by the client application. If the method is matched we apply our own MethodVisitor subclass, called EnterExitAdviceAdapter.

This is a subclass of org.objectweb.asm.commons.AdviceAdapter, which is a subclass of org.objectweb.asm.MethodAdapter, which allows adapting the bytecode of methods.

A MethodAdapter is used to insert before, after and around advices in methods and constructors. In AOP before advice is applied before the pointcut, after advice is applied after reaching the pointcut and around advice is applied both before and after the pointcut.

Our EnterExitAdviceAdapter simply overrides the onMethodEnter and onMethodExit functions of AdviceAdapter.

Our implementation is loosely based on **profiler4j** [9], a java profiler that allows the user to take snapshots of the execution. During our tests it did report inexact call graph in the scenario described below (in which we justify using RMI). One reason for this could be that it is based on static context and does not use IPC. It is a profiling tool and does not have any explicit goal of aiding code comprehension; call graphs are just a feature along measuring memory consumption and CPU. Profiler4j itself is based on JIP, The Java Interactive Profiler.

The **onMethodEnter** function is responsible for injecting bytecode into the beginning of method definitions, we also insert bytecode on method exit but this is not yet used to gather data.

```
@Override
protected void onMethodEnter() {
       if(!ipcStarted) {
              mv.visitMethodInsn(
              INVOKESTATIC, "org/balazsbela/symbion/profiler/ThreadProfiler",
              "startRMIService","()V");
              ipcStarted = true;
       }
       mv.visitLdcInsn(new String(methodName));
       mv.visitMethodInsn(
       INVOKESTATIC, "org/balazsbela/symbion/profiler/ThreadProfiler", "enterMethod",
                                 "(Ljava/lang/String;)V");
}
```

Our IPC is based on Spring RMI, since it's easy to use. When the first recorded method is reached, we start our RMI IPC service. Our IPCServices interface contains methods to start and stop profiling (so BytecodeTransformer.enabled is set to it's right value, regardless of shared static context) and to retrieve execution information, such as the called methods, the list of matched methods and the list of active threads.
In order to understand why we need RMI for Inter Process Communication and why we can't just rely on global static shared information between the host application and the TCP server try the following:

- Write a very simple java application, with a few dummy method calls
- Export it as an executable jar from Eclipse.
- Profile it with our application, without using RMI and defining a public static ExecutionTimeline shared between the host application and the server.
- Observe how ThreadProfiler, called from the host application reports the correct values while the Server doesn't.

Using RMI solves this problem and allows us to correctly record method calls even when the static context is not shared.

ThreadProfiler uses reflection to determine the calling function by accessing the current stacktrace and constructs Function objects which are added to the static ThreadProfiler.ExecutionTimeline. This ExecutionTimeline contains all the method calls, with parent, target and the line number on which the function call occurred (if available).
Also contains the list of classes that have been profiled (i.e. matched rules) and the list of the profiled threads.
When the profiling is stopped, BytecodeTransformer.enabled is set to false via RMI; methods are still intercepted, but not recorded. The timeline is retrieved via RMI, and marshaled using Jaxb2Marshaller. The FunctionCallListWrapper class is JAXB annotated, so we can fine-tune how the resulting XML looks like, to reduce its size.
The resulting XML which is sent and stored to the symbion-console has the following format:

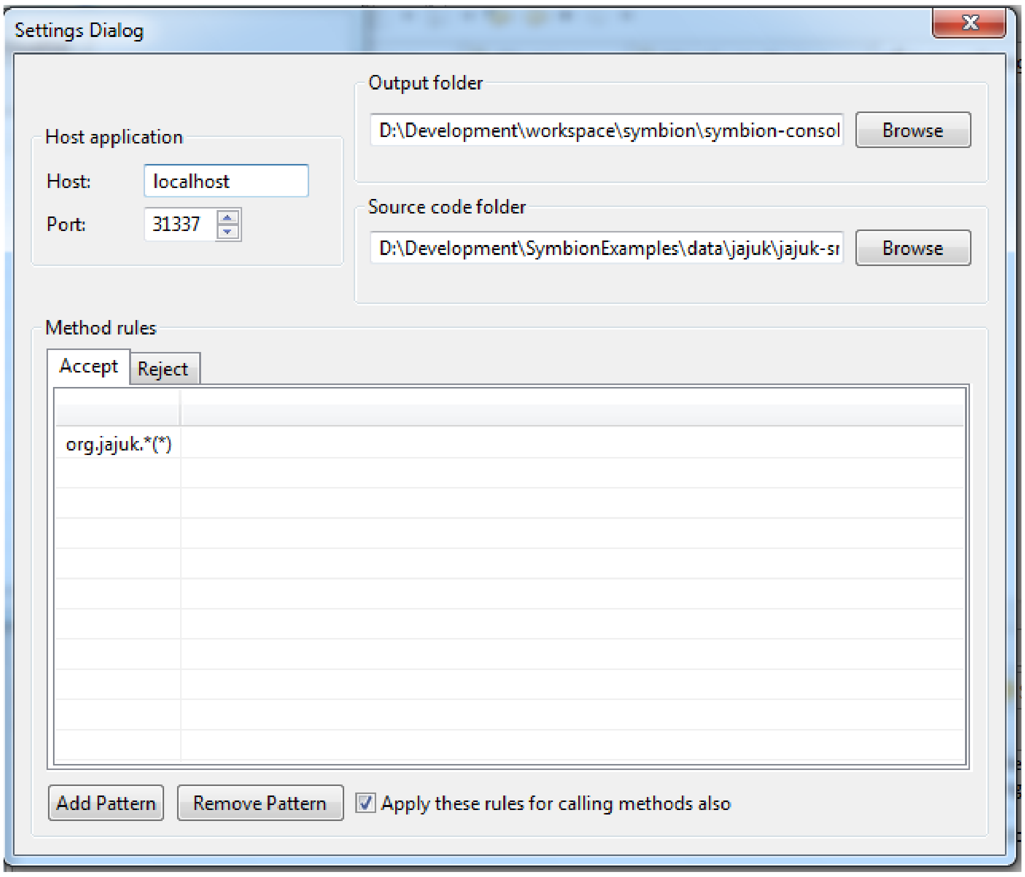
```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<functioncalls>
        <calledMethods>
                <lineNumber>-1</lineNumber>
                <parent>
                        <containingClassName>sun.reflect.NativeMethodAccessorImpl
                        </containingClassName>
                        <lineNumber>-2</lineNumber>
                        <methodName>invoke0</methodName>
                        <methodSignature></methodSignature>
                </parent>
                <parentThread>
                        <groupName>main</groupName>
                        <id>1</id>
                        <name>main</name>
                        <priority>5</priority>
                        <state>RUNNABLE</state>
                </parentThread>
                <target>
                        <containingClassName>org.balazsbela.test.Main</containingClassName>
                        <lineNumber>-1</lineNumber>
                        <methodName>main</methodName>
                        <methodSignature>(java.lang.String[])</methodSignature>
                </target>
        </calledMethods>
</functioncalls>
```

Periodically the symbion-console will send polling requests to the server, to retrieve the list of matched classes and threads, these are also marshaled and sent through the socket, but these are expected to be small so XMLEncoder/XMLDecoder is used. Proper synchronization is necessary.

4.2.2 The console application

The implementation of the profiling application is pretty straightforward. It has a frontend built in SWT using the Eclipse WindowBuilder plugin and a separate TCP Client thread that communicates with the frontend through callbacks, all this is governed by a MainController class. The separate thread is needed in order to not block the user interface when waiting for the response of the server. It also has **a settings dialog** which allows the user to edit the filtering rules, both for accepting and rejecting methods.

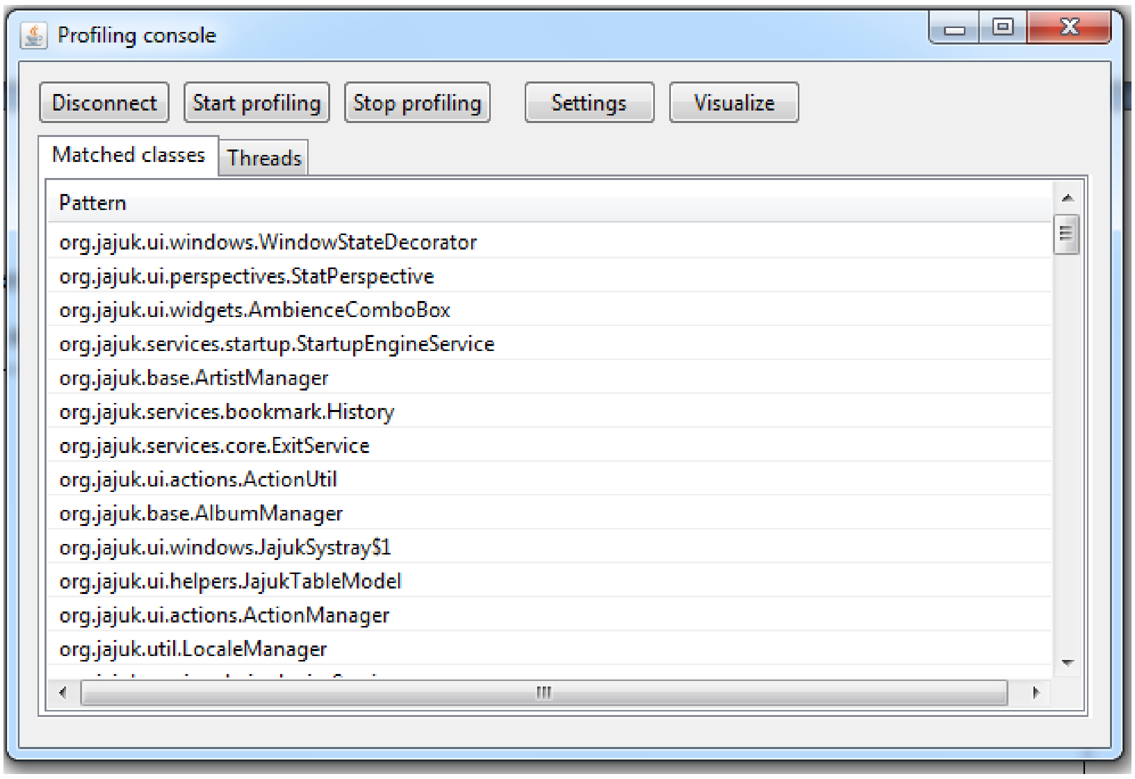
Methods that match rules in the accept tab but also match rules in the reject tab will be rejected, unmatched methods are rejected by default. The settings dialog allows the configuration of the output folder where the execution-timeline.xml file is saved and also allows the user to choose the path to the source code of the profiled application and the host and port where the server is running also must be given. This allows the analysis of applications running on remote hosts.



Host: localhost
Port: 31337
Output folder: D:\Development\workspace\symbion\symbion-consol
Source code folder: D:\Development\SymbionExamples\data\jajuk\jajuk-sr
org.jajuk.*(*)
[✓] Apply these rules for calling methods also

Figure 16 the settings dialog of the console

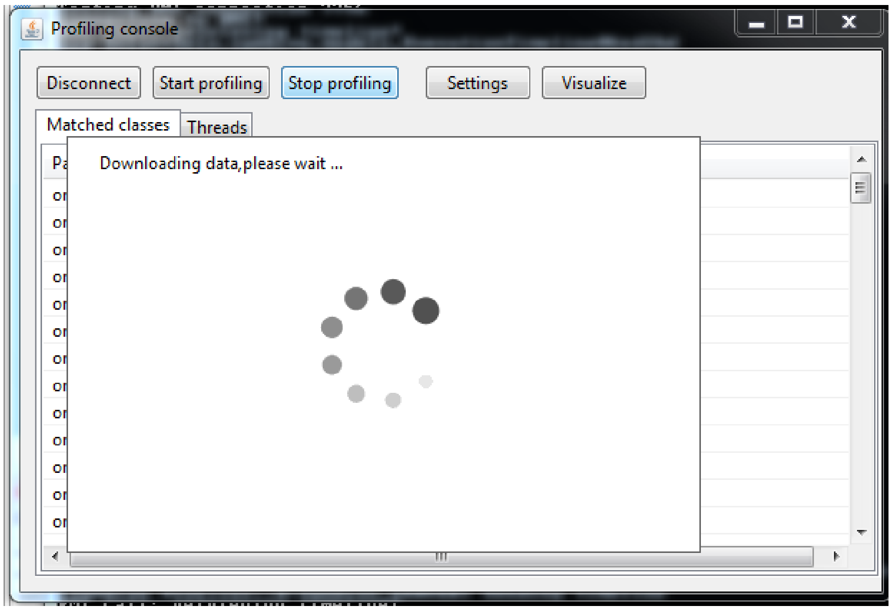
The client has a timer that polls the server every 10 seconds to retrieve the list of profiled classes and threads.



```
Pattern
org.jajuk.ui.windows.WindowStateDecorator
org.jajuk.ui.perspectives.StatPerspective
org.jajuk.ui.widgets.AmbienceComboBox
org.jajuk.services.startup.StartupEngineService
org.jajuk.base.ArtistManager
org.jajuk.services.bookmark.History
org.jajuk.services.core.ExitService
org.jajuk.ui.actions.ActionUtil
org.jajuk.base.AlbumManager
org.jajuk.ui.windows.JajukSystray$1
org.jajuk.ui.helpers.JajukTableModel
org.jajuk.ui.actions.ActionManager
org.jajuk.util.LocaleManager
```

Figure 17 the main window of the symbion-console

When the profiling is stopped and the resulting XML file is sent through the socket, an animated loading screen is displayed, since this XML file can be quite big (20-30 MB or more).



Downloading data,please wait ...

Figure 18 the animated loading screen

4.2.3 The visualizer

The visualizer application is started by the console on a separate thread and runs in it's context. It is triggered when the users clicks the Visualize button after stopping the profiling. The visualizer application also follows the Model-View-Controller principle. The central view allows the user to navigate within the 3-D environment using WASD controls.

When the application starts, a singleton MainController is instantiated. This MainController has two data sources, one MainRepository and a SourceProvider. The MainRepository is responsible for reading the settings.xml file created by the SettingsDialog in the console application in order to access the source path of the application and to gain access to the output folder where the execution-timeline.xml was saved. The MainRepository uses the same Jaxb2Marshaller (defined in the spring application context of symbion-commons) to unmarshall the function-call list. A graph is constructed from this list, where each node has the caller functions as parents and its callee functions as children. The roots are extracted and assigned as children to an artificial start node. The functions are stored in a hashtable for easy access, the full function name is the key in this hashtable and the FunctionModel (containing method name, class, parents and targets) is the value.

To get a call-tree out of this call-graph we duplicate each target node, this simplifies our model a lot and makes it a lot easier to position in a non-overlapping way and makes it easier to follow for the user since the children nodes are expanded in expected positions and there are a lot fewer arrows pointing to one node, this makes our visual model less chaotic.

When a node is **expanded**, it is retrieved from the hashtable and the list of target FunctionModels is traversed and a FunctionNode is created for each target. This FunctionNode contains the geometry object and the scene node; it also has an assigned label and owns its arrows too. This function node is attached to the expanding function's scene node. FunctionNodes are also stored in a hashtable for easy access. Their key has the following format: full parent function name + "->" + full function name. This makes them easy to identify and retrieve. The search functionality relies on this. Nodes are expanded when right clicked; ray querying is used to cast a ray from the current mouse position and the first intersecting node is expanded. Nodes change their color during expansion.

When a node is expanded a distance is calculated where its children will be positioned on a sphere, the radius of the sphere depends on the number of children, the more the larger radius. Each target node is assigned an expanding direction vector. This direction vector is generated randomly, for this we use an algorithm to generate random points on a sphere.

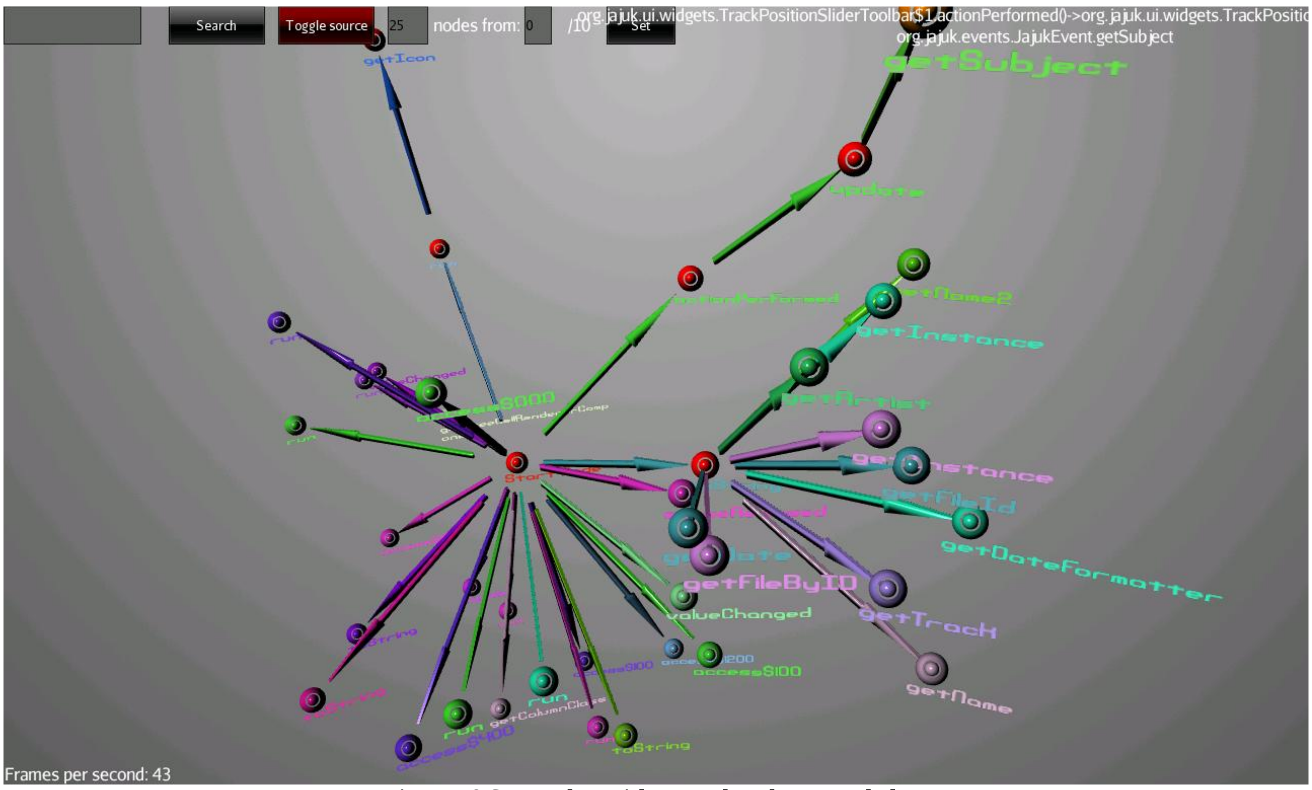
```
public Vector3f generateNewDirection(int nrNodes) {
       // http://www.cs.cmu.edu/~mws/rpos.html
       // Algorithm to generate random points on a sphere
       float distanceCoef = getDistance(nrNodes);

       float max = 2 * (int) distanceCoef;
       float min = -(int) distanceCoef;
       float z = FastMath.nextRandomFloat() * max + FastMath.nextRandomFloat() * min;
       float phi = FastMath.nextRandomFloat() * 2 * FastMath.PI;

       float theta = FastMath.asin(z / max);
       float x = max * FastMath.cos(theta) * FastMath.cos(phi);
       float y = max * FastMath.cos(theta) * FastMath.sin(phi);
       Vector3f direction = new Vector3f(x, y, z);
       direction.normalize();

       return direction;
}
```

One of the challenges we have to overcome is avoiding the case when nodes are expanded over or intersecting other nodes. For this we use ray-querying to cast a ray in the generated direction and see if something intersects within the radius of the sphere, if a collision is detected we generate a new random direction. We repeat this for a given number of iterations, until we can find a free direction, if the number of iterations is exceeded, we simply give up and expand the node anyway, but the number of iterations is high enough to avoid this for a reasonable number of nodes. We also favor directions that have a higher angle relative to the arrow pointing towards the expanding node, this makes the nodes expand outwards and makes the whole structure more linear and easier to follow. Again, if a direction that does not fit these constraints is not found within a reasonable number of iterations, we expand anyway.

When a good expanding direction is found we schedule them for animation. We store the AnimationStates of each node in the nodeAnimationBuffer, and we update the position of each node and arrow in simpleUpdate. Arrows are scaled according to the expansion sphere radius. When the expanding is done, the expanded node is colored red, this helps highlight the expansion path. When a red node is right clicked, it is collapsed and all its children are detached from its scene node.

Since a node can have a fairly large number of children, especially the start node, we implemented **windowing**. Windowing allows the user to specify a range for the nodes to expand. It’s the same concept as pagination on the web. As an example the user can choose to expand the next 25 nodes starting from the 42-th node out of 120. The total number of child nodes is shown, the maximum number of nodes that can be expanded at once is set to 100 (it is configurable from code) but the default is set to 25 since it’s a reasonable value for which the graph is still fairly navigable and the user is not encouraged to concentrate on more nodes at once. If the range changes, it will be applied for the next expansion.

All nodes and arrows are **color coded** according to the class the function belongs too. The arrows are assigned the same color as the function node they are pointing to, so we can see what classes the functions belong to even if the whole path is expanded (i.e. the nodes are red).



org.jajuk.ui.widgets.TrackPositionSliderToolbar$1.actionPerformed()->org.jajuk.ui.widgets.TrackPositic
org.jajuk.events.JajukEvent.getSubject

Figure 19 Screenshot with several paths expanded

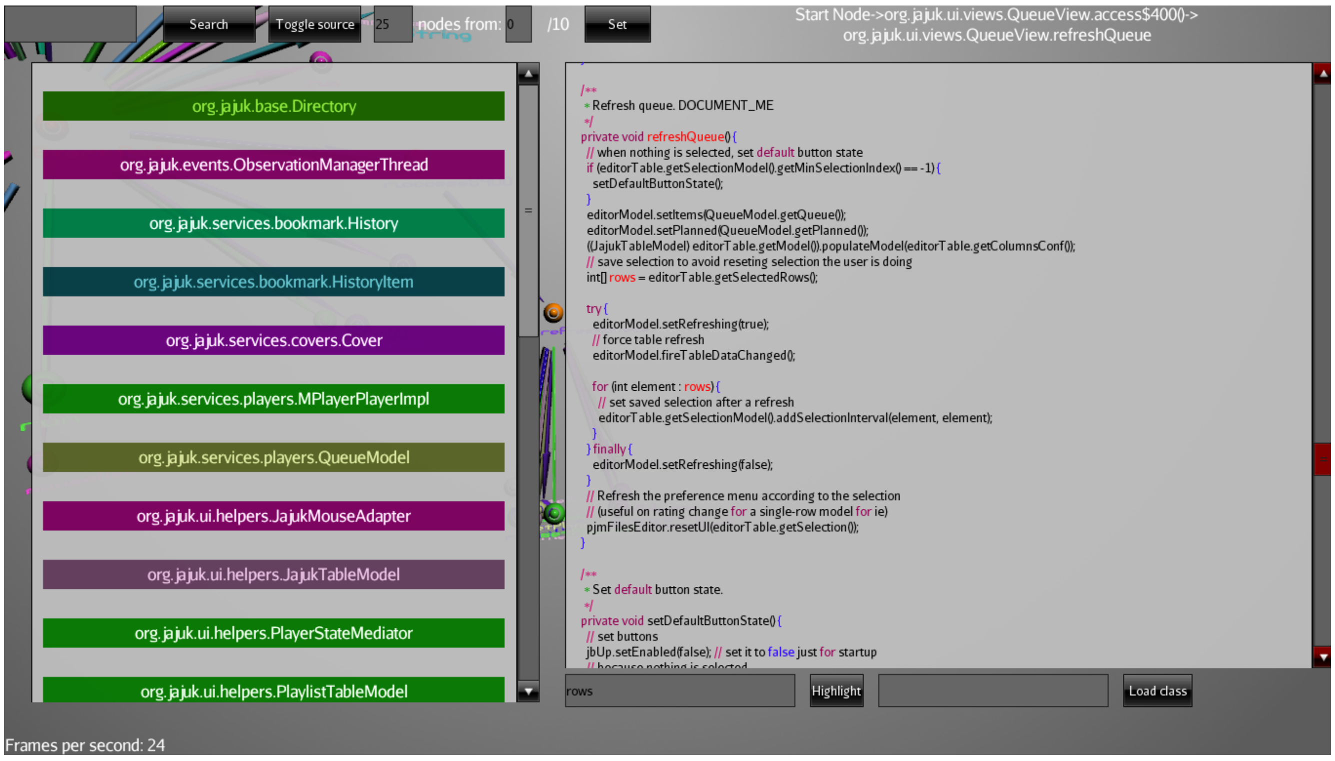
The **search** functionality allows function nodes to be searched, if a match is found the camera automatically is positioned on the node. If more than one node is matched by the query we can traverse through all the search results by repeatedly clicking search with the same query. This will cycle trough all the search results. The search function is designed to ease navigation in 3D space and to prevent the user from getting lost.

A node can be **selected** by left-clicking it, then the SourceProvider loads its source based on its class name on separate thread, it is added line-by-line to the **source code viewer** as it is loading. When the source code is loaded, javaparser is used to parse the class files, visits each method the same way ASM's ClassReader does and therefore reliably determines the line where the function is defined. The source-code viewer attempts to scroll to that line and highlights the name of the method throughout the source file.
Basic **syntax highlighting** is provided; tokens are highlighted to make the source more readable. One can also **highlight** custom words in the source code in order to find them easier, this is useful for finding field definitions used in functions.

We can also **load classes based on their name**, this is especially useful for abstract classes and interfaces that do not appear in the color coded legend, since they are not concrete classes. The **name of the currently selected function or class** is always shown in the upper right corner along with its parent function in case of functions.

The **source code browser** also features a **legend** containing the classes of all the expanded functions and their color coding. This list is clickable and ordered alphabetically, when clicked the source code of the corresponding class is loaded in the source code viewer. The color coding is based on hashing, therefore a unique color is constructed for each class name string and these are cached. In the source code browser, the class color is made a bit brighter and set as the color of the overlay text ensuring visibility.

The source code viewer and the source code browser are non-standard nifty controls implemented from scratch, each have an xml control definition and a controller class TextareaControl and LegendControl. The HudController contains all the method definitions of the HUD (Heads-Up Display, the overlay widgets). The layout of the 2D overlay is defined in ui.xml, therefore keeping presentation separate from logic.



Start Node->org.jajuk.ui.views.QueueView.access$400()->
org.jajuk.ui.views.QueueView.refreshQueue

- org.jajuk.base.Directory
- org.jajuk.events.ObservationManagerThread
- org.jajuk.services.bookmark.History
- org.jajuk.services.bookmark.HistoryItem
- org.jajuk.services.covers.Cover
- org.jajuk.services.players.MPlayerPlayerImpl
- org.jajuk.services.players.QueueModel
- org.jajuk.ui.helpers.JajukMouseAdapter
- org.jajuk.ui.helpers.JajukTableModel
- org.jajuk.ui.helpers.PlayerStateMediator
- org.jajuk.ui.helpers.PlaylistTableModel

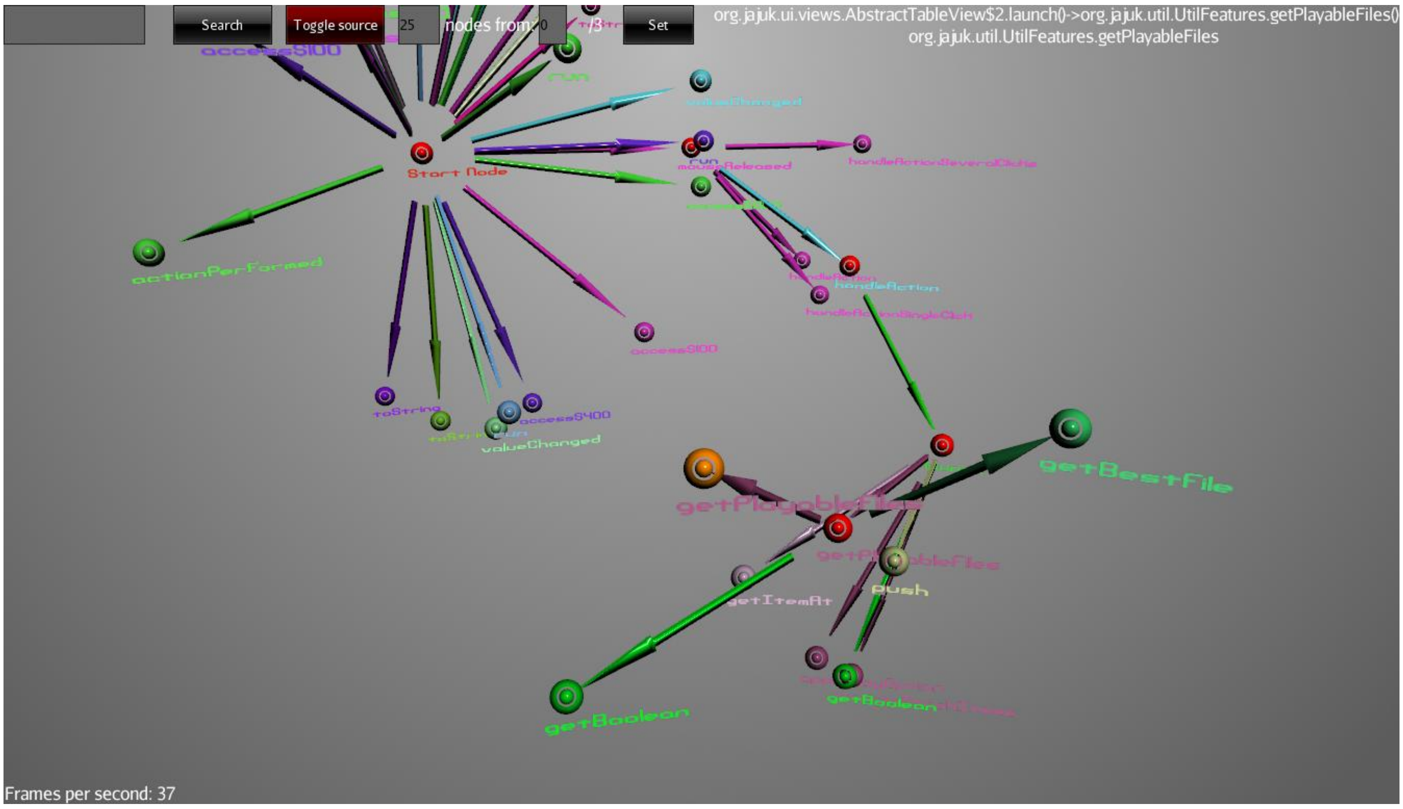
```
/**
 * Refresh queue. DOCUMENT_ME
 */
private void refreshQueue() {
  // when nothing is selected, set default button state
  if (editorTable.getSelectionModel().getMinSelectionIndex() == -1) {
    setDefaultButtonState();
  }
  editorModel.setItems(QueueModel.getQueue());
  editorModel.setPlanned(QueueModel.getPlanned());
  ((JajukTableModel) editorTable.getModel()).populateModel(editorTable.getColumnsConf());
  // save selection to avoid reseting selection the user is doing
  int[] rows = editorTable.getSelectedRows();

  try {
    editorModel.setRefreshing(true);
    // force table refresh
    editorModel.fireTableDataChanged();

    for (int element : rows) {
      // set saved selection after a refresh
      editorTable.getSelectionModel().addSelectionInterval(element, element);
    }
  } finally {
    editorModel.setRefreshing(false);
  }
  // Refresh the preference menu according to the selection
  // (useful on rating change for a single-row model for ie)
  pjmFilesEditor.resetUI(editorTable.getSelection());
}

/**
 * Set default button state.
 */
private void setDefaultButtonState() {
  // set buttons
  jbUp.setEnabled(false); // set it to false just for startup
  // because nothing is selected
```

rows

Figure 20 the source code viewer and color coded browser

The camera speed can be increased by holding shift while navigating; this allows faster access to elements in the scene.

The source code viewer and browser are activated when the toggle source button is pressed. The Set button applies the new range to the windowing feature. Windowing is applied to all expanded nodes, but the total list of child nodes is updated when a node is expanded.



org.jajuk.ui.views.AbstractTableView$2.launch()->org.jajuk.util.UtilFeatures.getPlayableFiles()
org.jajuk.util.UtilFeatures.getPlayableFiles

Frames per second: 37

Figure 21a path in the call tree, getPlayableFiles (orange) node is selected and its source code was loaded

5. Evaluation

The evaluation of our project can be done driven by two criteria:

- From a usability perspective, how effective is our approach, how much does it ease code comprehension, how do the implemented features help the user, is the goal of the project achieved.
- From a technical perspective, measuring performance and pushing the implementation to the limits, seeing how edge cases are handled, maintainability of codebase

5.1 Usability and effectiveness

Clearly, usability has a different meaning when we are building tools that target programmers. The Linux terminal maybe a usability disaster for average computer users but is definitely a very effective and powerful tool in the hands of the technically skilled.
The features that increase the usability of our program are:

- **Integration:** the visualizer is integrated into the console application and the gathered data is immediately available for visualization after it has been gathered, it only takes a click of a button.
- **The profiler has low impact on the execution of the host program**, the overhead of gathering data is barely noticeable, the host programs run at normal speed from a user perspective, of course this also depends on hardware factors but this is true on modern hardware.
- **The profiler runs constantly** after the host program has started it's execution, the recording of the execution (profiling) can be started and stopped at any point in the execution of the host program without it needing to be restarted, this allows gathering several datasets during a single execution, the data is immediately available for visualization so multiple analysis sessions can be held without interrupting the execution of the host program.
- **The search function** allows easy access to function nodes in 3D space.
- **Highlighting** eases searching in the source viewer.
- **Classes can be loaded on demand** without searching for them in the list of classes; this is especially useful for interfaces and abstract classes.
- **The class list legend** allows easy access to classes present in the 3D environment.

- **Windowing** allows the restriction of the number of nodes and makes the environment clearer and less cluttered.

These are the main features that have the target of making the program more usable, but this does not validate our initial idea. Since real-life usability tests have not been done and the application has not been deployed in real world environments we propose a case study to understand the power of our approach.

5.1.1 A case study: Jajuk

In order to evaluate its effectiveness we need to test our projects on a real codebase. For this purpose we have chosen an open-source music player application, Jajuk (Java Advanced Jukebox, http://jajuk.info). Jajuk is a relatively complex application; it has many features like collection support, retrieving covers, lyrics, Wikipedia information about the played tracks, Last.fm integration, statistics, album management and a lot more. Its source code is composed of around 50 packages, each with an average of around 10-15 classes. It may not seem like a large codebase, but the nature of the application requires many threads, services and custom ui widget implementations. It would definitely take a while to familiarize with.

We choose to explore this codebase, by checking it out of its repository and we have no prior knowledge about its architecture, technical design or developer documentation.
We give our exploration session a **goal**: to understand the process of how jajuk retrieves and loads the lyrics of the track we're currently listening to, this may seem trivial, but we would like a clear overview about what actually happens in the application during this functionality.
We start the profiling console, point it to the Jajuk source folder add one single simple acceptance rule: **org.jajuk.*(*)**, check to apply the rules to parent functions and start the application with the profiling agent attached. We start profiling, this will record the entire startup process which can be very complex and is not really relevant to our task, so after the application loaded we stop profiling, the data is sent but we choose not to visualize, instead we start profiling, choose a song and play it and change the view to the display tab, where the lyrics are displayed. We now stop profiling and visualize.

Upon expanding the start node, we notice that we have 66 root node functions.
We notice that there are many run functions; these are probably different threads so we explore each of them by searching and cycling through the search results while selecting them

to see if any of them is relevant, since loading the lyrics probably happens on a different thread. The first run function returned is from org.jajuk.ui.WikipediaView, it probably loads Wikipedia info, so we are probably searching for something like LyricsView, we search for lyrics and we find the <init> function of the LyricsView, part of the UI, but it’s not populated on creation and we have no other results so we expand the next 25 starting from 25.
We search for lyrics and we find the run function of the lyrics update thread org.jajuk.ui.View.LyricsView.$LyricsUpdateThread.run, this is relevant so we expand it. It calls getCurrentProvider, getSourceAddress, notify and getLyrics.
We expand getLyrics, which calls getProviders, setAudioFile and getLyrics multiple times, but these getLyrics nodes have different color, so they are defined in different classes. We select the parent getLyrics to view its source code and we see that it calls the getLyrics methods of different LyricsProviders, if we check the class of the expanded nodes we notices that these providers are: TagLyricsProvider, TxtLyricsProvider and LyricWikiWebLyricsProvider. The last one seems interesting so we expand its getLyrics method; it expands into a different getLyrics method which has different parameters, callProvider, cleanLyrics and other methods. We inspect the source code of the new getLyrics node and we see that it relies on callProvider to return the html of the lyrics and on cleanLyrics to extract the text. We expand callProvider and view its source code, we see that it calls DownloadManager.getTextFromCachedFile which expands into downloadToCache which we see makes the concrete http request if the file doesn’t exist, we expand it and we see that the http function NetworkUtils.getConnection is not called, so the file already existed.

Of course, nothing that we have learned this way is unique to our approach, we could have learned the same things while searching for lyrics in Eclipse and ctrl-clicking through classes and methods, but one may argue that data visualization does not replace data processing; it just presents the results in a way that is easier to understand. What matters is that we reduced our focus on functions that actually did execute during runtime and that through the visual representation we arrived to conclusions and we accomplished our task. The recorded execution provides direction for our exploration and even if we get sidetracked on functions that have no real connection to what we seek, they did get executed during our profiling session, so they provide context and lead to a greater overview of the behavior of the relevant components and how they interact. Armed with this overall view, we already know a lot about the system and we can use this knowledge in our next profiling-visualization session, since the goal is to get familiar with a new codebase.

One may also argue that the same thing could be achieved with recording an execution, linking it to an Eclipse plugin that displays a clickable 2D call graph and jumps to the relevant source code portions, opening files on click, using our same windowing functionality. This is true and such a plugin would help developers a lot, but 2D space is severely limiting, especially in eclipse where each plugin has very limited space and we want to view other plugins too, so this process would involve a lot of scrolling in order to get to the relevant parts of the 2D image and to explore it. 3D space gives us more freedom and we can navigate more data.

One such use-case does not validate our approach, but it does provide insight about what is possible and how powerful our system is.

5.2 Technical boundaries

The testing environment: Intel Core 2 duo laptop 2.0 GHz, 4 GB of RAM, Nvidia G102M 512MB, Windows 7 32 bit, JDK 7.

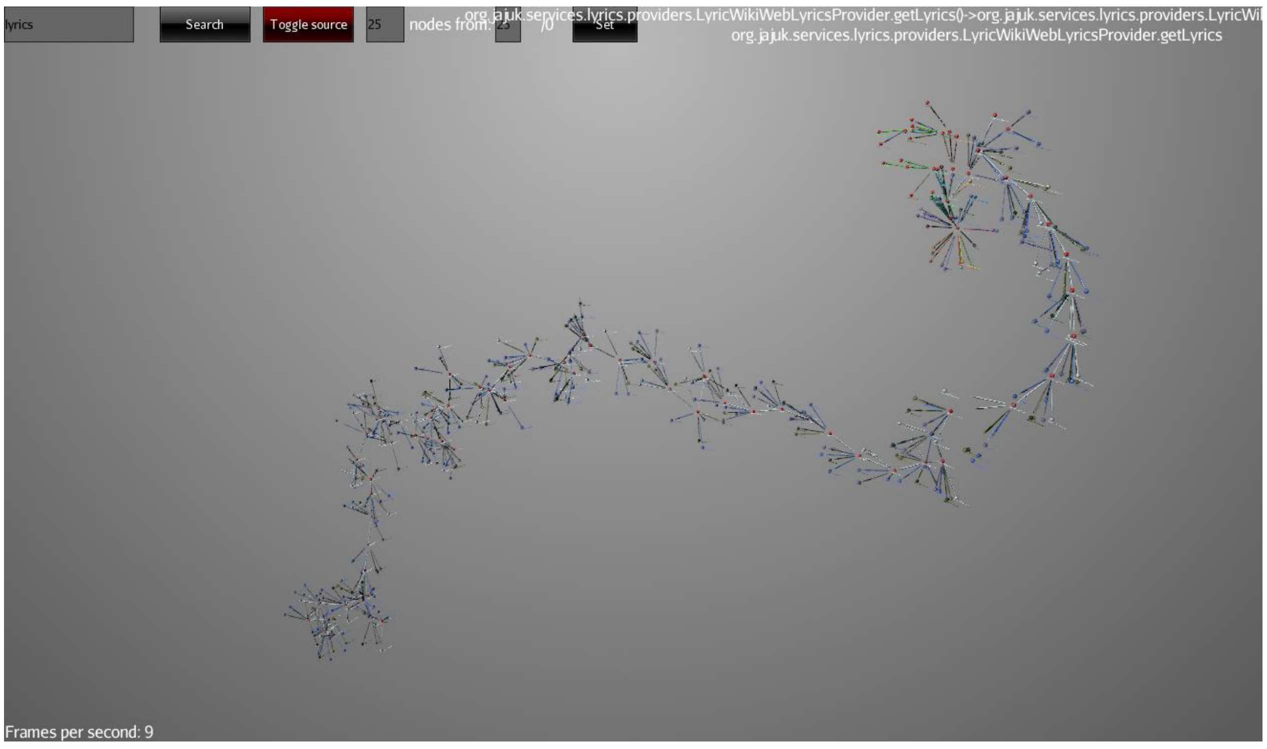

lyrics
Search
Toggle source
25
nodes from:
25
/0
Set
org.jajuk.services.lyrics.providers.LyricWikiWebLyricsProvider.getLyrics()->org.jajuk.services.lyrics.providers.LyricWi
org.jajuk.services.lyrics.providers.LyricWikiWebLyricsProvider.getLyrics
Frames per second: 9

Figure 22 stress test with many expanded nodes

During the testing the profiler worked well for 55440 function calls but the marshaller ran out of heap space for 66468 function calls, with the default setting of 1 GB of heap space, this can be increased by giving a JVM argument.

The visualizer managed to expand around 300 nodes at once, just for test purposes even more were tested but they clutter the scene and reduce the frame-rate. Expanding many nodes one-by-one was tested, the visualizer did not crash, but the frame-rate suffered.
The application works well for the expected amount of data, for which performance is acceptable. The application hangs occasionally when loading really large classes into the source-code viewer, even though this happens on a different thread.
No other crashes or performance problems were encountered.

Validation and Verification: Test applications were written to ensure the exactness of the reported call-graph and the correctness of the visual model.

6. Future plans

Below we describe possible improvements that need to be investigated in the future:

- Offer alternative ways of positioning nodes, maybe rearranging them in different formations.
- Offer a way to position a node to a custom spot, similar to Blender's translate function, when X, Y or Z is pressed we highlight the axis and allow the user to position a node along this axis using the mouse.
- Explore a way to select and highlight paths in a sub-tree.
- Explore grouping nodes based on threads
- Integration with IDE-s, especially Eclipse. Explore the possibility of building an eclipse plugin and integrating the visualizer.
- Explore gathering and visualizing state information and presenting it during the execution to see the influencing factors of the behavior of a component.
- Represent object instances and lifecycle.

Most of these items are not concrete and rely a lot on the feedback we obtain from the users of the current application; many of them need thorough investigation and analysis. Threads are visible in the current application too, but we could find better ways of focusing on them.
Gathering state information using reflection was explored, but we found that storing all the current values of all the fields in a class boosted the size of our resulting file significantly and was not explored further.

7. Conclusions

The ever increasing complexity of software systems makes it harder and harder for developers to understand unfamiliar codebases, the accommodation time to these code bases can be very long. Software visualization tools can help reduce this time and aid the developers in the process of understanding a system by analyzing execution cases. The Symbion project has the same aim. As a concrete example if a developer has to work on an instant messaging client, he could use this software to attach itself to the analyzed program, intercept its relevant function calls, perform a common operation in the host application like starting a file transfer with a friend, then visualize the gathered data in 3-D. By navigating the call-graph and viewing the source code simultaneously, he can have a better understanding of where the execution path is located within the code and he can access it instantly and modify whatever is needed. This provides a better overview upon the inner workings of the whole software and can save a lot of time that would be spent on textual search within random files.

The main focus points of this project are to minimize the footprint of the profiler as much as possible, so it doesn't disrupt the normal execution of the host program, to encourage the user to focus on a single functionality at a time, to provide easy and intuitive navigation and a clear overview over the events that occurred during the execution, to highlight the execution paths and pinpoint relevant locations in the source, leading to a more efficient way of gathering insight into the workings of the target program. Recording and playing back the execution provides direction for the developer in the process of exploring the source code, even if he gets sidetracked to functions that are not relevant to his task, he gains insight since those functions where indeed executed during the recording phase and either run in parallel with the functionality he focuses on or complement it, therefore providing context and a wider overview.

The Symbion project provides facilities to accomplish these goals, allowing the user to record multiple time intervals in a single execution and to immediately visualize them within a single analysis session. This provides an interesting alternative to the standard features of integrated development environments which accomplish this by textual search and by linking source code elements within each-other. This approach does not have the goal of being a viable replacement to these standard facilities, but to complement them and to provide a tool for developers that uses a different approach.

8. References

Each publication or source of information used for the research of this article is listed below.

The application and source code are also available at: https://sourceforge.net/p/symbion

[1] A. Fronk ,D. Gude,G. Rinkenauer, "Evaluating 3D-Visualisation of Code Structures in the Context of Reverse Engineering". [Online]. http://softeng.polito.it/events/WESRE2006/02Fronk.pdf

[2] Aarniala, Jari, (2005) Instrumenting Java bytecode. Department of Computer Science, University of Helsinki, Finland. [Online]. http://www.cs.helsinki.fi/u/pohjalai/k05/okk/seminar/Aarniala-instrumenting.pdf

[3] B. Wuensche, J. Grundy, J. Hosking B. Kot, "Information visualisation utilising 3D computer game engines case study: a source code comprehension tool," *CHINZ '05 Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: making CHI natural*, pp. 53 - 60, 2005. [Online]. http://www.cs.auckland.ac.nz/~john-g/papers/chinz2005.pdf

[4] Baecker, Ronald, "The Early History of Software Visualization," *University of Toronto*, 1998.

[5] Bruneton, Eric. (2007) ASM 4.0, A Java bytecode engineering library. [Online]. http://download.forge.objectweb.org/asm/asm4-guide.pdf

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

[7] Friendly, Michael, "A Brief History of Data Visualization," in *Handbook of Computational Statistics: Data Visualization*. 4700 Keele Street, Toronto, ON, Canada M3J 1P3: Psychology Department and Statistical Consulting Service.

[8] Friendly, Michael, (2009, August 24,) Gallery of Data Visualization The Best and Worst of Statistical Graphics. [Online]. http://euclid.psych.yorku.ca/SCS/Gallery/milestone/milestone.pdf

[9] Gomes, Antonio S. R. (2006, July) Profiler4j Site. [Online]. http://profiler4j.sourceforge.net/

[10] Harel, David, "Biting the Silver Bullet. Toward a Brighter Future for System Development.," *IEEE Computer*, vol. 25, no. 1, pp. 8-20, January 1992.

[11] J. Joshi, B. Cleary, C. Exton, (2004) Application of Helix Cone Tree Visualizations to Dynamic Call Graph Illustration. [Online]. http://www.dcs.warwick.ac.uk/pvw04/p10.pdf

[12] Kuleshov, Eugene Using the ASM framework to implement common Java bytecode transformation patterns. [Online]. http://asm.ow2.org/current/asm-transformations.pdf

[13] M. Lanza, C. Wysseier O. Greevy, "Visualizing Feature Interaction in 3-D," *SoftVis '06 Proceedings of the 2006 ACM symposium on Software visualization*, pp. 47-56, 2006. [Online]. http://www.inf.usi.ch/faculty/lanza/Downloads/Gree05d.pdf

[14] M. Czerwinski and G. Robertson R. DeLine. Microsoft Research, Microsoft Corporation, Redmond, USA. [Online]. http://research.microsoft.com/en-us/um/redmond/groups/hip/papers/vlhcc05-submitted.pdf

[15] Raymond, Eric S., *The Cathedral & the Bazaar*.: O'Reilly, 1999.

[16] Sosnoski, Dennis, (2005) IBM developerWorks. [Online]. http://www.ibm.com/developerworks/java/library/j-cwt05125/index.html

[17] Tsui, Andrew, Software Visualizations for Code Comprehension and Analysis. California Polytechnic State University. [Online]. https://blackboard.calpoly.edu/webapps/lobj-wiki-bb_bb60/wiki/CSC-0480-fkurfess/course/Home?cmd=GetImage&systemId=visualizations__0.pdf

[18] Wilcox, Andrew, (2006) IBM Developerworks. [Online]. http://www.ibm.com/developerworks/java/library/j-jip/index.html