Submitted for the Degree of B.Sc. in Computer Science, 2011

Summarising Execution Traces

Student Registration Number: 200706370

Student Name: Artur Jonkisz

Disclaimers

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

I agree to this material being made available in whole or in part to benefit the education of future students.

Signature: _____ Date: _____

Abstract

Complexity of software has been sharply increasing over last few decades. Understanding of medium and large scale systems proves to be an incredibly daunting task and many different techniques were developed over time to ease this problem. One of the most adopted techniques are diagrams, namely class diagrams, and recently with powerful graphical engines visual information representation gained much popularity.

Rich graphical information representations are to this day not very common, however there is a growing number of researchers trying to put visualisation into greater use for computer scientists. The project described in this report examines the feasibility of dynamically summarising execution traces and visually representing them at runtime.

Acknowledgements

I would like to thank Dr Marc Roper for supervising this project and providing his invaluable advice, support and guidance from the beginning.

I would also like thank Jamie Carmichael for reviewing drafts of this report and providing suggestions and all my all participants who sacrificed their time, took part in my evaluation and provided indispensable feedback.

Finally I would like to thank my fiancée for her continual support and patience throughout my time at university and especially while I was working on this report and the project.

Contents

Chapter 1

Introduction

Software engineering has always been a great engineering challenge and it does not look like that will change any time soon. It could actually be feared that the difficulty associated with developing and maintaining new and existing systems could become greater in the feature. There are various reasons for this.

The influence of the Internet on computing has been really apparent and with the new standards emerging, websites becoming more interactive, web applications, cloud services and even full operating systems designed to be only cloud enabled; tasks associated with the life cycle of these types of software get more complex. Now it is not enough to focus on resources available only locally. One needs to think about the inherent asynchronous behaviour of the Internet, local/remote resources and the way they interact with each other. So what exactly is happening at the client and server sides?

Today's systems are much larger and much more complex, and this is partially due to the exponential growth of memory and CPU speed, but this would not happen without the demand for it. And this demand will drive growth in size and complexity for a long time to come. Software systems are already really complex and teams to develop and maintain them are not made of a few people anymore. Let's for example look at Facebook where, as young as this company is, and as focused as it is, it actually employes a small army of people - over 2000 of them [12]. And this shows how some systems that might appear to be not so complex on the outside might indeed by incredibly complex on the inside. So how could one comprehend this?

The exponential complexity mentioned above is related to Moore's Law and there has been a debate over last few yeas, as to whether this law still applies to computing. The short answer would be "Yes, it does.". These days it might not be

possible to see a sharp increase in the performance simply caused by an increase in the clock rate of a processor. Parallelism and multiple processors on a single chip could increase the performance of a modern system. This drastic turn in hardware architecture demands some even more drastic changes in software architecture so it can receive the full benefit of it. These changes in software certainly make it more challenging to capture a bird's eye view of programs. So how can this view be obtained?

All of the above questions are not trivial, they are a cause for lots of frustration experienced by software engineers across the globe and cost businesses a substantial amount of money. There has been surprisingly little done to tackle these problems.

Stack traces are the entire history of a program and almost everything about the program can be extracted from them. As simple as it may sound, this extraction is incredibly problematic. Traces are constantly generated and their size quickly becomes an issue. Summarising this data statically could be quite tough, but doing so dynamically is where the real challenge begins. On the other hand the whole community of programmers can benefit from it and with visual aid could make the above problems much easier to deal with.

It is often said that "A picture is worth a thousand words.". One possible interpretation of this is that visual information can be processed much quicker, in larger quantities and more effectively by human beings when compared to text/words. Yet the use of this information in computer science is not widely adopted. The most adopted form of a graphical representation in computer science is still, fairly old, UML. UML is incredibly useful, but is it all that computer scientists can use to deal with their most common problems? The development in hardware and software brings some great opportunities for dynamic information visualisation.

Medium to large systems are becoming much more common and their complexity often cannot be lowered, whether it is due to the ultimate objective of the systems, lack of financial resources or other factors. The least that can be done to ease this problem is to provide some tool that would make the understanding of the systems more fluid and help locate needed functionality. Fortunately there is plenty of not realised potential of stack traces, computer resources and visual information representation, all of which need to be dealt with care. If the potential is utilised right it would hopefully result in creation of a tool that makes it much easier to answer some of the most common questions asked by software engineers.

Chapter 2

Related Work

The Number of available applications devoted to making programmers' tasks easier is remarkable and year after year there are new ones being created or more plugins being developed for existing ones. The vast majority of these applications follow some certain standards and conventions. Often an application is simply a combination of functionality present in two or three other applications. It does not happen very often that widely adopted tools stand out from the others. However, breaking out from this routine and creating a novel tool with a previously unseen interface can be immensely helpful to programmers.

Creating a tool that matches the above description and offers great support to software engineers is not an easy task. There are much fewer designs that this tool could be based upon. It often requires intensive research of areas that might not seem to be closely related.

Three separate research areas were identified and hopefully would help making the end product valuable to the programming community. All of the areas are described in this chapter and some include strengths and weaknesses of presented research.

2.1 Comprehension and Questions

Research about comprehension of complex programs and questions asked by programmers during software evaluation tasks date back to 90s and some of it is described in here. Because object oriented programming dates back another few decades most of the problems encountered back then would most likely still be problems today.

The size of the systems and their complexity keeps on growing, making maintenance and development tasks more and more difficult. A large number of commercial systems are so complex that it is not possible for them to be fully understood by an individual. So how could development and maintenance of such systems continue? As Einstein once stated, "It is not necessary to comprehend the world in which we live, it is enough to adapt ourselves to it." Similarly additional knowledge of the systems is not just "not necessary", but often is more time consuming and leads to more questions and exhaustive searches for answers.

Erdös and Sneed in their research back in 1998 [11] identified seven basic questions asked by the maintenance programmers. These are

1. Where is a particular subroutine or procedure invoked?
2. What are the arguments, results and predicates of a particular function?
3. How does the flow of control reach a particular location?
4. Where is a particular variable set, used or queried?
5. Where is a particular variable declared?
6. Where is a particular data object accessed, i.e. created, read, updated or deleted?
7. What are the inputs and outputs of a particular module?

Location of procedure invocation is the first task that must be completed by the maintenance programmer. Completing it is so crucial that without it making any progress is often impossible. For small systems it might be quite trivial to complete this task, but otherwise how is someone supposed to know where to insert a breakpoint or a log? It is easy to imagine how wandering from one class to another might be a waste of time; yet there still is no tool that can quickly and easily give the answer to this question. The stack trace on its own might not be sufficient to solve this problem, but combine it with visualisation and the solution can almost magically appear.

Once the procedure is located it is important to know what sort of arguments might be passed into it so that processing of the arguments can be analysed and the ultimate function of the procedure obtained. The first part of the answer to this question could be obtained from the stack trace, but there still is the problem of dynamically displaying the results to the users as it is unknown what information is important at different points of execution.

Flow of control and execution is really interesting. In theory there are infinitely many ways of getting to a specific location in a program. In practice though, it

is a much smaller number. Asking someone to remember and understand all the paths is unreasonable, but how about asking the same person to remember an image or a pattern representing these paths? If the person manages to remember the patterns and is presented with them, depending on the context, s/he might be able to identify the control flow very effectively.

Identifying where a particular variable is set, used or queried in code statically could be quite simple. Tools like Eclipse or VisualStudio are actually very good at it. However, doing so dynamically poses a much greater challenge. It is definitely possible, but the performance penalty introduced by it might not be acceptable.

Locating a variable declaration again can be simply and inexpensively done statically, where in a dynamic environment it might be impossible to be aware of a declaration of a particular variable until a moment of its creation.

Locating object accesses also can be done without great difficulty in both static and dynamic ways. However, there is an issue created by modern programming languages - garbage collection. Because garbage collection is carried out in some cases by virtual machines, and is indeterministic, it is often difficult to know when exactly an object is deleted.

Inputs and outputs of a particular module are not possible to obtain statically, and for dynamically performing this task debuggers are most often used.

Erdös and Sneed rightfully claim that maintaining a foreign program with little or no knowledge is possible without fully understanding it. Their further claims are that by simply answering the above questions, total productivity increases by 30% [19].

Murphy et al. [23] expanded upon Erdös' and Sneed's research and conducted two qualitative studies of programmers performing change tasks on medium to large scale programs. In these studies they identified 44 questions that they were able to categorise into four groups. These, presented with some short examples, are as follows:

1. Finding initial focus points - points in the code that are relevant to a task.
2. Building on those points - communication between points, where what is called, created, accessed or updated, and what data is used.
3. Understanding a subgraph - logic behind given relationships and objects, runtime behaviour, execution flow, and previous findings
4. Questions over groups of subgraphs - mappings between different groups,

where what functionality is taking place and possible effects of changes on the system as a whole.

Even though their study took place 8 years later it is still possible to see an correlation between the two. This is proof that the problems from the past are still present. Many of the questions related to the above groups could again be answered using stack traces or visualisation - or even better: the two together.

Implication and conclusions from Murphy et al.'s research are much more informative. They state that the participants in their study struggle to map their questions to tools and that the reason for this is that most of the tools are lacking high-level question support. It is common with IDEs that they are text based and visual information is barely ever used. Even the best IDEs like Eclipse, VisualStudio or Xcode target a very narrow set of questions and their support for high-level ones is rather poor. Their results indicate the need to develop tools with programmers' questions (and answering them) in mind.

The project described in this report will not attempt to answer a particular question but provide true support to programmers' needs to complete a task.

2.2 Stack Trace and Dynamic Tracing

Execution of any useful program cannot take place without the use of a stack. Effectively all the information about the execution can be gathered from its traces. And as easy as it may sound, it poses many challenges, which are not easy to overcome. Additionally they change from architecture to architecture and the objective of a task.

Arnold et al. [2] created a system that works by analysing traces and is effectively a parallel debugger. It is called a parallel debugger because it was developed specifically for large-scale computer systems with thousands of processors and tasks running simultaneously. Even though the focus of their research is not on tracing and understanding an individual process, lots of challenges that they had to overcome has to be addressed in this project. Challenges that they list are

1. Overwhelming channels of control.
2. Large data volumes.
3. Excessive data analysis overhead.
4. Scalable presentation of results.

$$U(r) = \frac{Fanin(r)}{N} \times \frac{Log(\frac{N}{Fanout(r)+1})}{Log(N)}$$

Figure 2.1: Formula used to calculate utilityhood metric of a routine

The first point probably sounds the most mysterious and the reason for this might be that it is closely related to parallel debuggers and the fact that a single front-end process controls interaction between back-end processes and spends an unacceptably long time managing connections to the processes.

In fact it does apply to monitoring a single process as well, but not to such a great extent. Agent connecting and process monitoring might be really slow and memory intensive. As it is described in Section 2.2.1 it is now possible to connect and monitor already running processes and it is important to not induce too much overhead.

Large data volumes were always associated with stack trace generation and analysis because, as stated above - every call made, and lots of CPU instructions, manipulate the stack. The more work is being done by a process the faster vast amounts of data are being generated.

Excessive data analysis overhead is an inherent problem caused by large data volumes. Whenever there is a large amount of data, and live analysis of it is being performed, the performance overhead might really be unacceptable to end-users. Excessive overhead is also unacceptable when creating a representation of the traces.

Scalable presentation of results is again an inherent problem created by points 2 and 3, so presentation has to be done efficiently and be informative. Presenting data in some standard forms, like text, is overwhelming to the users and prevents quick information processing.

Lethbridge and Hamou-Lhadj have tried to summarise the content of large traces [15] by taking an execution trace as input and returning a summary of its main content as output. The most important point and a great motivation for this research is the fact that programers almost always want to be able to look at a big picture and then request more detail on demand. They have approached this problem by creating a formula, listed in Figure 2.1, that is used to calculate a utilityhood metric for various routines. The algorithm used to clear utility routines from execution traces and summarise them consists of four steps:

1. Setting the parameters - exit condition, threshold value and methods to be manually removed i.e. Java ones.

2. Removing implementation details - algorithm removes methods like constructors/finalizers, accessing methods, etc.

3. Detecting utilities - calculating metrics, removing utilities and repeating the process until the exit condition is reached.

4. Further manipulation of the results - using additional tools to make further adjustments and convert the trace into a UML sequence diagram.

This process is quite simple but does require quite a significant amount of manual input from the user and assumes that users will manually adjust the algorithm's parameters if they wish to improve the generated results. Their own byte-code analyser and manipulator is used to inject code needed to generate the traces that are saved to a file and parsed after execution is finished. These downsides are making the presented algorithm cumbersome and difficult to use live.

2.2.1 DTrace

A real breakthrough in dynamic tracing was made by Sun Microsystems when after four years of intensive development they finally released, in 2005, a comprehensive dynamic tracing framework called DTrace. This framework is currently available on Solaris, Mac OS X, FreeBSD and NetBSD. It is worth pointing out that the most popular operating system, and thus most software, is unsupported; this system is obviously Windows. A limitation like this can be quite significant when one is considering creating software that rely on DTrace.

DTrace as described by Sun Microsystems, now Oracle, is a "Dynamic tracing framework for troubleshooting systemic problems in real time on production systems" [6]. One of the reasons why it has been developed is because performance analysis more and more often is being done not by the developers during the development process but by the system integrator in production. As peculiar it might sound it is happening mainly because of systems consistently getting larger and more complex.

DTrace offers dynamic instrumentation of both user-level and, quite surprisingly, kernel-level software. The use of it is absolutely safe. The features of it are far ranging: dynamic instrumentation, unified instrumentation, arbitrary-context kernel instrumentation, data integrity, arbitrary actions, predicates, a high-level control language, a scalable mechanism for aggregating data, speculative tracing, heterogenous instrumentation, scalable architecture and finally virtualised consumers. So how can it be used? Firstly it must be run on one of the supported

operating systems. Secondly a script for tracing must be written in a programming language called "D" and executed against the desired system or process.

One of the most appealing aspects of DTrace is its performance. The tasks specified in a D script are not executed by DTrace but delegated to instrumentation providers. It uses CPU architecture specific instructions to perform some of its magic. It has been successfully used for various reverse engineering tasks like: stack and heap overflows, code coverage and intrusion detection [3]. A detailed specification of DTrace framework is outside of the scope of this report. For more detailed information please refer to the Centrill et al. publication, "Dynamic Instrumentation of Production Systems" [6].

DTrace has been included in Java HotSpot 6 and provides the following probes to be used by subsystems: VM Lifecycle, Thread Lifecycle, Classloading, Garbage Collection, Method Compilation, Monitor and Application. All of these groups offer different functionality that can be used against the target system [21]. VM Agents are a way of using DTrace on Java systems prior to version 6.

Sun Microsystems added even greater support of DTrace to Java by providing BTrace. It has all benefits of DTrace, plus it is platform independent as it works by byte-code instrumentation using ObjectWeb ASM [1]. BTrace scripts, unlike DTrace, can be written entirely in Java; however, DTrace scripts are also supported.

2.3 Visualisation

In Section 2.2 a paper by Arnold et al. [2] was mentioned. Some of the challenges encountered in their project might apply to visualisation as well. Their project used visualisation and they have identified challenges related to it when dealing with traces. Telea et al. [24] in their publication presented challenges typically encountered when developing visual tools for software engineers. And finally quite a few hints were presented in Roberts and Zilles research [22]. There is some overlap in the challenges presented in all three publications. They are:

- Visualise everything.
- Easy access to high-level information.
- Low-level information on demand.
- Scalability is crucial.
- Colour should inform, not entertain.

- Visualisation should provide meaningful labels.
- Tool should integrate well with existing tools.
- It should be easy to deploy and customise.
- Must clearly add value or diminish waste.
- Must correctly identify stakeholder type: developers/testers/etc., managers or consultants.

Meeting all of the above individually obviously does not guarantee success, although mixing them together should greatly increase its probability. The presence of a top-down approach is almost non- existent in modern IDEs, yet it is often preferred by programmers when they are starting to work on a new system or even when working on a system with which they are already quite familiar. Including more information on demand seems really intuitive in this approach.

With the systems growing and getting more complex it is crucial to keep scalability in mind. With the amount of information an image can represent, not using colours would be a true waste; also labels should be used, but not be a main feature of a tool. All professionals evaluated by Telea et al. uniformly agreed that strong IDE integration is the most important tool-effectiveness factor.

Innovative architecture visualisation tools in nearly all cases are met with moderate to strong scepticism by professionals [24], thus it is important to make sure that the tool is customisable and easy to deploy, especially if it is not integrated within an IDE. Due to the aforementioned scepticism it is a must for a new tool to clearly add value, otherwise it is at high risk of being rejected.

2.3.1 Examples

Recently there has been a lot of interest around totally novel ideas for a user interface in IDEs. It is not about creating a new IDE but adding lots of functionality to an existing one. The first ones to achieve this were Bragdon et al. [4] with Code Bubbles presented in Figure 2.2.

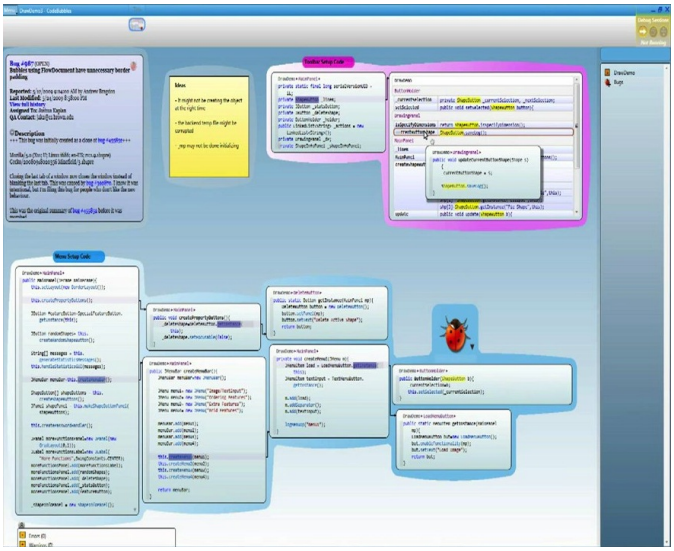


Figure 2.2: Code Bubbles.

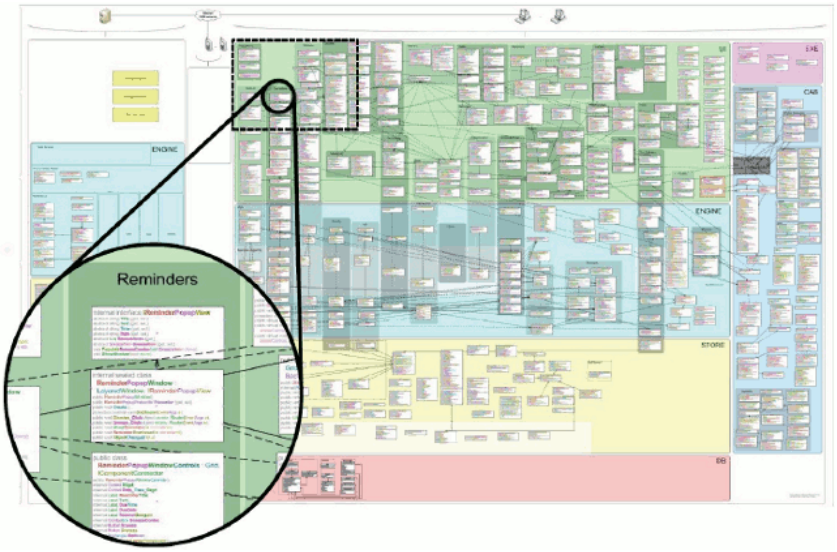


Figure 2.3: Code Maps.

They state that the programmers spend between 60-90% of their time on reading and navigating code, so to address this issue they came up with the metaphor of a Bubble, which is a fully editable fragment of code. When they performed their evaluation Code Bubbles were warmly welcomed by professionals and even though they are so drastically different to the paradigm programmers are used to, it did not take developers long to adopt and soon become more productive.

Soon after the research began on Code Bubbles and their integration into `Eclipse` another team arose with a slightly different idea. It is based on Code Bubbles and it is not difficult to see the similarities. Deline et al. [9] created Code Maps that are presented in Figure 2.3. They worked at Microsoft Corporation and integrated the tool into `VisualStudio`. The focus of this team shifted slightly from the idea of Bubbles and focused more on diagrams since software engineers are so familiar with them. In particular they focused onUML and this is where lots of the inspiration came from. The most remarkable feature of their plugin is dynamic zooming in and out that enables quick navigation within the project, is intuitive to use and as seen above enables users to easily distinguish between different system layers.

Both of the above examples are truly remarkable and positive feedback from evaluation participants is really encouraging, but both systems have very high system requirements and for most of the time not much is happening in there. This is an indication of the performance issues encountered by systems that are so graphically rich.

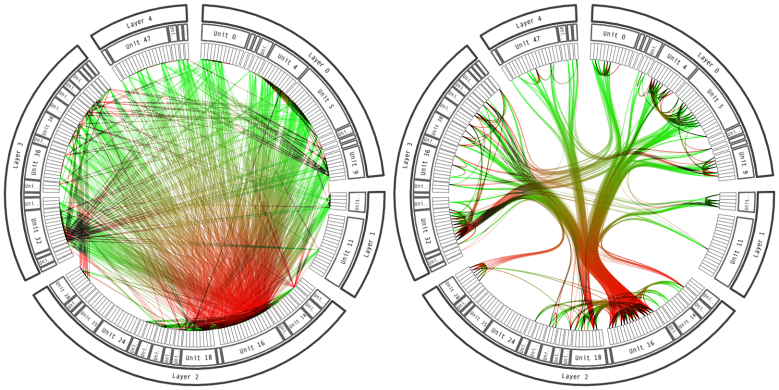


Figure 2.4: Edge Bundles. Image on the left is of unbundled nodes, where image on the right is of bundled nodes.

The last visualisation example at the first glance appears quite complex and not particularly intuitive to programmers; it is presented in Figure 2.4. This visualisation technique was developed by Danny Holten and was named Edge Bundles [17]. He has evaluated it on people closely related to software engineering to asses its potential in this specific field, and his results prove that almost immediately users find it easy to use. The tremendous advantage of this system is that it is scalable and massive amounts of information are readily available to the users, like green ends indicating callers and red callees. With this it is easy to see fan-in/out factor and localise potential utility classes.

Practical implementation of Edge Bundles was delivered by Cornelissen et al. [8] and was enriched by Massive Sequence diagrams and Trace Tiew. It is called EXTRAVIS and is shown in Figure 2.5. The use of Trace View adds a few more goals specified by Robers and Zilles [22]. These goals are interactiveness, searchability and the ability to detect patterns, phases and anomalies. There appears to be great potential for saving resources by rendering the Circular View only once and then simply redrawing the nodes. The only real issue with EXTRAVIS is that it cannot work dynamically but only on already generated traces created by logs. The great trouble with logs is that in order to know where to position them one must have a certain understanding of the system.

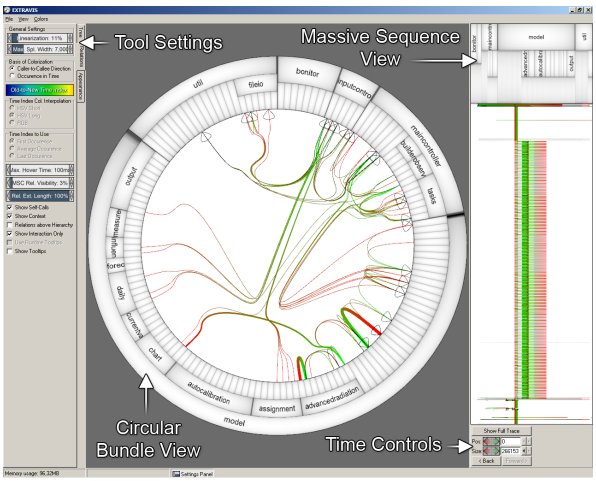


Figure 2.5: ExtraVis.

Chapter 3

Problem Description and Specification

3.1 Problem Description

Understanding a medium to large system is a difficult and challenging problem, yet it is one that is all too common for software developers. Whether they are developing an existing system, maintaining or building upon it, the understanding of the inner workings of the system is essential. There still is not a tool available that would allow the developers to gain a picture of the system's internal behaviour when it executes. This information would make it much easier to answer many, if not most, of the questions identified in the research. The most common questions asked by the software developers have barely changed over last decade and now with the help of the tools recently developed it should finally be possible to answer them.

Software engineers can be exposed to medium or large systems even while being at university and it is almost certain that, as their career progresses this exposure will be greater. For example a student of computer science secures an internship with a big corporation where he is asked to provide some additional functionality to an existing system. The functionality he needs to implement is not very difficult in itself, but the system is very complex indeed. It is made of 400 classes spread across 30 packages and uses frameworks and tools he was never exposed to before. If this was not enough, there were three people responsible for creating it and their programming style varies slightly. One of the members of the team sets up the working environment and starts up the system. It turns out it is a web application.

Initially everything the student can do to gain some understanding is to interact with the website and watch the results coming in from the localhost server. After

a while he understands what the application is doing, but not how is it doing it or more importantly where it is doing it. He considers using logging or print statements to try and identify hotspots, but then realises that he has no idea as to where he could put them. So he thinks to himself, if only there were a tool that could graphically show him what is happening inside the system when he presses a button and sees some results coming back, he would save lots of time and frustration.

The problem described above is one of many scenarios. Other ones could be where one changes job, changes team, is asked for expertise on an unknown system or is trying to understand the workings of a system being used in development. It is much more straightforward to build the overall understanding of a part of a system once the location of a subroutine of interest is known. Accompanying the subroutine location with a visual indication of the path taken to reach it brings further benefits. It provides information about another subroutine that invoked it, which builds up an understanding of dependencies in the system. Being able to see this information for all the subroutine calls should result in a partial comprehension of a control flow of the system. Once this comprehension is established and continually expanded upon it is much easier and quicker to find a way in the system.

Representation of a control flow alone sounds too beautiful and simplistic to be a full solution to the large number of questions asked by software engineers. It is often the case in maths, physics and other sciences, that when a solution to the particular problem is presented and it is not elegant and simple it is considered incorrect, or at least believed that there must be a better one. Why should this be any different in computer science, where instead of navigating through tens of thousands of lines of code a snapshot view of the whole or a part of a system could be sufficient. All of the data needed to present the snapshot view of the system is available from stack traces, however doing so during execution of the system poses some difficult problems.

Runtime extraction of a stack trace is difficult, processing it is resource intensive and live graphical representation of it is probably be the most expensive. Creating a new image representing a snapshot of a system for every method called would quickly degrade the performance of the machine and would result in the monitoring system becoming the top process. Execution of a single method can be as short as a few milliseconds or even nanoseconds and this is much less then the amount of time needed by a human brain to process a series of images.

3.2 Specification

The specification presented here deviates from the initial one. The initial specification can be viewed in Appendix D. A use case diagram of the updated specification is presented in Figure 3.1.

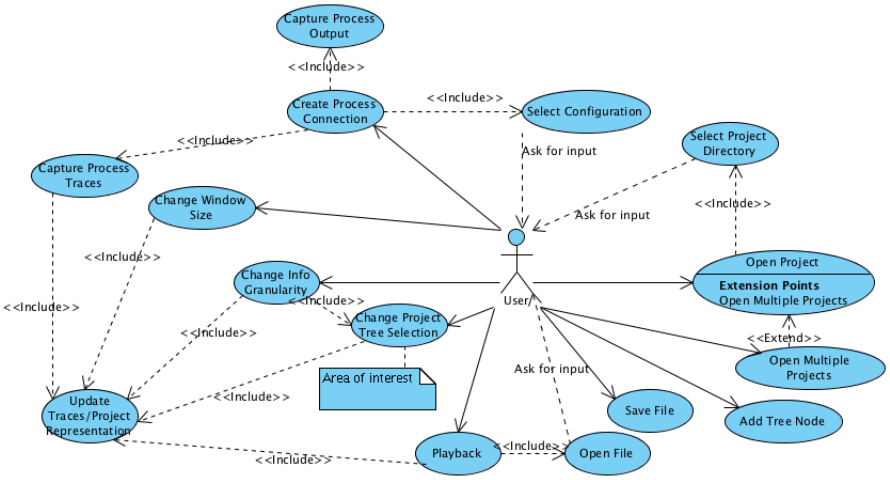


Figure 3.1: Use Case diagram of the most basic functionality.

The user interface must be novel, easy and intuitive to use. At the same time it should provide as much functionality as possible. Zoom in/out, search and filter functionalities should be supported and detecting patterns, phases and anomalies should be made as easy as possible. Stakeholders of the tool are developers/testers/etc. and the tool should be created to meet their needs and clearly add value to their existing toolkit.

3.2.1 Project Representation

The minimum a user should be able to do is select a project of interest by pointing the tool to the source code or binary directory. If neither is available the user should have the ability to create a project tree by manually creating tree nodes.

By default language specific build-in libraries should be excluded from the project tree as they could have great impact on the performance. The user should be able to add these libraries by either manually adding new tree nodes or by adding a whole tree by pointing to source code or binary directories.

The look and feel of the language's most adopted IDE should be mimicked in the project tree if IDE integration is not implemented. This should include icons and labels too, if legal.

The user should be allowed to change the areas of interest of the project through the project tree. It must be possible to do so statically, before the connection is established, but it is preferred to do so dynamically, when the connection is already established and the process is executing. Whenever the change occurs the representation should be updated to indicate to the user how the selected ares look when represented graphically.

3.2.2 Connection

Creation of a new process and establishing a connection to it is a minimum requirement. When a new process is created the user should be able to specify all, or as many as possible, input parameters. All of the output from the new process should be captured including print and exception statements. Some projects offer multiple entry/start points and the user should be able to choose which one he wants.

If possible a connection to an existing local process should be provided. The user must be presented with the list of available processes to choose from before establishing the connection. This list should exclude the process in which the tool is executed.

Connections to multiple processes simultaneously should be possible. This feature could be welcomed and greatly benefit users working on client/server architectures. The users must be able to disable individual connections.

All traces generated by connected processes must be captured and processed. Information such as time and thread data should be captured too.

3.2.3 Safety

It is critical that the extraction of traces from processes, whether they are new or already existing, must be safe. Under no circumstances should a monitored process be forced to terminate due to establishing a connection by the tool or otherwise.

An Established connection to a process must not allow the user to perform any malicious tasks like executing methods, changing variable names or values, hijacking processes or hacking it in any other way.

3.2.4 Representation of Traces

Traces representation must be a key element of the user interface and represent as much information as possible, but not too much as this could result in confusion.

Scalability: New information captured from connected processes must not result in the view becoming cluttered or difficult to read. Representation should automatically scale itself up or down when the size of the tool changes.

Granularity of captured information should be modifiable. The user should be able to specify at least component levels supported by a programming language, i.e. in Java these are: packages, classes and methods. This should be possible to change statically or preferably dynamically.

Representation must be as abstract, high-level as possible and low-level information should be available on demand. Colours should be utilised for representing additional information. Some degree of representation customisation should be implemented in order to meet users' personal preferences.

3.2.5 Playback

It must be possible to save recorded traces and open them. It must also be possible to play them back with functionality similar to that which can be found in typical media players such as play, pause, stop and change playback position.

The user should be able to change the speed of playback. Zoom in/out functionality should be implemented or at least an option to save a sub-trace. A medium to play a section of a trace would be a nice addition.

3.2.6 Performance

Performance in the tool is critical. Stack trace extraction and processing from the connected processes must be done efficiently and not cause performance degradation of the monitored system, monitoring system or the operating system.

The representation must be updated with new data from connected processes as often as possible, but not more often than can be perceived by humans. The process of this update must be efficient and only required areas of the representation must be updated.

The tool under no circumstances should use so many resources that the monitored processes are slowed down to the extent that they are unusable.

Chapter 4

System Design

This chapter focuses on high level aspects of the design of the system. The design and implementation of the system did not take place until all the research was done, thus the final design, methodology and implementation differ from ones proposed in the project specification and plan, which can be found in Appendix D.

The ultimate goal of the system is to provide dynamic trace visualisation, hence the name TraVis.

4.1 Target Language

It is not a goal of TraVis to provide dynamic tracing support and visualisation to multiple programming languages. It is about proving the concept of dynamic tracing and visualisation and examining whether it is possible and how helpful software engineers would find it. For these reasons the focus is made on a single programming language.

DTrace is a very powerful tool, but does come with some limitations. One of these limitations definitely is not language support, as it is compatible with over ten programming languages. The main restriction is operating system dependency. Currently it is supported on a wide range of UNIX operating systems, but the main player, Windows, is excluded.

Alongside DTrace support in Java 6 the BTrace tool was developed and it offers some great benefits, the vast majority of which are shared with DTrace. It is compatible with the "D" language but its scripts can also be written entirely in

Java. It is not platform dependent, which brings the benefit of targeting all the developers using the Windows platform.

Until DTrace support is expanded to the Windows platform it is reasonable to use BTrace, which suggests choosing Java as a target programming language. It does not enforce using Java as the main programming language for TraVis. BTrace simply is a tool that could be used inside any system and the dependency to it is minimal. The most TraVis needs to provide are the means to generate a BTrace script, compile it, attach it to a process and capture the output from BTrace. This is where the dependency ends, and if implemented with care it should be easy to convert it to use DTrace in the future.

Some of the further benefits of using Java are the vast amounts of open-source tools and frameworks developed by the community for example, ASM which is an all purpose Java bytecode manipulation and analysis framework, would greatly ease the task of reverse engineering Java projects.

4.2 Methodology

Agile software development, illustrated in Figure 4.1, was chosen as the main methodology for the development. Due to the specification of TraVis and the fact that the development must be done by an individual and not a team there are some deviations from the methodology.

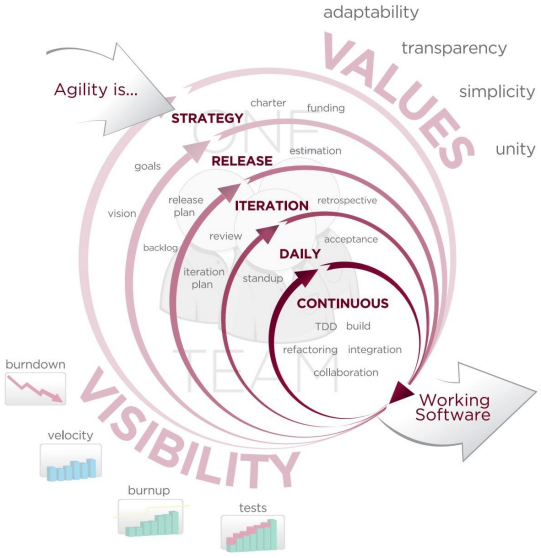


Figure 4.1: Agile Development - Accelerate Delivery poster.

Collaborative work, like Extreme Programming (XP), is a big part of agile development; it is adopted by most teams using this methodology, and arguably is what gives it some of its strength. Obviously in a team of one it is not possible to do any collaborative work.

Test Driven Development (TDD) ensures that the working software can be delivered on a regular, short interval basis without the worry that some of the implemented changes would cause the system to fail in another of its areas. It does require some time and practice to become a good TDD developer and, in some cases more than others, it might be more difficult to implement tests first.

TraVis definitely falls into the category where writing tests first is more challenging and there are a few reasons for this. The first one is that to a large extent it is dependent on external tools like BTrace and ASM. The second one is that it is continuously bombarded with large amounts of data generated by BTrace. The collected data processing is minimal to reduce the performance overhead. It is highly dependent on external files, like binary directories. And last but not least TraVis's main functionality comes from its Graphical User Interface (GUI) and GUIs are most often tested manually.

There are many areas in which the development of TraVis meets agile development methodology. All of the features of TraVis were developed in an iterative manner. In fact the iteration plan and goals were very clear from the start and features were actually built as modules, where all the effort was made to keep them simple, efficient, scalable and most importantly easy to adopt in other parts of the system. One of the great benefits of this approach is that coupling is minimised and refactoring can be done on small single modules rather than numbers of them. This approach also ensures high cohesion of the modules. Testing of the modules was done before they were integrated into the system.

4.3 Architecture

Combining good architectural decisions with agile methodology should result in a product that is easily manageable and scalable. Even if the finished product is considered as a medium scale system, adopting the above practices would make it easy to navigate after little practice.

The most important pattern that must drive the development of a graphically rich system, like TraVis, is Model-View-Controller (MVC). Diverging from this pattern is unacceptable under any circumstances, as in the long-term it would

result in limitations and complexities that would greatly diminish progress and productivity.

In TraVis the responsibilities of the model are: building, representing and handling the project tree; generating BTrace scripts and compiling them; establishing and managing process connections; capturing all the data from the monitored processes, processing it and passing it to controllers using the observer pattern; and finally handling trace playback. The controller handles communication between the model and the view; and between various view elements. The view is the only means of communicating with the model and is described in more detail in Section 4.4.

A reasonable effort was made to include design patterns like singleton, abstract factory, factory method, adapter, composite, decorator, observer, command, state, proxy, strategy and template method [14]. Implementation of these patterns in TraVis is discussed in more detail in Chapter 5.

4.4 User Interface

The user interface and its performance are probably the most important aspects of TraVis and certainly ones by which it will be judged by its users. So it was critical to make the right choices and decisions when it come to the GUI.

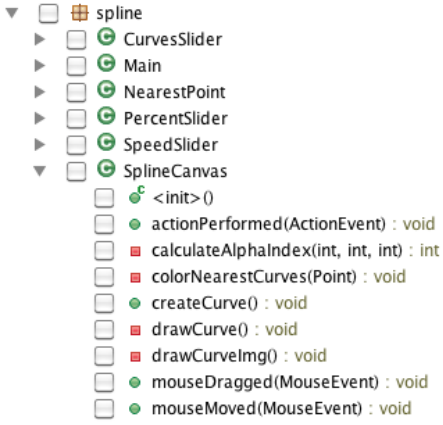


Figure 4.2: Java project tree representation in TraVis.

For the Java programming language, Eclipse is by far the most popular IDE. Due to the limited available development time the integration of TraVis into Eclipse will not take place. Another reason for not doing so is: the integration would require having source files of a project and having it properly set up in Eclipse. Fortunately Eclipse is an open-source project with Eclipse Public License, which means that it is alright to use their icons for Java project tree representation. It is possible to gain a lot of information from expanding project tree nodes in Eclipse and this is mimicked in TraVis. Even the colouring of return types next to method names is maintained. The project tree implementation can be seen in Figure 4.2. Checkboxes present at every tree node are the easiest possible way of selecting areas of interest in the project. Their great benefit is that they do not require from the users *any* prior knowledge of the project.

There was a number of user interfaces and techniques researched as potential

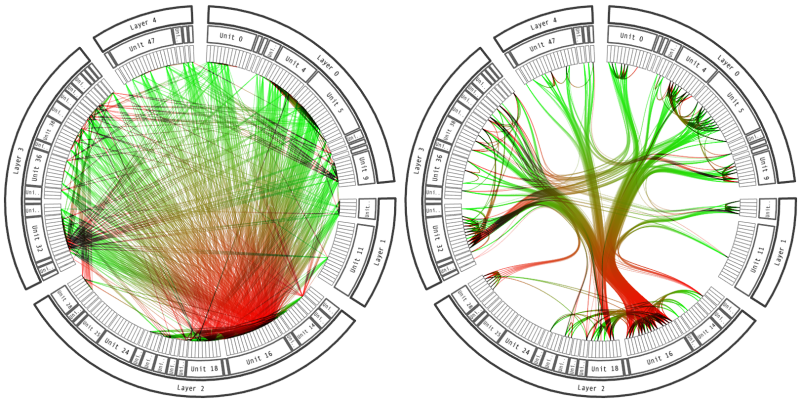


Figure 4.3: Edge Bundles as created by Danny Holten

candidates for use in TraVis. Their strengths and weaknesses were weighed. Edge bundles, presented in Figure 4.3, are the most novel and offer the greatest potential of them all.

UML inspired Code Bubbles and Code Maps do have phenomenal potential as tools used for code creation. However, dynamic manipulation of the views presented in these tools is too resource intensive for today's machines. Even if it was not so resource intensive, visualising interaction between various project elements at runtime, while keeping the view readable for everyone, is close to impossible because of the infinite number of ways that can represent one configuration.

Edge bundles are scalable and the circular graph representing a project is easy to create and guarantees to not change over time as new traces are recorded. The only way that can affect the circular graph is direct user intervention. Edges are the single elements that change over time. Bundling strength of the edges their number and presence are only some of the features that are customisable by the users. Their colour and information associated with each node are another excellent and inexpensive source of information that could be displayed on demand.

In the practical implementation of edge bundles by Holten et al. a massive sequence diagram, presented in Figure 4.4, is provided for easing the navigation in large amounts of data. This is a very powerful tool, which without providing lots of details that can be seen in a typical sequence diagram has got huge potential. It could be used by software engineers to spot the patterns and anomalies in an execution trace, and these often are a big part of the fundaments of the understanding of software.

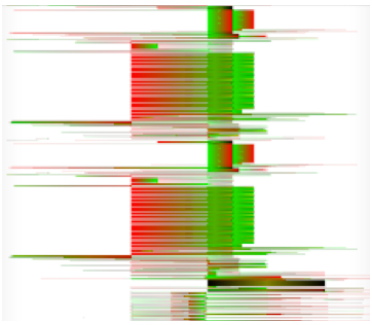


Figure 4.4: Fragment of a massive sequence diagram.

Unlike in tool created by Holten et al. this sort of navigation feature is only present in playback mode in TraVis as creating it dynamically is more challenging. It is not possible to determine at the execution time how wide the sequence diagram should be. Providing this sort of view would consume more resources and potentially put the tool at risk of becoming less usable. Using a sequence diagram is not the only technique that visualises patterns and anomalies; a simple call depth is another technique that can be used.

Recorded traces playback is another core feature of TraVis. This was never tried with edge bundles, but it is not difficult to imagine how well it would work with them. It might be that this feature in combination with all the others would save the developers the most time, frustration and add the most value.

4.5 Data Management

Data management is a big part of every system that is dealing with stack trace processing. TraVis is not any different in this respect. Actually, because it is handling data at runtime, its data management approach must be better and more efficient than those of similar systems.

For every received trace it is necessary to process it and store it in a form that provides maximum information at minimum overhead. Let's not forget that every received trace must be represented on the graph. It is possible to take the advantage of TraVis being used at runtime. Since there is so much happening at runtime in medium to large systems it seems reasonable to limit the amount of information the users can see at any one time. The users must retain the ability to configure some of this information.

Allowing the users to view too much information could in fact be a disadvantage, as the users would think that they would see and understand more by enabling more data. In reality it is quite the opposite. At runtime it is particularly important to carefully and effectively filter the data, hence the decision to limit the users' ability to show all the data. Another obvious reason for it is the performance.

When data is received by TraVis the necessary processing is done, it is saved to a file and passed over to the view so it can be reflected on the graph. The reflection does not happen for every received trace. The rate is capped at 25 Frames Per Second (FPS), as this frame rate is sufficient to provide a smooth animation, when perceived by humans. Any higher FPS is simply a negligence of resources. Actually television uses 25 FPS whereas movie uses 24 FPS [13].

Chapter 5

Detailed Design and Implementation

In this chapter the inner workings of TraVis are described in detail, highlighting the most challenging aspects of the implementation as well as their solutions. TraVis was developed with feature development in mind as currently there is not a single tool that provides the functionality that TraVis does. These plans for future extensions have implications, which can be experienced throughout the design.

5.1 Language and Libraries

5.1.1 Language

As described in Chapter 4 Section 4.1, Java has been chosen as the target programming language of the monitored application. It is not uncommon that programs written in one language are used to work with other languages. This applies to TraVis too. Any programming language, or mix of a few, could be used for configuring and running BTrace and creating and displaying the GUI.

TraVis is performance-critical and getting it right or wrong might mean the difference between the audience accepting it or not. The variety of programming languages today is absolutely overwhelming and choosing one over the other requires some knowledge and experience. Although, as stated above, it is not uncommon to mix programming languages when building larger applications, it is uncommon to mix them a lot within the main application; synchronisation and communication are two of the reasons why. The implications of choosing one language over another could be far-reaching.

With modern languages it is common that the software engineers put them into two groups. The first one is a group of languages that are able to run natively on hardware after compilation and linking. The most popular ones being able to do so are C, C++ and Objective-C. On the other hand, there are languages that are compiled into byte code, then linked at runtime and use just-in-time compilation (JIT) these include Java, C# and the likes.

The main advantage of the native languages is the speed and far greater support for native libraries like OpenGL. On the other hand, younger languages like Java or C# are much more forgiving, and easier to work with.

C# does not offer much more than Java. Both of these languages have their supporters and haters. The best available IDE for C# is Windows only VisualStudio. C# also has limited cross-platform support. Java on the other hand is platform independent and its performance is often better than that of C#. The author is also little bit more comfortable with Java than C#. These are considered to be good enough reasons to disqualify C#.

A very popular language for GUI development is Python, and its suitability for this project has been considered. It is a very flexible language, but its support for high level graphical elements is very limited and its performance is often criticised by software engineers.

A performance comparison was done between Java and Objective-C. It should be highlighted that at the time of the comparison the author believed that he would be able to use built-in curves for populating the edge bundles graph. Since the main goal of TraVis is to record traces and draw them on graph it is clear that drawing curves would be the most resource intensive task, hence the comparison focused on this area.

When drawing plain Bezier curves with solid colour and two control points the performance of Objective-C was much better, CPU usage was smaller, animation smoother and the curves looked smoother too. However there was a problem encountered in Objective-C when trying to draw the curves with gradient colour. There is a common misconception where people think that the curve is simply its stroke. Figure 5.1 shows an example of a Bezier curve with red stroke and blue fill. Objective-C does not support a gradient on a stroke and in order to achieve similar effect NSBezierPath had to be converted to its lower-level equivalent CGPathRef then a square of the entire curve area had to be

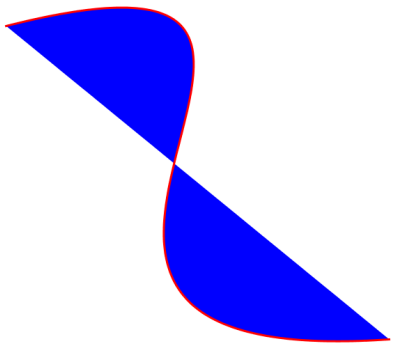


Figure 5.1: Bezier curve with red stroke and blue fill.

created with gradient and clipped to CGPathRef. This is an extremely inelegant and inefficient solution that caused a huge performance degradation and put Java in the lead.

Not knowing at the time of the comparison that curves for the graph would have to be manually implemented, considering small performance differences, rich data structures and community support, Java was chosen as the primary programming language.

5.1.2 Libraries

There are three major libraries and two external classes used in TraVis. The license types of all of the elements used is stated to confirm legality of their use.

Libraries:

1. BTrace mentioned in previous chapters is used for connecting to running and new processes and capturing traces from them. BTrace is available under the GPL-2.0 License.

2. ASM mentioned in Chapter 4 Section 4.1 is used for parsing Java bytecode and creating the project tree. ASM is available under the BSD License.

3. MiGLayout is a Java layout manager that is being used throughout TraVis instead of Java build-in layout managers, which are much less flexible. MiGLayout is available under both the BSD and GPL Licenses.

Classes:

1. `TextStroke` is a class used to create curved text that is not natively supported by Java. `TextStroke` is available under the Apache 2.0 License.

2. `CheckBoxTreeCellRenderer` is a class used in representing Java project tree that enables selecting nodes using checkboxes. It is available under the GNU Lesser General Public License.

3. Partial use of `Bspline` from `http://www.ibiblio.org/e-notes/Splines/Intro.htm`. It is used for the graph nodes creation. All sources on this website were made available for free by Evgeny Demidov.

5.2 Maintainability

As mentioned in Chapter 4 in Section 4.2 it is aimed to further develop TraVis in the future. This aim had a significant influence on the development process.

5.2.1 Self-documenting code

It has been anticipated that after all the functionality is implemented the project will be of a substantial size. To address this, an agile methodology was used with an iterative approach adapted, where only one project module was developed at a time. For most of the time modules were developed outside of the main project to maximise cohesion and minimise coupling. This often leads to small changes to the code in the integration stage.

The small changes in the code often result in documentation not being updated, thus it is often very confusing in future development. Another problem with plentiful in-code documentation in an early, experimental stage of the project is that some design decisions and implementations are changed more than others. This, again, often leads to documentation that is not up to date. As a matter of fact, many companies forbid large amounts of documentation and encourage self-documenting code throughout. It is not only about the confusion. Documentation that is not up to date can and often will result in bugs in the program.

The obvious benefit of well-written, formatted and logically structured code is that it is always up do date and easy to understand. Using meaningful variable and method names often results in long names, but this is a price worth paying, especially when almost all modern IDEs offer autocompletion. Another way of making the code easy to read and understand is to keep methods' bodies as short as possible.

TraVis follows all of the above rules: meaningful names of methods and variables are used, code is well-structured and formatted throughout, and methods' bodies are always kept to minimum. In fact, in TraVis there is a mean of 5.77 and standard deviation of 8.27 lines of code per method. Although it might sound high to some, please bear in mind that graphically rich applications always exhibit the problem of having long method bodies due to the way drawings are done and user interactions are handled in today's programming languages.

The total number of lines of code in TraVis is *10,156*. Taking into account the short time available for the development after research was performed, it would be incredibly difficult, error-prone and time-consuming to properly maintain the

documentation. It is worth indicating that the author is not trying to say that a large number of lines of code means a good, sophisticated solution; actually it often is the opposite. It is important for medium to large systems to clearly define the control flow, this can be done in code too, and even when self-documenting code is used, comments should be used for resolving ambiguity.

It is incredibly important for a project to have full documentation once it is released to the general public, since at this stage it is clearly defined what is being done by which methods. At the earliest development stages, where lots of refactoring is done and design decisions are often changed, it is not always possible to determine the final outcome of a method or whether it will actually make it to the final release.

5.2.2 Scalability

The term scalability applies to two aspects of this project and both of them are described below.

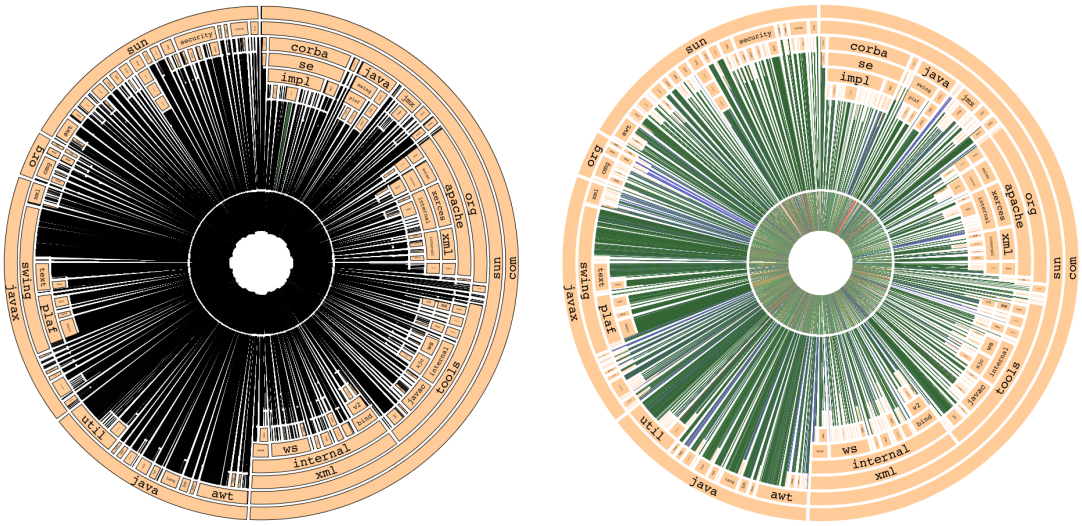
The first one is the scalability of the software and how this was achieved. As described above TraVis was developed in an iterative/modular fashion. Having different parts of the program as decoupled from other parts of the program as possible makes it incredibly easy to scale it up. It is simply achieved by the ability to create a whole system by connecting various modules with the fewest links possible. Scalability of TraVis could be improved by implementing more abstractions.

Very often software developers find it incredibly difficult to implement multilingual support. The reason is so simple it is actually embarrassing. Every major programming language supports internationalisation in one way or the other, and all it usually involves is instead of having hard coded names, notifications' text, etc., storing them in a separate file as a map. When the need arises to add a language, a copy of the file is made with a different name and all records are updated to the desired language. TraVis is internationalisation-ready and *all* textual content is accessed via a very simple method (comment added for descriptive purposes):

```
1 public static String get(String key) {
2         //messages = ResourceBundle.getBundle(
3         //         "travis.resources.messages", Locale.getDefault());
4         return messages.getString(key);
5 }
```

The second aspect is the scalability of the graphical elements of TraVis. Scalability of custom graphical elements and images in software can be achieved by not using any hard coded values for any user interface elements. As trivial as this might sound, the implementation of it is becoming much more challenging because of not being able to determine statically or predict the future dimensions of the elements on the screen.

Scalability of graphical elements applies to the previously discussed graphs as well. An attempt to reverse engineer a class diagram for the entire Java library failed after 15 minutes of generating it by VisualParadigm software. The graphing technique used in TraVis proves to be much more efficient: 58 seconds to generate the graph, and it offers more information at first glance. The Java graph is shown in Figure 5.2. It also guarantees not to change when recording traces and not to get less readable when recording traces. The class diagram would become almost impossible to navigate with even a hundred of the recorded and represented traces.



(a) With black borders **(b)** Without black borders

Figure 5.2: Entire Java library from (Mac OS X) represented in TraVis with all packages, classes and methods.

5.3 Package Details

5.3.1 travis.model Package

travis.model.project Package

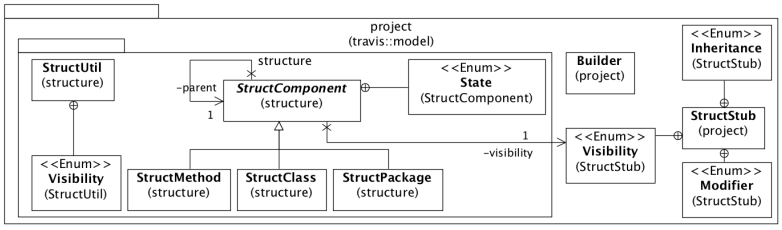


Figure 5.3: travis.model.project class diagram.

`travis.model.project` is a package representing a Java project. `Builder` is used to parse a folder containing binary Java files. It does so by recursively navigating from the root folder, making every folder a sub-package, and using the ASM framework it breaks down all **.class* files into classes and methods.

The root folder is marked as *default package* and the composite pattern is used to make the folder structure into a tree. `StructComponent` with the classes inheriting from it is one of most important elements of TraVis as it is used to build trees in the `view`, which are used to generate the BTrace script.

`StructStub` is a class that is used by the `view` when creating new elements of a Java project. Instead of creating various elements like *package, class* or *method*, a stub can be created, which can then be used to create the listed elements.

travis.model.attach Package

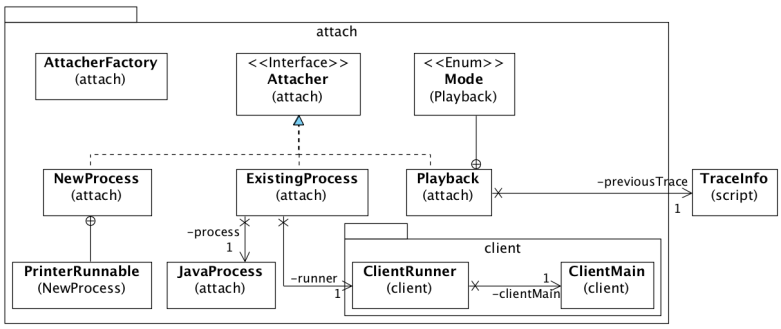


Figure 5.4: travis.model.attach class diagram.

The `Attacher` interface, as its name suggests, indicates a type of object that is used to attach to various processes. There are three concrete implementations of this type.

The first one is `NewProcess`. This class, given the parameters used to start up a new Java process, configures a BTrace agent and creates a new Java process with the configured agent attached to it. It uses `PrinterRunner` to capture all the output from the created process and forwards it to the print stream.

`ExistingProcess` on the diagram looks more complex than `NewProcess`, but is actually simpler. It takes a `JavaProcess` representation as input and connects to it using its PID. `ClientRunner` is used to create the BTrace client, `ClientMain`, which can be shut-down on request.

The last type of `Attacher` is `Playback`. It is used to play back traces that are saved to a file. It uses a state pattern to handle its distinguishable playback `Modes`. These are `PACKAGE`, `CLASS` and `METHOD`. Depending on the mode chosen, appropriate traces will be consumed without `Thread.sleep(wait)` during playback, creating skip illusion in the `view`. It is called skip illusion because all the traces (`TraceInfos`) are still sent and recorded but not drawn by the `view`. Timer classes provided with Java were not suitable for the playback, so `Playback` is the author's custom creation of a timer as well as `Attacher`. It manages, plays, pauses and stops threads seamlessly and safely without blocking any other threads.

The last class in this package is `AttacherFactory` which uses the abstract factory pattern to create appropriate `Attachers`.

travis.model.script Package

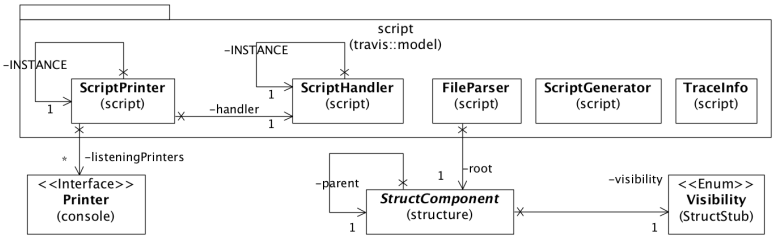


Figure 5.5: travis.model.script class diagram.

The `travis.model.script` package is responsible for various tasks related to script and trace processing. `FileParser` opens a TraVis file (extension **.vis*), restores the root `StructComponent` from it and obtains information like: the start of traces and their length (used by `Playback`), and calls depths (used by the `view` to create trace history graph).

`ScriptGenerator`, given a set of `StructComponents`, generates the BTrace script for methods inside these components, compiles it and preserves the methods' IDs, as these are unique and used to create the graph nodes.

`ScriptHandler` uses the observer and singleton pattern, where `ScriptPrinter` uses them too, as well as the decorator pattern. `ScriptPrinter` uses these patterns because it is used in place of `System.out`. Ordinary system out data is printed and sent to listening printers using the observer pattern. Whenever it captures a trace it passes it over to `ScriptHandler` which wraps it in a lightweight object (`TraceInfo`), passes it to its observers and writes it to a buffer. Actual writes to a file take place on demand, i.e. file save, or every 2000 traces using `FileWriter`, as this demonstrated the best performance. `ScriptHandler` is also used to perform file saves and copies using `FileChannels`, which on many OSs transfers bytes directly from the source channel into the filesystem cache without actually copying them.

5.3.2 travis.controller Package

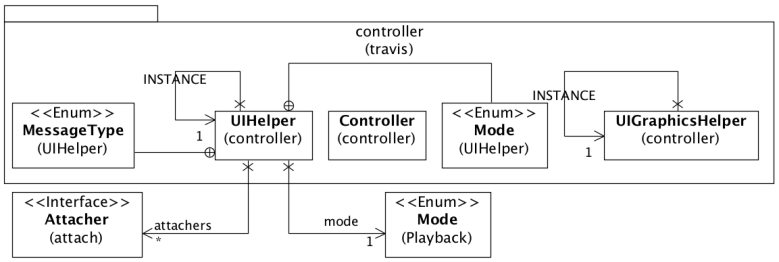


Figure 5.6: travis.model.controller class diagram.

`travis.model.controller` does exactly what is expected of a `controller`. The `Controller` is responsible at the start of the application for initialisation of the system and afterwards works as an observer, controlling the graph repaints and delegating the tasks to other elements via `UIHelper` and `UIGraphicsHelper`.

`UIHelper` and `UIGraphicsHelper` use the singleton pattern so they can be accessed from any location in the view. `UIGraphicsHelper` can be contacted to delegate repaint jobs of different types to graphical components. `UIHelper` uses the state pattern to keep track of which `Mode` it is in (`PLAYBACK`/`ATTACH`), is used throughout the view for various notifications (`MessageType`), handles process connections and splits and delegates tasks like file opening/saving.

5.3.3 travis.view Package

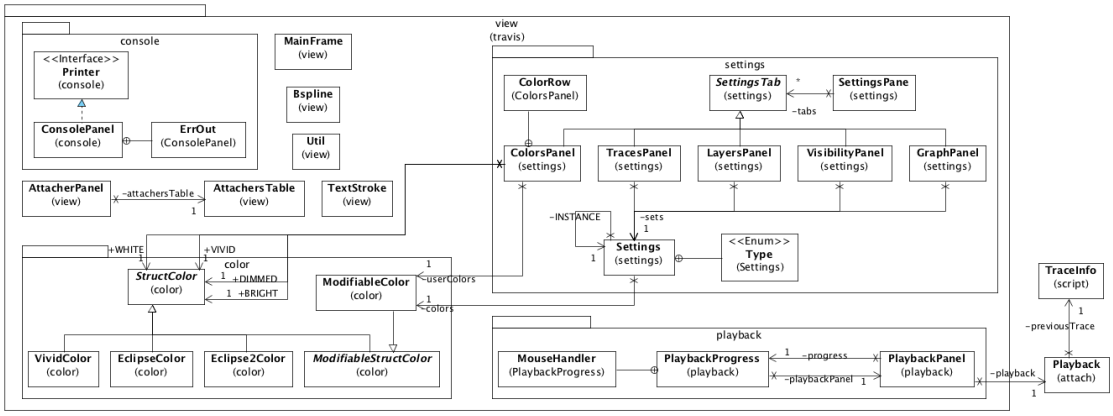


Figure 5.7: travis.model.view class diagram.

Throughout the `view` package the adapter pattern is used, this pattern is ubiquitous in Java Swing. In the `view` most of the packages appear to be more complex than elsewhere. This is something that applies to the vast majority of graphically rich applications.

The `Console` package is responsible for a console view inside TraVis. It prints out standard output and errors from processes created by Travis and from TraVis itself. `MainFrame` is simply a main window with all of the menu items and other `view` elements nicely laid out in it. `TextStroke` is used for curved stroke creation. `Util` provides static methods that are used by some of the `view` elements.

`Bspline` is a Cubic B-Spline implementation. It is very efficient but its implementation could be improved upon. For the needs of the graph it was extended with functionality like: spline straightening [17], nearest point, transparency and gradient. Gradients for various transparency levels and coefficients are precalculated for performance reasons.

`AttacherPanel` and `AttacherTable` are used to allow users to establish connections to existing Java processes, create new ones and manage them all. All active attachers are presented in a custom-made table, since Java table implementation does not support buttons.

`StructColor` represents read-only colours of the graph. Read-only colours are used for themes, where the users can use `ModifiableStructColor` to change the graph colours to their preferences via `ColorsPanel`, which is one of the `SettingsTab`s.

`SettingsPane` handles all its tabs and delegates updates to them. `SettingsTab`s, as the name suggests, allow the users to edit `Settings`. A single `Settings` object is shared for the entire application and can be accessed from anywhere since `Settings` uses the singleton pattern. Whenever the `Settings` object changes it notifies its observers of a change and includes the `Type` of change. This greatly increases the overall performance of TraVis since only the elements that need to be updated are updated. Different `view` elements depend on different settings.

The most interesting `SettingTab` is `VisibilityPanel`. It not only allows the user to change various visibility settings, like which classes or methods they want to see, but it automatically handles enabling and disabling different checkboxes using one unified method. This functionality is needed when, for example, a user hides methods and classes and it is needed to block him from hiding packages too.

Playback controls are only active in playback mode. All `PlaybackPanel` is responsible for is laying out play controls, delegating their actions to `Playback` and managing things like speed and mode (packages, classes, methods) of playback. `PlaybackProgress` is responsible for creating a trace history graph (pixel

by pixel), updating playback position, and allowing the users to select a section of a trace to play it back or save as a subtrace.

travis.view.tree Package

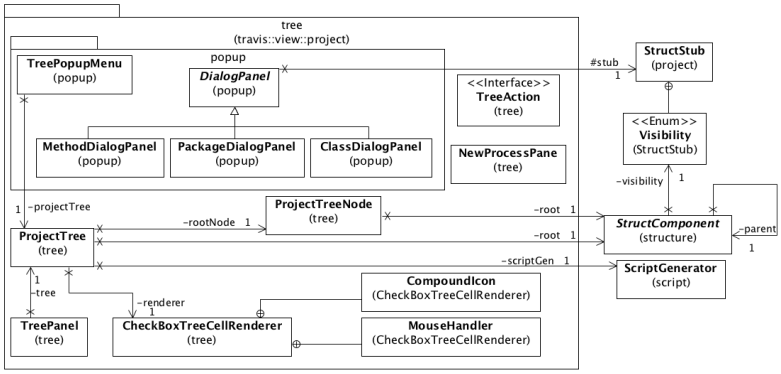


Figure 5.8: travis.model.tree class diagram.

`travis.view.tree` in the second biggest and most interesting package, and in this section is only described in small detail. Further details are included in Appendix B.

The complexity of this package was to a large extent caused by the way in which Java Swing `JTree` is implemented and how communication is handled between a tree, nodes and a renderer.

`CheckBoxTreeCellRenderer` extends `TreeCellRenderer` and provides additional functionality in the form of the ability to select an arbitrary number of children. The original implementation of this class was extended by implementing: a, ubiquitous in Swing, listener functionality that is fired on selection change; resetting checked paths; and support for ECLIPSE icons and labels. The icons are created by, not listed in the diagram, `IconFactory`. This factory uses the abstract factory pattern and, based on the `StructComponent` provided, it builds (or uses cached) icons for *class*, *enum*, etc. with appropriate annotations like *static*, *private*, etc.

`ProjectTree` manages `JTree` and handles operations like: handling state changes, building the tree, rebuilding the tree, attaching the tree, delegating script generation from selected children, dealing with node selection and actions related to selected methods. For quite a few of the listed operations it is using the proxy pattern. A combination of the template method and command patterns is used for rebuilding the tree; `TreeAction` is used in this process.

`ProjectTreeNode` extends `DefaultMutableTreeNode` and overrides many of its functions to simplify code elsewhere that uses the nodes. It also provides some extra navigational functionality and, most importantly, a custom-made binary search that works very efficiently and as a bridge between `StructComponent` and `ProjectTreeNode` (yet another `JTree`-related type...).

`TreePanel` is simply responsible for laying out `ProjectTree`. `NewProcessPane` is somewhat complex dialog window that is presented to the users immediately before Java process creation.

`TreePopupMenu` is responsible for displaying the correct choice of menu items that are used for the project tree manipulation. This manipulation might include the creation of new *packages*, *classes*, or *methods* or the deletion of any of these elements. Upon completion of the task it constructs the right `TreeAction` to rebuild the tree.

The appropriate `Dialog panel` is displayed upon `TreePopupMenu` item selection. `Dialog panel`s use `StructStub` for creation of new elements. This adds scalability and hides `StructComponent`'s implementation.

travis.view.graph Package

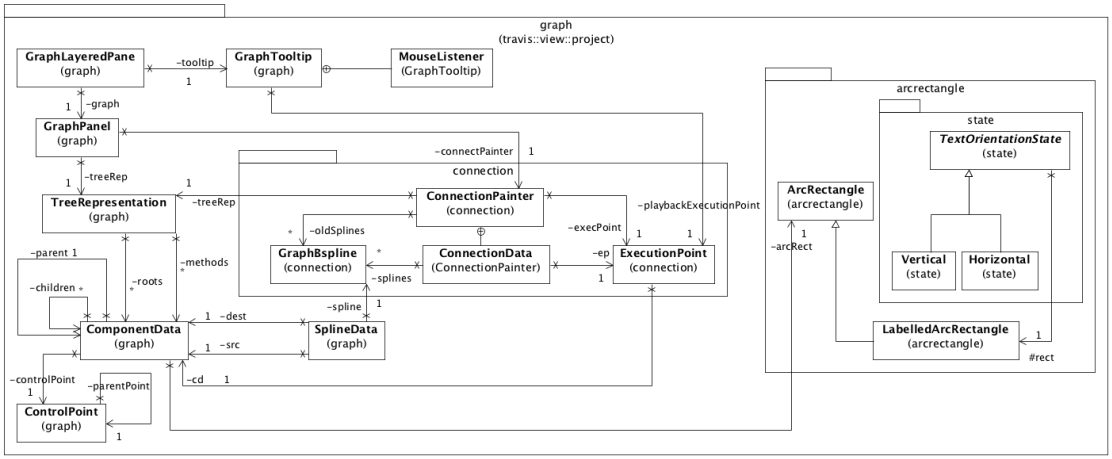


Figure 5.9: travis.model.graph class diagram.

`travis.view.graph` is by far the most complex and intriguing package. This section contains the minimum information about it and further details and challenges related to it are included in Appendix B.

`ArcRectangle` is the most basic, custom-made, building block of the graph. It provides various methods required to perform its manipulation and layout. `Labelled-`

`ArcRectangle` adds a label to `ArcRectangle`; it does so at runtime by using `TextOrientationState`, which uses the factory method pattern. `TextOrientationState` determines the exact size, position, curvature and orientation of the text dynamically.

`ArcRectangle` is used by `ComponentData` which is an ultimate wrapper class for *all* the data needed to be captured in the graph. `ComponentData` uses the composite pattern to become a bidirectional tree. It uses `ControlPoint` to indicate the first connection point of its `ArcRactangle`. `ControlPoint` is a custom-made point that differs from an ordinary point by knowing its path to the graph's centre and by being able to navigate to anywhere in the graph.

`TreeRepresentation` is effectively a graph representation and is the most complex class in the entire system. It makes extensive use of highly complex logic and mathematics, especially trigonometry. All this complexity was used to create the main graph, its inner layout and ultimately to meet absolutely all visibility scenarios; that it was possible to achieve with the available `Settings`.

`GraphPanel` aligns `TreeRepresentation` and `ConnectionPainter` to the centre of the available space. `GraphLayeredPane` creates an additional layer on top of `GraphPanel`. The created layer is used by `GraphTooltip`; which, by being in a separate layer, can change the look of the graph without needing to repaint the entire `GraphPanel` every time a mouse move is registered. This would be extremely performance intensive. `GraphTooltip` displays different tooltips for different elements on the graph in different modes. It also displays a tooltip for the execution point in playback if fewer than 4 traces are displayed per second; it is able to display information for multiple graph nodes at the same time - `SplineData` is used to obtain information about the source and the destination of the node.

`ConnectionPainter` deals with painting of all the connections in all different modes. Each connection/node is drawn as a `GraphBspline`, which is a `Bspline` extended with the information about the "caller" and the "callee" of the trace. `ExecutionPoint` is a point that is able to change its size and keeps track of its current `ComponentData`.

Finally `ConnectionPainter` uses `ConnectionData`, which is essentially a wrapper class for the connections to draw. `ConnectionData` uses the strategy pattern to allow the users to switch between *Unique Calls/Traces* and *Latest Calls/Traces*. `ConnectionPainter` uses a very complex, but clever, function to create graph connection. The function is both recursive and iterative. This approach was needed to maintain access to the entire call history from any point and time in execution.

Chapter 6

Verification and Validation

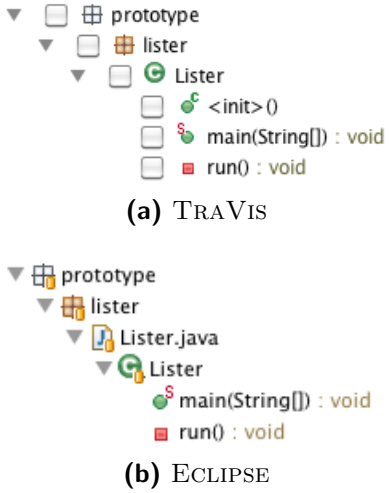
Some of the testing challenges were mentioned in Chapter 4 in Section 4.2 and will be discussed in far greater detail in this chapter. User testing with the results is presented in Chapter 7.

6.1 Testing

All the elements were tested rigorously as modules. The following subsections only describe testing of the main TraVis elements. More detailed test specifications can be found in Appendix C.

6.1.1 Project Tree Representation

Project tree representation validation was actually quite trivial to do, since it is based on Eclipse's look and feel. There are two package representations in Eclipse. The first one is, a quite unpopular, flat representation that simply lists packages in alphabetical order. The second one is hierarchical, which is quite similar to folder representation. The hierarchical presentation was adopted in TraVis and the comparison of the two can be seen in Figure 6.1. The only difference in tree 6.1a is that it represents bytecode and not source code. This is why it contains <init>(), which is the default constructor.



(a) TraVis

(b) Eclipse

Figure 6.1: Java project tree representation.

6.1.2 Graph Representation

When the graph representation is working perfectly it is incredibly difficult to describe the amount of effort and testing involved in getting it right. There is a vast amount of features and coding involved in the graph, and the number of possible generated graphs is infinite.

The main building blocks of the graph are `ArcRectangle`s, thus they were the first to be throughly tested by changing the rotate angle, angle extent, height, scale and fill colour. After this was done the same tests were performed on a multiple number of `ArcRectangle`s on the same layer.

Text support was implemented and tested before multiple layer support. Testing `LabelledArcRectangle`s involved the above while ensuring that the text never goes outside its boundaries and remains in the middle of the element. These tests were performed for both horizontal and vertical labels, the choice of which is made dynamically at runtime. Constraints like text size and orientation added to complexity. Vertical text is the easiest to read from 0° to 179°; from 180° to 359° it should be flipped by 180°. The same applies to horizontal text but the ranges are 270° to 89° for normal text and 90° to 269° for flipped text. Double precision caused additional problems.

When multiple layers were added testing began for all of the above scenarios, and new tests were added for hiding various layers as a whole, hiding parts of them, changing the gaps between the layers, and linking them with actual packages, classes and methods. After all of the above tests were satisfied the inner layout was created and tested. Inner layout is simply the graph structure mirrored inside. This layout provides control points for Cubic B-Splines connecting the graph elements.

The scenarios listed above were exhaustively tested and currently there is not a single known issue related to the graph creation. The representation was tested multiple times on real Java projects, and Java libraries, and it never created an inaccurate representation.

6.1.3 Trace Capture and Filtering

Capturing and filtering traces in TraVis is done via the standard output stream. When TraVis is connected to an already running program this task is made a little easier as captured traces cannot be mixed with the monitored program output. This becomes little bit more difficult for processes created by TraVis as

their output often is mixed with standard output of the created process, instead of being received as a new line.

Testing of this problem was mainly done by tweaking the BTrace script and making sure that the regular expression that is created for trace filtering works with all possible scenarios. Additionally traces were printed to the console and manually compared with the execution flow of multiple Java processes.

6.1.4 Playback Representation

From the view, as in MVC, perspective playback representation is identical to drawing captured data. Because traces are created so quickly at runtime it is not possible to test the correctness of drawn traces at runtime.

Playback validation was done by playing and pausing drawing traces at a very low speed and manually comparing the on-screen results - graph nodes - with traces stored in a file. This process was repeated a number of times for many Java processes. Once this worked correctly for methods, testing began (in the same manner) on playback at class and package level. Upon satisfaction of these tests, thread handling was tested, where nodes were created only between threads with the same IDs.

TraVis allows changes to visibility of packages, classes, methods and their subtypes (abstract, enum, public, private, etc.) at runtime and during playback. These were tested in a similar fashion to the one described above, while ensuring that the traces for hidden elements are ignored and in playback mode playback speed is maintained even if traces are skipped for elements that are not visible.

6.2 Performance Analysis

The Fallacy of Premature Optimisation [18] is a great article describing the problems encountered, and mistakes made, by many software engineers when creating optimised software. Awareness of this issue not only made the development of TraVis more enjoyable, but the optimisation of it much less problematic.

Finding bottlenecks in software is hardly easy, and fixing them often requires multiple changes across many classes and packages. Because of the way TraVis is structured, it was always easy to identify bottlenecks and fix them. Despite the fact that the author is inexperienced in graphics development the final product is really efficient and pleasant to use.

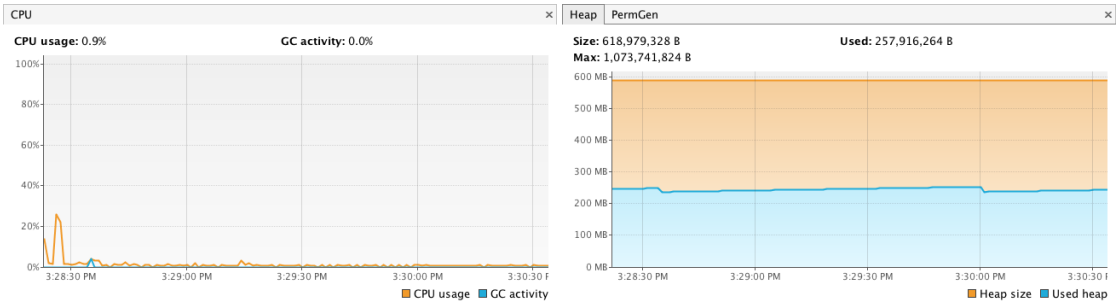


Figure 6.2: CPU and memory usage of TraVis when connected to a process.

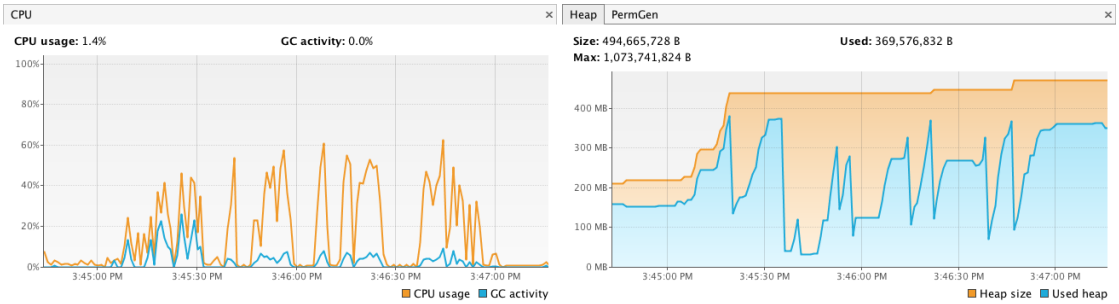
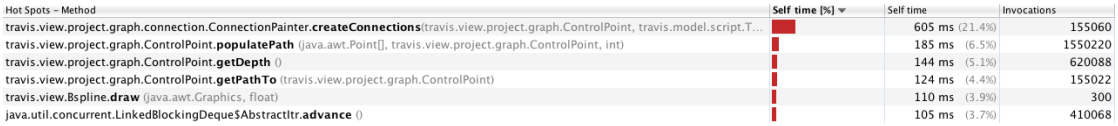


Figure 6.3: CPU and memory usage of TraVis when connected to a busy process.

TraVis, similarly to BTrace, has been designed not to utilise system resources when in idle mode or connected. Figure 6.2 shows TraVis usage statistics when connected to another process and the monitored process is not performing any work. The data was collected using VisualVM, which is a lightweight Java profiling tool. The early spike in CPU usage is caused by VisualVM connecting to TraVis. Memory usage stays on about 250 MB. It is so high because a standard garbage collector, *serial collector*, is used in this process and, as can be seen from the graph, it has not run since the profiling started. There was no need to run the collector as over 50% of available heap space was free.

Presenting idle usage in profiling tools, like TraVis, is important because no additional performance overhead should be created if a monitored process is not performing any work. When a monitored process does perform some work TraVis is using similar amounts of CPU as the monitored process. In some cases it is less than the process itself; it all depends on how many methods are called. Implementing a 25 FPS cap on updating the graph often results in collecting lots of traces but committing only a few updates. The CPU usage rarely goes beyond 60% because of this cap. Figure 6.3 shows the resource usage of TraVis when a monitored process performs some tasks.



Hot Spots – Method	Self time [%]	Self time	Invocations
travis.view.project.graph.connection.ConnectionPainter.**createConnections**(travis.view.project.graph.ControlPoint, travis.model.script.T…		605 ms (21.4%)	155060
travis.view.project.graph.ControlPoint.**populatePath** (java.awt.Point[], travis.view.project.graph.ControlPoint, int)		185 ms (6.5%)	1550220
travis.view.project.graph.ControlPoint.**getDepth** ()		144 ms (5.1%)	620088
travis.view.project.graph.ControlPoint.**getPathTo** (travis.view.project.graph.ControlPoint)		124 ms (4.4%)	155022
travis.view.Bspline.**draw** (java.awt.Graphics, float)		110 ms (3.9%)	300
java.util.concurrent.LinkedBlockingDeque$AbstractItr.**advance** ()		105 ms (3.7%)	410068

Figure 6.4: Most expensive methods when drawing connections.

The usage range of 40% to 60% might feel a little bit excessive. However, graphical calculations were always the source of the highest CPU usage. TraVis is not hardware-accelerated and these spikes should not concern anyone. Actually, even the simplest tasks like scrolling through a webpage on the same machine, on Mac OS X using the Chrome browser, uses 35% of the CPU.

The CPU usage is not noticeable by the users as it is happening in bursts. Nonetheless, it is important to identify which methods are causing this issue. Figure 6.4 presents the most expensive methods and, unsurprisingly, `createConnections` is by far the most expensive. This method is doing a significant amount of work, it is by far the most complex in the entire system and definitely could be improved upon. Details of possible improvements are described in Chapter 8, Section 8.1.

The most important thing about TraVis' performance is that the presented data does not directly affect the monitored process. Since TraVis uses BTrace, the overhead induced is negligible. It should not come as a surprise that monitoring methods like `hashCode`, `toString`, etc. induce much more overhead.

6.2.1 System Requirements

Graphical tools for software engineers, like TraVis, Code Bubbles or Code Maps have one requirement or recommendation in common. This is the screen resolution. TraVis' features are best utilised on two monitors with at least one of them having a resolution close to or greater than 1080p. It is not just the tool that looks better in this resolution; the user benefits from having much better access to the information by seeing more of it.

Memory is crucial for creating the graph, creating the BTrace script, and compiling it, so it is recommended to use TraVis with heap space increased to a maximum of 1024 MB, however 512 MB should be sufficient in most cases. Any modern CPU over 1.6 GHz should be enough for satisfactory operation.

Chapter 7

Results and Evaluation

7.1 Preparation

TraVis is a tool designed to help programmers understand medium to large systems without the need for prior understanding of the systems. User experience was at the highest priority throughout the development of TraVis, and extra care was taken to ensure that the users, independently of the programming experience, would find it intuitive to use.

In order to perform user evaluation, a system of appropriate size and complexity had to be chosen. The aim was to present the users with a fairly large system that they were not familiar with. Despite the fact that in similar research evaluated systems contain a minimum of 145 and a maximum of 344 classes [7, 8, 15], for TraVis evaluation jEdit was chosen as the subject system. It is an open-source project written entirely in Java and contains a total of 158,210 lines of code in 1099 classes (including inner classes) that are spread across 28 packages.

In order to design exercises, a certain understanding of jEdit had to be established. This was achieved by using TraVis alongside Eclipse, which was used for code modifications. The user group for the evaluation consisted of one undergraduate, one postgraduate, three PhD students and two professionals from JP Morgan Chase. All the users had a good understanding of Java with programming experience varying from 4 years, for the undergraduate student, to over 20 years, for the Vice-President at JP Morgan Chase.

Prior to the exercise part of the evaluation, users were presented with the introduction, which was followed by an explanation and demo of the main features found in TraVis. Due to the fact that TraVis is a novel, relatively complex

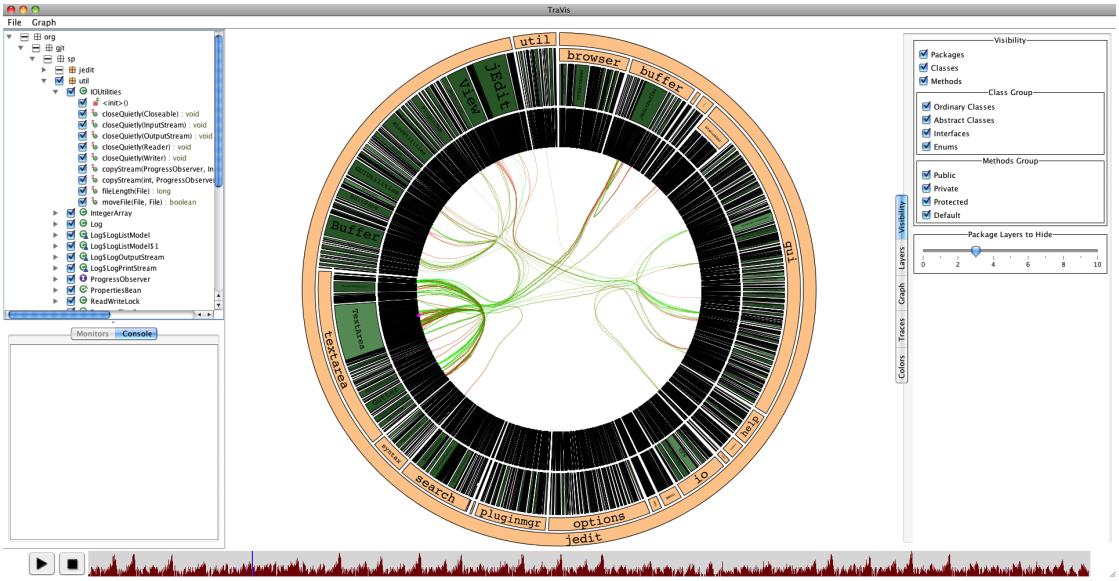


Figure 7.1: TraVis in playback mode.

tool, presented in Figure 7.1, and it is used on quite a large system the number of exercises is small and the difficulty of them steeply increases. After the exercises were completed, users were asked to fill in an evaluation form. Evaluation introduction, exercises and form are all located in Appendix E.

7.2 User Evaluation

The graphing technique used in TraVis is intuitive to read for humans [17], but there was a worry that this might be spoiled by nodes on the graph constantly disappearing and reappearing when actions are performed by the user or jEdit itself. The execution of a large number of methods is happening incredibly quickly on today's computers, and capturing and visually representing the interaction that is happening between the methods, classes or even packages has never been attempted before. When participants connect to jEdit they are presented with an already quite complex looking graph, shown in Figure 7.2, and multiple things

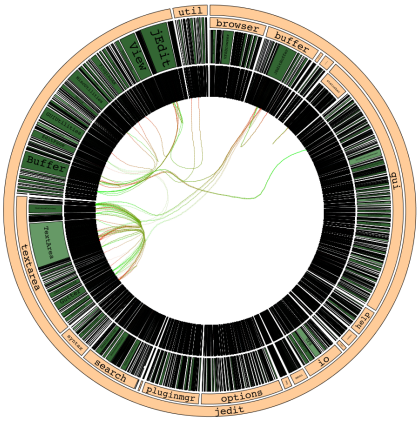
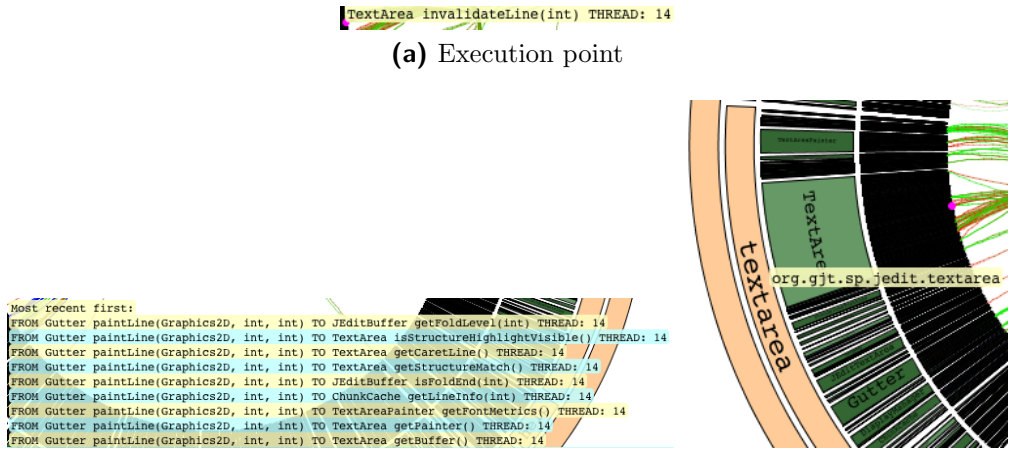


Figure 7.2: jEdit graph generated by TraVis shortly after establishing connection.

happening concurrently in jEdit are making matters even more challenging by spontaneously creating new connections on the graph.

Surprisingly none of the participants found the issues described above confusing, or probably more importantly, distracting. Everyone started by learning about TraVis and jEdit by exploring the graph and using tooltips, displayed in Figure 7.3, to obtain more in depth information. In fact, being used on a system of this scale, if TraVis had not had the tooltips it would not have been of much help. Tooltips were listed many times as one of the most useful feature of TraVis.



(a) Execution point

(b) Multiple nodes

(c) Package

Figure 7.3: Tooltip examples.

Observations during the first exercise are probably most important, as at this stage users are really not familiar with the tool. It was possible to observe a slight confusion as to where what controls are located and how the users should go about solving the tasks. Figure 7.4 shows the graph users were presented with at the end of the first exercise. Some users intuitively navigated to the execution point, others limited the number of drawn nodes so they only saw the latest one that occurred and a majority decided to increase the size of execution point. Everyone obtained information about classes and methods by using the tooltip, then in Eclipse located the actual cause of termination.

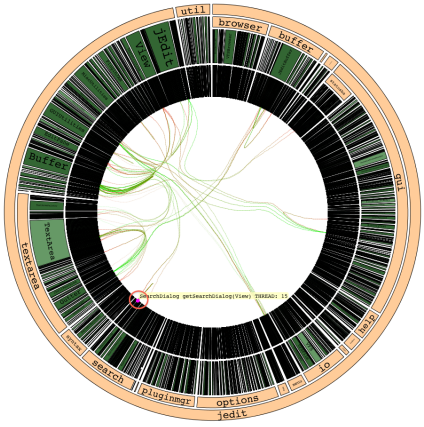
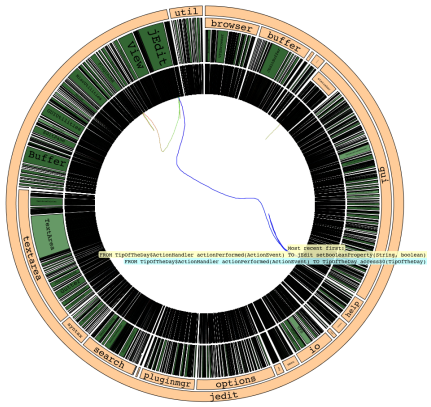


Figure 7.4: Graph at the point of jEdit termination with highlighted execution point and displayed tooltip.

In the second exercise users were asked to locate the cause of the performance dropping every 30 seconds. With useful TraVis features listed next to the exercise description, and by understanding how the execution point works, users knew that all they had to was to watch the graph and see what was happening when jEdit stopped responding. In this case it was possible to find the bottleneck without statistical information, because the bottleneck was happening for long enough - about 2 seconds - to be spotted on the graph. Statistical data was listed a couple of times as the missing feature.

The next exercise was much more challenging and required more TraVis functionality to be utilised. Finding a single field in a system of jEdit's scale could be a really tedious task, and even with a tool like TraVis it is not immediately obvious as to how the users should go about it. Two of the most effective and time saving solutions derived by the participants are shown in Figure 7.5. Figure 7.5a shows the graph generated by connecting to the jEdit process at the time when *Tip of the Day* is already visible and clicking the appropriate checkbox. Using tooltip on the graph mitigates the need for saving the traces. Another approach is where the user repeatedly clicks on the appropriate checkbox and then saves the trace. When the traces are reopened it is easy to spot the pattern, presented in Figure 7.5b, and identify the methods that were called on the event. In the cases of both solutions, Eclipse was used for analysing jEdit's code of the methods of interest and identifying the field that was required to solve the exercise.



(a) Graph

(b) Execution trace playback

Figure 7.5: Exercise three - possible solutions.

The final exercise, and by far the most challenging, required from the user identification of the class responsible for handling operations like *Copy* or *Cut*. By observing users' behaviour it was possible to immediately spot the problem. Most, but not all, of the users did not think about choosing the shortest path to call the operations which would generate minimal amounts of traces. Performing tasks like typing the text in `TextArea`, selecting it, choosing the Copy operation through the menu, instead of a toolbar button or using a keyboard shortcut, resulted in

(a) Trace graph

(b) Subtrace graph

Figure 7.6: Exercise four - generated graphs.

generating much larger amounts of data and traces than were necessary and by it made the task even more difficult. Because of the use of many patterns in jEdit with every operation traces were generated, but most of the users had learned by this time that repeating the operation would make it easier to spot the source of it during the playback. Figure 7.6a shows an example of a trace graph generated from traces captured during the connection to jEdit. Not many patterns can be seen in there and it is difficult to make sense of anything in there, hence most of the users decided to look at the region where a spike occurred. Since proper zoom-in functionality was never used participants were asked to save the area of interest as a sub-trace, results of which can be seen in Figure 7.6b. This not only improved the graph's readability but also made the patterns more visible.

From the above saved sub-trace it still is not crystal clear where the patterns are and even playing part of them back often was not providing a clear answer. The common problem encountered in here was that the participants were focussing on a really small subsection of the trace and kept playing it back. Often it lead them to incorrect conclusions. Despite that there were many interesting observations made.

TraVis allows the dynamic disabling of packages, classes and methods; one user decided to use this as a zoom-in function so that he could focus on the areas he found important. The result of this can be seen in Figure 7.7 with a circle highlighting the area which called many other classes but was barely called itself. This conclusion was made on the fact that all the nodes coming from it are green on its end and red on the other, which means it is a caller and others are callees. A different user played-back many of the patterns at high speed to see what classes were regularly invoked when the *Copy* operation was called. Another interesting observation was made when one of the users decided to play the patterns back at low speed with methods made visible. When a trace is played back at less than four traces per second the execution point is accompanied by the tooltip, which includes the current class and method names. What is really impressive, is that the user who decided to use this strategy was actually able to locate the very method that was performing a copy from the `TextArea` into a `Buffer`.

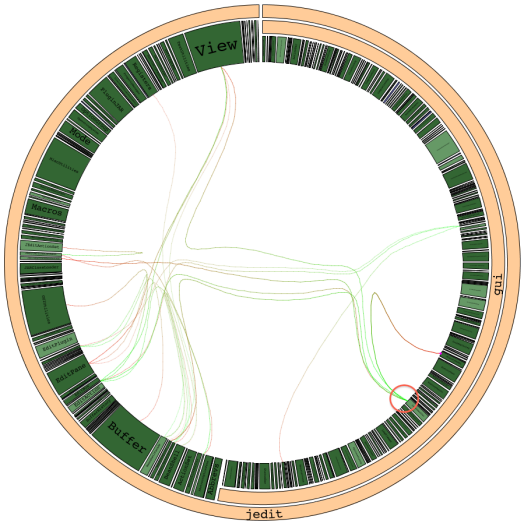


Figure 7.7: Graph generated after disabling unneeded components.

1. How difficult did you find the exercises?
2. How easy did you find it to use?
3. Was it of help in dealing with the exercises?
4. Do you feel it saved you time?
5. Did it help you learn about and understand jEdit?
6. Would you use it for learning about medium/large systems?
7. Would you use it for locating functionality of medium/large systems?

Table 7.1: User evaluation questions.

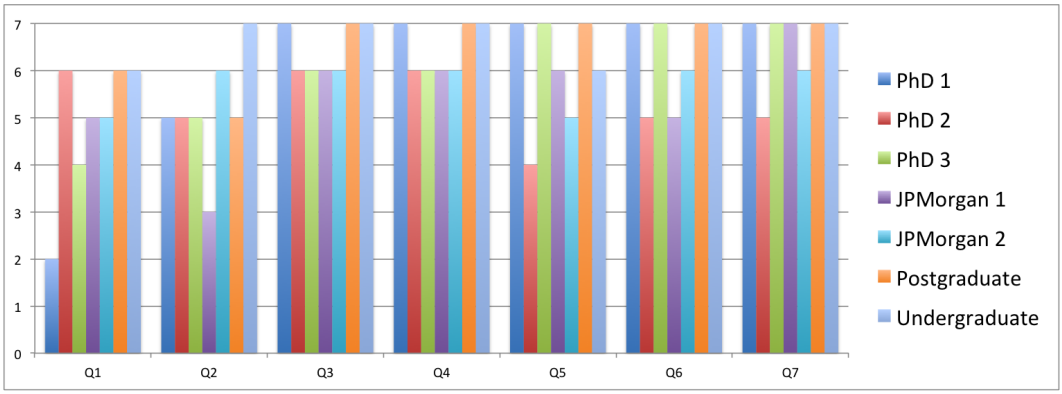


Figure 7.8: Chart representing questionnaire results. Scale: 1 meaning poor/difficult/no and 7 meaning excellent/easy/yes.

7.3 User Evaluation Results

Results of the evaluation are very promising, as presented in Table 7.1 and Figure 7.8. The exercises appeared to be very difficult when considering the use of a new tool, the size and complexity of jEdit, and the knowledge required to accomplish the exercises, yet on average it took the participants about 50 minutes to accomplish all the exercises. Only one of the users found the exercises difficult, 2 in a scale to 7. The likely cause of it might be a lack of recent experience on a medium or large system. Generally all of the users found the exercises relatively straightforward, however when asked "How difficult, do you think, would you find the exercises if you could not use TraVis?" all of them uniformly said that they would find the exercises very challenging.

In terms of ease of use of TraVis, all of the participants found it quite easy and intuitive to use. Two of the participants stated that it appears to be quite complex, but after using it for a short time they quickly got used to it. However, both of the participants said they would welcome more features.

When participants asked why they would/would not use TraVis again the most common responses were positive: saves time, great addition to programmer's toolkit, easy and quickly locate hotspots and calling methods and finally accessible, clean and intuitive interface.

Favourite pieces of functionality included: playback, playback speed, tooltip, execution point, the way it shows multiple paths and their information as a group, most called class / method, hiding package layers, and others like add / remove packages, classes or methods and connect to multiple processes in parallel i.e. client / server. The last two points were presented outside the evaluation and are described in more detail in the User Guide located in Appendix F, it also includes a more in-depth description of the functionality.

Chapter 8 includes the details of the future development and functionality that is planned to be added to TraVis. The most missed functionality noticed by the participants included: a log of the most recently executed traces, proper zoom in/out on both main and trace graphs, various statistics about traces, integration with IDE - the ability to click on a method to see code, step back in playback.

In the *Other comments* section, participants included plenty of compliments and less important requests like, in the tooltip a message separate class and methods with a dot instead of a space i.e. `MyClass.myMethod()` instead of `MyClass myMethod()` as this is more intuitive to programmers.

Most importantly, all of the users feel very positive about TraVis. Everyone evaluated responded extremely positively to questions 3 to 7. These questions are a great indication of the potential of TraVis and the benefits it can bring no matter what your programming experience is. One of the PhD students, who is involved in teaching undergraduate students, said it could be very useful when teaching young students about execution flow. The Vice-President from JP Morgan Chase stated that he could see its potential and the benefits of using it for many people and corporations, including big, successful ones like JP Morgan Chase itself. All of the gathered feedback is evidence of TraVis's potential for being used in many different areas of industry and education, while benefiting everyone. With some more work and input it could become widely used in no time, as said by many participants.

7.4 Project Results

The most detailed functionality and results of TraVis can be found in the User Guide located in Appendix F.

TraVis turned out to be very successful; it not only meets all the requirements specified in the project outline, but goes beyond them. It allows the users to navigate to the location of the binary files of any Java project, it then parses the project and creates an adequate, Eclipse-like, tree representation of the project. Throughout the use of TraVis the users are prompted with informative messages if anything unexpected or erroneous happens.

The creation of a new process is an obvious use of TraVis. It is possible to provide all the Java parameters that can be included when running the process from an ordinary Command Line. These include selection of working directory, Java options (i.e. *-jar, -Xmx256m*), custom program arguments, and selection of custom start source (i.e. *my.jar, main.class*) or selection of a main method, from the list provided. When the connection is established, the entire output of the process is captured and printed to TraVis's console.

A connection to any, already running Java process can also be established. This connection is inherently safe and will never result in termination of the monitored process. Multiple connections can be created at the same time to allow the users to monitor interaction between various processes simultaneously.

TraVis is very robust and making connections to any Java processes is only limited by the Java compiler, Java VM and available resources, like disk space or memory. The time needed to establish a connection is dependent on the number of selected project tree nodes; the more that are selected the longer it takes, since more probes needs to be inserted.

The key elements of TraVis are: graph representation of the project, edges representing interaction between elements and trace graph, available in playback mode. The user interface is highly customisable, from colours, through the amount of drawn edges to the height and gaps of individual graph's layers. Various elements of the Java projects can be included in/excluded from monitoring, and the design of TraVis allows changing of all available settings. All of these features are available dynamically, at runtime, meaning that users are free to amend the areas of interest of the processes, after the connections have established.

TraVis is intelligent enough to distinguish between the various settings, and change its view, tooltips and edge drawing techniques to accommodate new changes. With this the users can vary the granularity of information at runtime.

In the playback mode the users are free to take advantage of all of the above functionality and some extras. The speed of the playback can be set manually, enabling the users to watch the execution of the programs in slow motion; execution of less than 4 traces in a second enables a tooltip next to the execution point. The

current playback position is indicated on the graph and can be paused at any given time. A subsection of the gathered traces can be selected, played and saved to a separate file.

By far the greatest benefit of TraVis is empowering its users to effectively and efficiently work on, develop and maintain systems of their choice. With TraVis, the need of prior understanding and knowledge of the systems is completely removed.

Chapter 8

Conclusions and Future Work

TraVis turned out to be a much greater success than anticipated. Everyone, not only evaluation participants, who had chance to experience it was really positive about it. There were two factors that had the greatest impact on the development process, performance and user interface. Although they are implemented very well there are still many ways in which they can be improved upon.

8.1 Performance

It is a common problem with analysis tools that they hinder performance and become the top process, even if the system they are attached to is not resource-intensive. TraVis on the other hand, depending on the configuration, has the potential not to use up large amounts of resources and the delays imposed by it to the monitored process are often not noticeable. Having said that, in some cases performance penalties might be high. One case is where a user monitors methods that are not doing a significant amount of work, like `equals(Object obj)` or `hashCode()`, and these methods are called many times. `Collections` are prone to it, as for example `HashMap` uses both of the mentioned methods for most basic operations like `get(Object key)` and `put(K key, V value)`. This problem can be fixed by presenting to the users a list of common Java methods that can create the above problems, and let them choose if they want to automatically disable them across all the classes and not create a `BTrace` script. The users would be able to add methods of their choice too.

Creating the main graph is by far the most resource expensive and complex task in TraVis. With some effort it should be possible to optimise and make it less

complex. In its current implementation, every time the graph is rotated or rescaled it is recreated; creation of it includes getting all selected nodes from the project tree and creating all the graphics. The obvious way of greatly improving the performance of these two cases is to use transformation methods provided with Java; unfortunately, they are far from perfect and the transformed image is of significantly lower quality.

When TraVis is in playback mode, or connected to a process and traces are captured only the splines are redrawn on the graph. The graph itself is only redrawn if the user changes its parameters like layers' gaps, layers' size, rotation, disabling of some components, etc. All splines representing connections are created every time new connections need to be drawn. For every spline, lines that are making it are recalculated and redrawn on the graph. This is all happening after *all* cached traces are parsed. Fortunately, the implementation of the splines is really efficient and this does not have a large impact on the performance. But this very process is most likely to be responsible for the majority of resources used by TraVis.

Parsing cached traces is another area of TraVis which can be improved upon. Its current implementation is using a thread-safe queue, but every time connections are repainted they are iterated over from the beginning to the end. A more efficient solution, that should be adopted to solve this problem is where cached traces are iterated over from the most recent one until sufficient number of splines are gathered, and then the iteration stops. An attempt was made to implement this solution, but it proved to be incredibly challenging and due to the limited amount of time it was disregarded. The difficulty with it is that it is a bottom-up approach and it is never known up front if the previous trace is the return method of another trace or if it is the method that created this trace. As simple as it sounds, or not, it is a really complex problem.

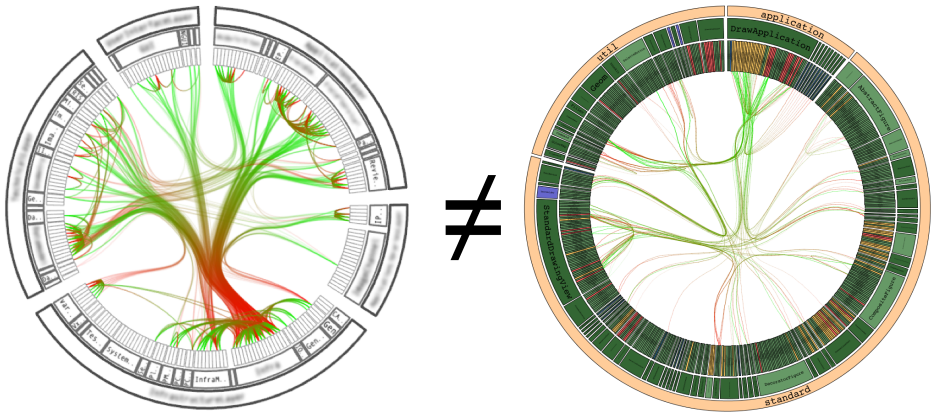
The performance problems described above might seem really severe and one may wonder if TraVis is really as efficient as it is claimed by its creator. The short answer to this is "Yes, it is really efficient". Not a single participant in the evaluation or anyone else ever said that its performance was poor. For the complexity of the task it is performing and the fact it is doing so dynamically it is really efficient. Performance and overhead induced to monitored systems were probably the reasons why no one ever before created a tool that would do what TraVis is doing. There of course are tools that are providing similar functionality and are doing so dynamically too, like VisualVM, but often they are more difficult to use and most noticeably they are not providing any visual representation of execution flow thus it is difficult to quickly understand the inner workings and communications of monitored systems.

8.2 User Interface

Lots of effort was put into making the user interface of TraVis as intuitive as it is. At first it might appear quite complex because there are many different buttons, tabs, sliders, etc. and there is the graph that is not familiar to computer scientists.

Most, yet not all, programmers are used to class diagrams as the representation of projects. There are some limitations to them though. There is visual clutter with a large number of edges or nodes. It is difficult and resource-intensive to keep them organised and readable dynamically. They offer limited visual information and can be represented in an infinite number of ways by simply dragging classes into new positions.

The graphing technique adopted by TraVis on the other hand mitigates all of the above problems related to class diagrams. It reduces visual clutter. It is easier to maintain, thus does not need as many resources. The level of visual information is greatly increased and additional details can be obtained on demand, i.e. tooltip. And finally, it is scalable. No matter how big a project is, how many packages, classes or methods it contains, it will always look the same - a circle with layers selectable and modifiable by the users.



(a) Desired look of edge bundles [16]

(c) Edge bundles in TraVis

Figure 8.1: Graphs comparison.

Edge bundles, as described by Danny Holten [16], differ in looks when compared with the ones that can be found in TraVis; this is presented in Figure 8.1. There are a few reasons for this. Probably the most noticeable one is that for the creation of the graph in Figure 8.1a OpenGL was used and it provides number of functions that drastically improve final results on top of overall better performance since OpenGL uses hardware acceleration. The second reason is that the implementation of cubic B-Splines appears to be much better, as the curvature of them is

smoother, and it seems to have better values of pre-calculated knot-vectors for the blending function. The latter might sound mysterious; it is one of the main factors responsible for their shape by setting the influence of control points.

The graph in Figure 8.1c has some advantages over the one in Figure 8.1a. The main one is that it is easy to distinguish between different layers of the graph and different types of components. Even with a large number of components it is still quite easy to see where interfaces, abstract and public classes are and different methods like public, private, protected or default.

The blending technique employed in Figure 8.1c is different to the one from Figure 8.1a because the functionality in TraVis is different. The focus is on the most recent traces executed. This technique can be improved upon by including more of the features described in the article on edge bundles [16].

Finally to improve the look of the graph and the overall performance of TraVis, Java OpenGL (JOGL) can be used for graphics generation instead of using build-in Java graphics functionality. The benefits of using JOGL are multidimensional. Firstly, and most importantly, it would be possible to make the graph much more readable by using minimax blending. Secondly some of the performance issues described in Section 8.1, like rotate and rescale, can be mitigated. And thirdly by improving on B-Spline implementation and making it use JOGL, overall performance, look and fell of splines would improve greatly.

By far the most common request by the user is zoom in/out functionality. A primitive zoom in/out can be achieved by changing the selection of tree nodes or different graph layers, but this can be really frustrating to the users. All the graphic elements of TraVis were designed and developed with scalability in mind; everything possible uses percents instead of pixels, thus implementing true zoom in/out should not require too much effort and should result in much higher user satisfaction, not that it is low just now.

Traversal of the project tree to search for packages, classes and methods would be a nice addition that can be further improved by highlighting the element on the graph and maybe allowing filtering where the users can select whether they want to focus on connections; fan-in, fan-out or both.

8.2.1 Integration as Plugin

Implementing TraVis as a plugin can be beneficial in many ways. There are two great systems into which TraVis can be quite easily integrated as a plugin.

The first one is Eclipse. As a plugin, TraVis would benefit from access to the project tree, so all packages, classes and methods that are responsible for the project tree representation inside TraVis can be removed. A console view is present in Eclipse, so the classes associated with it in TraVis can be removed too. A link between packages, classes and methods in TraVis and Eclipse can be established so there would be option to move directly to the code location in source files. TraVis even in its current form can be used with Eclipse debugging and, as a plugin, this relationship can be made even stronger.

The second one is VisualVM. VisualVM is a debugging and performance analysis tool that already supports BTrace. In fact it uses BTrace to connect to applications. If TraVis was used as a plugin inside VisualVM it would remove the need for implementation of most statistical information about the code as VisualVM provides a lot of it. This information can be used for better graph trace representation in playback mode, where data like time spent in methods, number of method calls and memory or CPU usage can be visualised in the graph.

In combination with the above systems, and probably many others, TraVis as well as the host systems would become much more powerful, future and functionality rich and surely be greatly appreciated by many programmers.

8.3 Final Remarks

At the very beginning of this project many people were very sceptical about the possible outcomes and whether it was actually achievable, as there must have been a good reason why this sort of tool had never been created by anyone for any of the programming languages. Quickly though it was realised that it not only is achievable but has got huge potential that can be realised in the future.

The main thing that was proven by using TraVis is that when appropriate visualisation is provided for the system and not much overhead is induced by monitoring it, the money and time saving potential is immense. If 32% of product's total lifecycle [20, 10] is software understanding than saving even 1% or 2% can in many cases mean millions of pounds. Further savings can be achieved by enabling remote monitoring of processes in TraVis. It is supported by BTrace, so enabling it in TraVis should be trivial.

TraVis can be used for various tasks. Localising the cause of failure, bottlenecks, features and functionality were demonstrated in the evaluation in Chapter 7, but the list does not end there. One of the main design goals of TraVis was to enable users to use it on any system without having the slightest bit of knowledge about

the system. And this goal was very much achieved. Normally when debugging or logging systems are used, it is absolutely necessary to know where to place a breakpoint or a log and it can be hit-and-miss. With TraVis this dependency is completely removed and this is precisely the reason why the participants performed so well in the evaluation of such a large system, in such a short period of time. In addition, TraVis can be used for localising dead code, understanding dependencies between multiple processes at once i.e. client-server, and if it used BTrace ability to capture method parameters, it can be used for live debugging without any interrupts to the running system. TraVis uses BTrace so it is inherently safe to use on running processes [1] and this can greatly benefit systems deployed into production that should not be interrupted.

Since BTrace is based on DTrace and TraVis' dependency on BTrace is minimal, the techniques used in this project for the Java programming language can be used for all the languages supported by DTrace; these currently include [5]:

- C
- C++
- Objective-C
- Java
- JavaScript
- Perl
- PHP
- Python
- Ruby
- Shell
- Tcl

The size of the above list will likely increase over time. Supporting all of these languages in one tool would certainly not be a trivial thing to do, but definitely is feasible.

There are almost endless ways in which TraVis can be expanded. Unfortunately, due to the limited amount of time, it was possible to implement only as much as has been. In the future all effort will be made to expand it further and promote this project as "it's too good to remain a 4th Year Project" - as was said by one of the evaluation participants.

Appendix A

References

[1] Kannan Balasubramanian A. Sundararajan. Btrace java platform observability by bytecode instrumentation. Technical report, Sun Microsystems, 2008.

[2] D. C Arnold, D. H Ahn, B. R.de Supinski, G Lee, B. P Miller, and M Schulz. Stack trace analysis for large scale debugging. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 07)*, Long Beach, California, 2007. IEEE.

[3] Tiller Beauchamp and David Weston. DTrace: The reverse engineer's unexpected swiss army knife. In *Blackhat Europe*, 2008.

[4] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[5] Jim Mauro Brendan Gregg. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall, 1st edition, 2011.

[6] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[7] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE Inter-*

national Conference on Program Comprehension, pages 49–58, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81:2252–2268, December 2008.

[9] Robert DeLine, Gina Venolia, and Kael Rowan. Software development with code maps. *Queue*, 8:10:10–10:18, 2010.

[10] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[11] K. Erdös and H. M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 98–, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Facebook. Facebook factsheet. `https://www.facebook.com/press/info.php?factsheet`, March 2011.

[13] Ian Ferguson. *Computer Graphics via Java*. Ab-libris, 2002.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12:741–748, September 2006.

[17] Danny Holten, Bas Cornelissen, and Jarke J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. *Visualizing Software for Understanding and Analysis, International Workshop on*, 0:47–54, 2007.

[18] Randall Hyde. The fallacy of premature optimization. *Ubiquity*, 2009, February 2009.

[19] Magne Jørgensen. An empirical study of software maintenance tasks. *Journal of Software Maintenance*, 7:27–48, January 1995.

[20] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems*. Springer Verlag, 2010.

[21] Sun Microsystems. Dynamic tracing support in the java hotspot virtual machine, 2005.

[22] James Roberts and Craig Zilles. TraceVis: an execution trace visualization tool. In *Workshop on Modeling, Benchmarking and Simulation*, 2005.

[23] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[24] Alexandru Telea, Lucian Voinea, and Hans Sassenburg. Visual tools for software architecture understanding: A stakeholder perspective. *IEEE Software*, 27:46–53, 2010.

Appendix B

Detailed Specification and Design

This appendix includes some of the most challenging aspects associated with the development of TraVis. These examples are often accompanied by the source code representing a solution. Due to the size of TraVis and complexity of the project, lots of challenges are omitted. These omissions particularly apply to extremely carefully chosen data structure, storage formats, readers/writers.

B.1 travis.model Package

Probably the most important part of `travis.model.script` package is the use of regular expressions. As it has been mentioned a number of times throughout the report the performance was an incredibly important part of development process of TraVis. Regular expressions operate on byte level, thus are offering much better performance and control over a character sequence. Java build in string `String` methods offer limited support for regular expressions and in some cases incur additional performance penalties for example, `String.matches(String regex)` always results in `Pattern` compilation and new `Matcher` creation. In TraVis these sort of performance issues are removed.

`travis.model.attach.Playback` is by far the most complex class in the entire `model`. This class is a timer, player and parser. It is entirely thread safe and can change its state/mode at runtime. A method that is responsible for reading traces from a file is sensitive to speed, selected traces range and mode (package, class, method), all of which can be changed at any point in time by the user. Different modes means playback at different granularity of information. For example `PACKAGE` mode consumes traces from a file until the control from reaches a different

package. Because all the traces saved in the file are referring to methods alone, the logic for meeting the above condition gets increasingly complex from `METHOD` mode to `CLASS` mode to `PACKAGE` mode.

B.2 travis.view Package

There are a few classes in `travis.view` package that are sophisticated, as a whole, or have some sophisticated methods. Not surprisingly the most of these classes can be seen in `travis.view.project` package.

The first one is `travis.view.project.tree` package. It is where the main structure of a Java project is represented and from where it can be manipulated. This package takes direct role in BTrace script generation, script playback, main graph and edges creation. It is all due to the fact that the tree nodes can be arbitrarily selected by the user at any point of the execution, even while connections to processes are established.

The most ingenious method in this package is the one responsible for efficiently finding a `ProjectTreeNode` given `ProjectTreeNode` (root) and `StructMethod` (method for which the root is searched). `ProjectTreeNode` extends Java's `DefaultMutableTreeNode`, which can have arbitrary number of children and only one parent. The only supported way of searching `DefaultMutableTreeNode` is by getting a breadth or depth first iteration and searching all elements, and this is really inefficient.

`findMethod(ProjectTreeNode root, StructMethod method)` takes advantage of binary search and the fact that `StructMethod method` can always, easily find its path to the root. The root component of `StructMethod method` is identical to a component encapsulated in `ProjectTreeNode root`, and the path from the root to target `StructMethod method` must be made of the same `StructComponent`s. The algorithm works by preserving the path from `StructMethod method` to `StructComponent root` and then restoring it, using binary search, from `ProjectTreeNode root` to `ProjectTreeNode` which holds the desired method. The algorithm is thread safe and is listed below.

```
public static ProjectTreeNode findMethod(ProjectTreeNode root,
      StructMethod method) {
   if (root.getUserObject().equals(method))
      return root;

   LinkedList<StructComponent> parents = new LinkedList<StructComponent>();
```

```
   StructComponent temp = method;
   while ((temp = temp.getParent()) != null) {
      parents.addFirst(temp);
   }
   parents.poll(); // Poll default.package
   if (parents.size() == 0)
      return null;
   parents.add(method);

   ProjectTreeNode level = root;
   for (StructComponent comp : parents) {
      @SuppressWarnings("unchecked")
      Vector<ProjectTreeNode> rootChildren = level.children;
      if (rootChildren != null) {
         ProjectTreeNode[] nodes;
         synchronized (rootChildren) {
            nodes = new ProjectTreeNode[rootChildren.size()];
            rootChildren.toArray(nodes);
         }
         level = binarySearch(nodes, comp, 0, nodes.length - 1);
         if (level == null) {
            return null;
         }
      }
   }
   return level;
}

private static ProjectTreeNode binarySearch(ProjectTreeNode[] nodes,
      StructComponent value, int low, int high) {
   if (high < low)
      return null;
   int mid = low + (high - low) / 2;
   if (nodes[mid].getUserObject().compareTo(value) > 0)
      return binarySearch(nodes, value, low, mid - 1);
   else if (nodes[mid].getUserObject().compareTo(value) < 0)
      return binarySearch(nodes, value, mid + 1, high);
   else
      return nodes[mid];
}
```

The second, and by far the most complex package is `travis.view.project.graph`. It is responsible for the creation of main graph and connections.

The `TreeRepresentation` class is responsible for creation of the main graph. It is doing so by fetching selected checking if the selection of the tree nodes has changed, if so it creates a `ComponentData` tree, which is an ultimate wrapper with all the data needed for graph and connection creation. `ComponentData` is a bidirectional project tree, which during its creation filters out any nodes that should be excluded from the graph due to selected settings, i.e. hide methods.

The next stage is drawing the main graph from methods outwards (towards classes and packages). This process is like inversion-of-control since here children are responsible for creation of their parents. `LabelledArcRectangles` are used for the creation of the graph. The data needed for drawing the subsequent elements is stored by passing it recursively and iteratively to the next levels. Throughout this process all user defined settings (layers height, gaps, hidden layers, etc.) are taken into account and accurate representation of the graph is created.

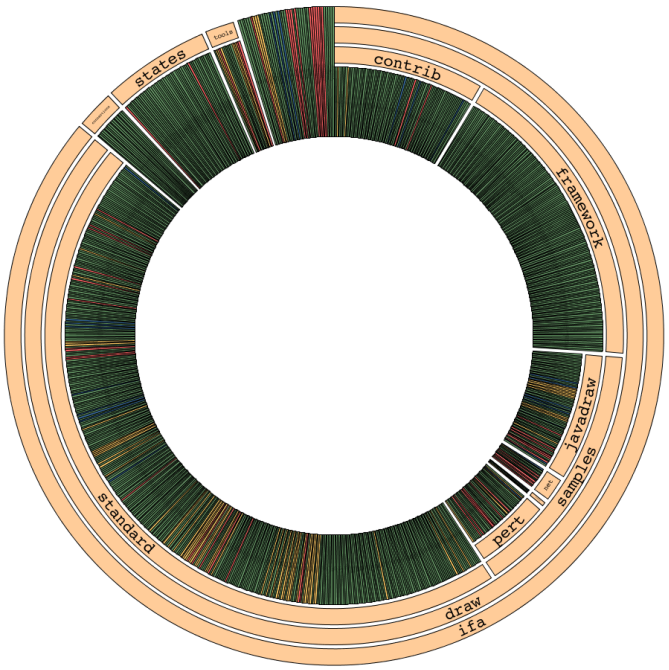
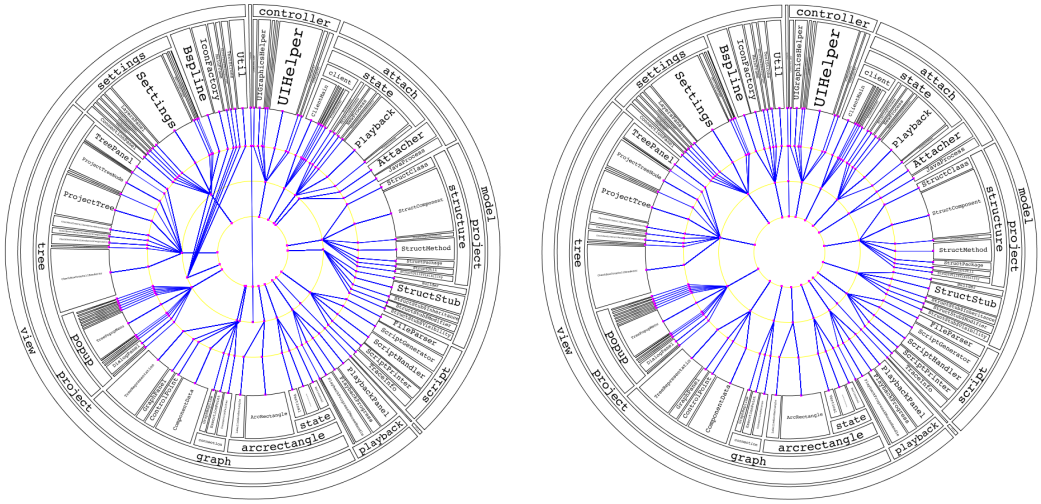


Figure B.1: Graph generated with hidden class layer and varying package depth.

Different layers have different characteristics like, packages being made of other packages. This adds to complexity of this class, as maths and logic employed to deal with the graph works for all the possible scenarios where different heights, widths and many other variables are set. Figure B.1 shows a customised graph.

The final step of the main graph creation is calculating an appropriate number of inner circles that are used for inner graph/layout creation. The inner layout is a radial tree layout which mirrors the outer graph. The inner layout is hidden and its structure is used as control points to guide the adjacency edges. One of the great challenges related to inner layout was the problem illustrated in Figure B.2a where the control points of some packages are pointing to the centre of their immediate parent. This was solved by dividing package are into two groups. First one was for all other packages and the second one was for classes alone. The result can be seen in Figure B.2b



(a) Problem **(b)** Solution

Figure B.2: Inner layout problem caused by various packages sharing a common parent.

Another great challenge was to design method that would draw traces/edges on the graph. The main difficulty was the fact that all methods could "randomly" call other methods or return. As a result of this it was necessary to create a method that iterates through the traces while maintaining a history to entire call trace, in case it is needed to return to a caller method. Additional complexity was added by a need of creating a path between arbitrary two method and retaining it, so it can be drawn in a later stage.

The above criteria was met by a single method. It keeps track of the history via recursion and to iterate over elements it uses iterator. Iterator is for `LinkedBlockingDeque` which is thread safe and makes the entire process thread safe. The method's behaviour is different for elements with depth 0 (first elements), since in this case the method should not terminate until the queue is empty. The method also keeps track of different threads. A body of this method is listed below. It does not include a code related to `ExecutionPoint`.

```
public void createConnections(ControlPoint cpStart,
      TraceInfo previousTrace, Iterator<TraceInfo> it,
      ConnectionData data, boolean isFirst) {
   for (; it.hasNext();) {
      TraceInfo trace = it.next();
      ComponentData cd = treeRep.getMethods()[trace.getMethodId()];

      if (cd == null) // only when not selected / visible
         continue;
      if (isFirst) {
         cpStart = null;
         previousTrace = null;
      }

      if (trace.isReturnCall()) {
         if (cpStart == null) {
            createConnections(null, null, it, data, false);
            continue;
         } else
            return;
      } else {
         ControlPoint cpEnd = cd.getControlPoint();
         if (cpStart == null) {
            cpStart = cpEnd;
            previousTrace = trace;
            createConnections(cpStart, trace, it, data, false);
            continue;
         } else {
            if (previousTrace.getThreadId() != trace.getThreadId())
               continue;
            Point[] path = cpStart.getPathTo(cpEnd);
            GraphBspline spline = new GraphBspline(previousTrace,
                  trace, path);
            data.addSpline(spline);
            createConnections(cpEnd, trace, it, data, false);
            continue;
         }
      }
   }
}
```

Appendix C

Detailed Test Strategy and Test Cases

Since TraVis is a front end application and it is close to impossible to determine what input will be captured (see section C.5), the main test strategy was to manually examine individual components and ensure they do not fail. A small set of test cases is provided below. All of the tests passed.

C.1 Graph

- Open, parse and compare with Eclipse a project made of classes alone (no packages).
- Open, parse and compare with Eclipse a large project.
- Fail to open incorrect binary directory.
- Create and rotate a graph for the project without packages.
- Create and rotate a graph for the project with some classes located in *default package* and others inside children packages.
- Change height of packages, classes and methods.
- Do not let packages, classes and methods fill in the centre of the graph - always have empty circle in the centre for connections.
- Change packages, classes and layers gaps.
- The minimum angle extent of a graph element is 0 - cannot grow in negative extent even if gap size demands it to shrink further.

- Repeat the above points with rotation.
- Hide individual groups of methods (public, private protected, default).
- Hide entire method layer.
- Hide individual groups of classes (interfaces, enums, abstract and ordinary classes).
- Hide entire class layer.
- Hide individual package layers.
- Hide entire package layers.
- Retain layers proportions when hiding individual the above layers/groups.
- Disable visibility groups' checkboxes if no further checkboxes can be unchecked.
- Repeat the above with rotation.
- Change colours of the graph.
- Resize the program and all its elements proportionally when window size changed.
- Generate inner layout of the graph.

C.2 Connection

- Generate BTrace script.
- Compile BTrace script.
- Connect to an existing process.
- Connect to a new process.
- Connect to a new process with various options and arguments specified.
- Capture standard output and error output from the newly created process.
- Capture traces from connected processes.
- Successfully filter and pass on traces to the observers.
- Draw Cubic B-Splines on the graph in right order from correct source to correct destination.

- Change splines bundling strength while connected and disconnected.
- Change minimum depth.
- Change cached traces size.
- Change max drawn traces.
- Switch between unique traces to latest traces.
- Change node selection on the project tree while connected.
- Display tooltips in various modes.
- Connect to multiple processes simultaneously.
- Kill individual/all connections.
- Reset drawn traces.
- Edit the project tree by adding and removing various elements.

C.3 Playback

- Save captured traces.
- Open captured traces and restore the tree with correct nodes selected.
- Open incorrect file.
- Create trace playback graph.
- Playback: play, pause, stop, speed change.
- Tooltip in playback when 3 or less traces are drawn per second.
- Playback in package, class and method modes.
- Select, playback and save subtrace.

C.4 Overview

A huge number of the tests listed above are interconnected and were performed in multiple combinations. Tests from all of the above sections were mixed together and performed both statically and dynamically (while connected or in playback). Not a singe of the above tests failed under correct configuration. The only failures

noted was due to memory or space limitations of TraVis, Java VM, compiler or external libraries (BTrace in particular).

A very detailed testing was performed by analysing captured traces line by line, and comparing them with the source code and the program execution flow (presented below). These tasks were repeated number of times in playback mode to ensure that the created graph representation is accurate.

C.5 Example of a Test Case

This example of a program, among the others, was used to determine what data is captured by TraVis and its correctnes.

```
public class AppMain {
        public static void main(String[] args) {
                Printer myPrinter = new Printer();
                myPrinter.printHelloTwice();
        }
}

public class Printer {
        public void printHelloTwice() {
                HelloPrinter helloPrinter = new HelloPrinter();
                for (int i = 0; i < 2; i++) {
                        helloPrinter.printStatement();
                }
        }
}

public class HelloPrinter {
        public void printStatement() {
                System.out.println("Hello World");
        }
}
```

Data captured by TraVis (method ID, nano time of a call, thread ID) with addition of a # followed by a comment:

```
1 1301415792666263000 1 # ENTER AppMain main ([Ljava.lang.String;)V
4 1301415792778303000 1 # ENTER Printer <init>()V
-4 1301415792778402000 1 # RETURN Printer <init>()V
```

```
5 1301415792778430000 1 # ENTER Printer printHelloTwice ()V
2 1301415792780844000 1 # ENTER HelloPrinter <init>()V
-2 1301415792780894000 1 # RETURN HelloPrinter <init>()V
3 1301415792780921000 1 # ENTER HelloPrinter printStatement ()V
-3 1301415792784536000 1 # RETURN HelloPrinter printStatement ()V
3 1301415792784956000 1 # ENTER HelloPrinter printStatement ()V
-3 1301415792785758000 1 # RETURN HelloPrinter printStatement ()V
-5 1301415792786028000 1 # RETURN Printer printHelloTwice ()V
-1 1301415792786276000 1 # RETURN AppMain main ([Ljava.lang.String;)V
```

Appendix D

Initial Project Specification and Plan

D.1 Overview

The project is about trying to understand the difficulties and the challenges that software engineers face when trying to adapt, maintain or build upon large existing systems. It is as well about helping them to overcome these problems. One of the approaches to tackle this problem is to try and gain a picture of the system's internal behaviour when it executes. This data is available from stack trace, but was found to be really difficult to work with, since human brain does not perform very well on large amounts of text.

One way of addressing this problem is by presenting the user with visualisation of the execution traces. It has been proven that the human brain can quickly and efficiently process large amount of visual information and easily spot patterns and anomalies. The hope is that users will gain the understanding of the system in a much shorter time, find it much easier to use and maybe even be able to spot some problems related to flow of execution or bottlenecks.

The main part of the project is about identifying a summarisation technique. This could be based on existing research done in this area or my findings from initial study. Extra care needs to be taken in researching the possible GUI tools and frameworks as they could greatly limit or enhance the functionality of the tool. One of the options is to make use of already available functionality from Eclipse IDE and integrate the tool into Eclipse IDE as a plugin.

D.2 Objectives

- Gain more rounded understanding of problems faced by software engineers exposed to large and medium scale systems.
- Produce novel solutions to overcome these problems in form of use cases.
- Choose stack trace summarising technique.
- Create framework that would make implementing solutions to the identified problems possible.
- Write software that would extract, from a file, the most needed information from the stack trace.
- Extract the information from stack trace of a running program.
- Deal with threads and polymorphism.
- Create GUI to let user view and manipulate displayed data.
- Perform an evaluation.

D.3 References and Related Work

Scientific publications that are related to my project are:

- "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System" by Abdelwahab Hamou-Lhadj and Timothy Lethbridge - `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01631120`
- "TraceVis: An Execution Trace Visualization Tool" by James Roberts and Craig Zilles - `http://www-mount.ece.umn.edu/~jjyi/MoBS/2005/program/roberts-tracevis.pdf`
- "Questions Programmers Ask During Software Evolution Tasks" by Jonathan Sillito, Gail C. Murphy and Kris De Volder - `http://www.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf`
- "Software Development with Code Maps" by Robert Deline, Gina Venolia and Kael Rowan - `http://queue.acm.org/detail.cfm?id=1831329`
- "Code Bubbles: Rethinging the User Interface Paradigm of Integrated Development Environments" by Andrew Bragdon, Steven P. Reiss, Robert

Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra and Joseph J. LaViola Jr. - `http://www.eecs.ucf.edu/isuelab/publications/pubs/ICSE0289.pdf`

D.4 Methodology

I personally believe that good specification, research and initial design of the project are the most important, and possibly most influential, aspects of development of this project. So I will attempt to create a well researched and thought through foundation for the project and leave coding until I believe that the design is right. Coding will be driven by use cases in an agile development style, with focus on test driven development, patterns and good practices.

D.5 Evaluation

The project could be evaluated in many different ways. The final decision about the evaluation could be strongly influenced by the research that has been done in the same or similar area.

Since the tool will be written to support user understanding and interaction with software projects, the evaluation would need to include a study of a small group of software developers with various experience on a mid scale software. It might include a questionnaire ranking ease of work with the tool and the usefulness of it. Could also observe them using it or use think-aloud protocol.

D.6 Project Plan

I plan to keep a log of the task I achieve, the problems I come across and novel solutions I develop. This should make writing Project Progress Report and Project Report much easier.
Monday 25 October Familiarise myself with the research that has been done in this area
Friday 29 October Perform a small study asking experienced software developers what information they look for in big projects
Wednesday 3 November Look into possible options for GUI representation
Monday 22 November Design use cases, class diagram, system specification and

architecture
Friday 3 December at 12:00 Project Poster due
Wednesday 8 December Project Poster Day
Friday 7 January Software able to extract and store stack trace from running program
Friday 28 January at 16:00 Project Progress Report due
Thursday 24 February Software able to display summarised stack trace
Tuesday 1 March User study and evaluation
Tuesday 1 March Start writing report
Friday 18 March End of additional functionality and software development
Wednesday 13 April Project Report due for Binding
Friday 15 April at 12:00 Project Report due
Wednesday 20 April Project Demo Day and Industrial Demo Day

D.7 Marking Scheme

A marking scheme that I have chosen in agreement with my project supervisor is Experimentation-based with Significant Software Development Project. We think that this scheme reflects best the characteristics of the project. It is not well known what is it exactly that developers would find most useful when analysing stack traces. The project is about creating a novel summarisation and visualisation for execution traces. To my current knowledge there is not much research done in this are and there are not many tools available that would help tackle this problem. Since there is not previous software addressing this problem it is necessary that I build the framework and the visualisation tools for stack trace summarisation from the very beginning.

D.8 Supervisor's Comment

Comment from Supervisor: Artur clearly has a sound understanding of the project requirements, has already developed some good ideas on how to approach it, and the proposed plan looks sensible and achievable. [Marc Roper 22/10/2010]

Appendix E

User Evaluation

E.1 Introduction

jEdit is a mature and well-designed programmer's text editor that has been in development for over 5 years. It is made of:

- 1099 classes including inner classes
- 483 classes excluding inner classes
- 158,210 lines of code including empty lines

As you can see it is a large system, but please do not be put off by the size of it.

Next you will be introduced to TraVis and then asked to perform 4 tasks related to jEdit. All the tasks will be accompanied by the list of TraVis features you might find useful in solving them. If you need extra help during solving them please do ask. It does not matter how well you perform in the tasks or whether you manage to do them all. For the purposes of evaluation please think aloud.

E.2 Exercises

1. Check if it is possible to search for *Regular Expressions* in *Search* menu, item *Find...*. Do you notice anything unusual? Could you find a cause of this issue? Useful features: execution point, execution point size, tooltip.

2. Find the cause of the performance dropping every 30 seconds or so. Useful features: execution point, execution point size, tooltip.

3. From *Tip of the Day* popup that can be accessed through Help menu delete *Show tips on startup*. Useful features: playback of repeated actions, save trace, save sub-trace, playback, section playback, change playback speed, tooltip.

4. Find out what class is responsible for handling actions like *Copy* or *Cut*. Useful features: playback of repeated actions, save trace, save sub-trace, playback, section playback, change playback speed, tooltip.

E.3 Evaluation Form

On the following scale, 1 meaning poor/difficult/no and 7 meaning excellent/easy/yes, please give your opinion on TraVis on the following:

	1	2	3	4	5	6	7
How difficult did you find the exercises?							
How easy did you find it to use?							
Was it of help in dealing with the exercises?							
Do you feel it saved you time?							
Did it help you learn about and understand jEdit?							
Would you use it for learning about medium/large systems?							
Would you use it for locating functionality of medium/large systems?							

Why would/would not you use it again:

Functionality that you found particularly useful:

Functionality that you missed:

Other comments:

Appendix F

User Guide

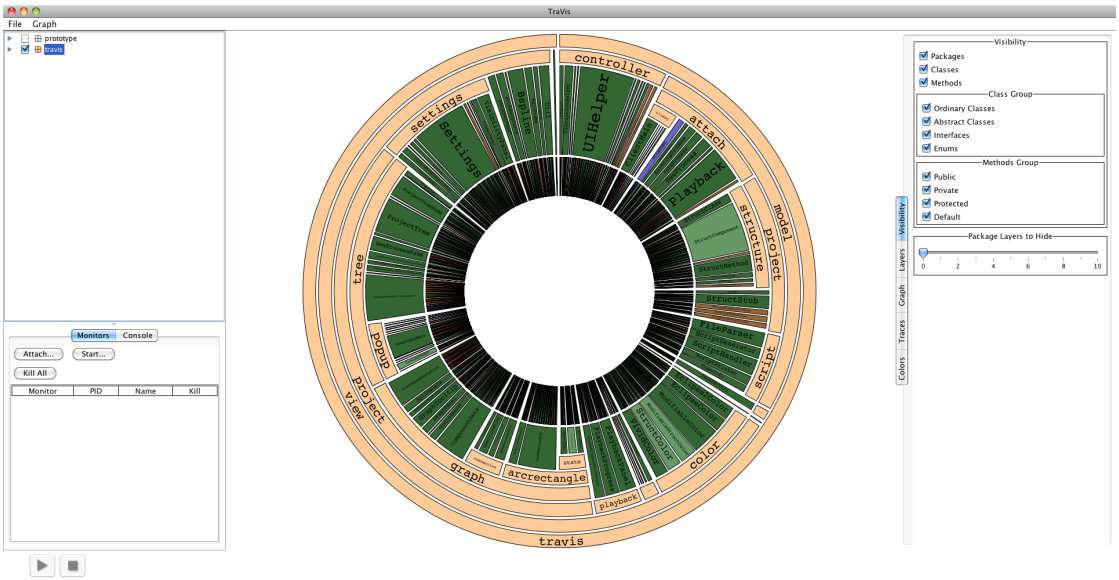


Figure F.1: TraVis window

TraVis Should be started via one of the executables provided. If for whatever reason this cannot be accomplished please ensure *dependencies* folder is added to the class path with all: *asm-all-3.3.jar, btrace-agent.jar, btrace-boot.jar, btrace-client.jar, miglayout-3.7.3.1.jar*. When TraVis is started from a *jar* file or from Eclipse please ensure that Java maximum heap space is increased. A minimum of 256 MB should be specified, where recommended is 1024 MB. This could be done by specifying `-Xmx1024m` as one of the Java parameters. Minimum heap space could be increased as well to 128 MB by specifying `-Xms128m` as another Java parameter.

On Windows platform it is necessary to add *tools.jar* to classpath and add *attach.dll* from TraVis folder or *<jdk_home>\jre\bin* into the default installed *jre bin* (in my case it is *C:\Program Files\Java\jre1.6.0_07\bin*). If issues are still occurring recompiling TraVis under Windows with *tolls.jar* included in the classpath should solve the problem.

F.1 Opening Project and Project Tree Building

There are two ways of building a project tree. The first one is automated and done by parsing the directory where binary files are located. And the second one allows building the tree node by node. The tree closely represents hierarchical package structure found in Eclipse IDE and all its nodes are stored in alphabetical order, packages taking precedence over classes and classes over methods.

Project tree is a key part of TraVis. It affects both the graph (Section F.2) and the monitors (Section F.3.1). Only selected nodes are drawn on the graph and are monitored. It is crucial to have selected all the nodes of interest before monitor establishes connection to the process as new ones, as selecting new ones after connection was established does not have an effect on the monitor. On the other hand deselecting nodes has an effect on drawn connections.

F.1.1 Automated Tree Building

By far the easiest way of building the project tree is by simply pointing the program to the right binary directories. This can be achieved through *File* → *Open Classpath...* or by pressing CTRL (Command on a Mac OS) + O. *Open Classspath...* will always result in creating a new tree.

Additional nodes can be added to the tree through *File* → *Attach Classpath...* or by pressing CTRL (Command on a Mac OS) + Shift + A. The way in which this differs from *Open Classspath...* is that it does not discard the already present tree.

F.1.2 Manual Tree Building

In some circumstances access to binary files might not be possible. This does not mean that the program cannot be monitored, as the project tree can be build node by node by right clicking on the project tree area or already existing node

and selecting appropriate item from a pop-up menu. Pop-up menu items depend on the are of the tree clicked.

Because the project tree is a binary representation of a project there are some limitations. All return types, methods and their parameters must be provided in binary form - instructions on how to determine binary forms are specified in *New Method...* dialog window seen in Figure F.2. Classes cannot contain another (inner) classes. Inner classes when compiled become a new class with default visibility, with a name equal to the parent class name followed by $ sign and inner class name.

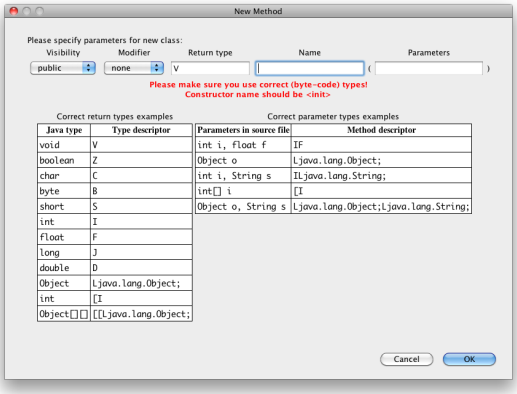


Figure F.2: New method dialog window

TraVis is insensitive to visibility modifiers. The option to specify these modifiers is provided for user reference.

F.1.3 Deleting Tree Nodes

If the project tree becomes too clattered or incorrect tree node was added it is possible to delete the node with all its children by right clicking on it and selecting *Delete* from pop-up menu. Multiple nodes can be deleted at once by selecting them in the same way multiple item selection is done on a operating system that TraVis is running on.

F.2 Graph

Every time a tree node is selected or deselected a graph is created. By default the graph contains all selected packages, classes and methods. Names of the compo-

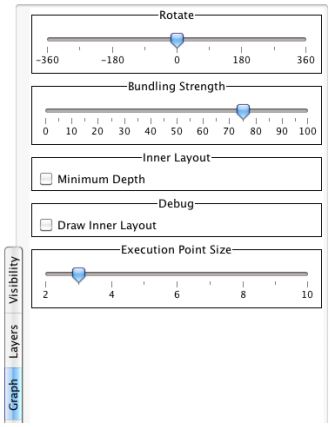


Figure F.3: Graph tab

nents are written on them, if the name is too small or not visible at all hovering over the component shows tooltip with component's name.

The graph can be rotated by adjusting the *Rotate* slider in *Graph* tab selected in Figure F.3. The available rotation angle is -360° to 360°.

F.2.1 Layers

As mentioned above the graph is made of three distinguishable layers. The space occupied by a layer is dependent on the number of selected methods inside this layer. The order in which the layers are drown represents the order in which they are represented in the project tree.

Package layer is the outermost layer of the graph. This is the only layer that may contain sublayers. If it contains any classes these are always drawn after the packages it may contain. Individual outer most layers of packages can be hidden by changing a value of *Package Layers to Hide* in *Visibility* tab selected in Figure F.4. One colour is shared for all the packages.

Class layer may contain four different class types: ordinary classes, abstract classes, interfaces and enums. A different colour can be assigned to each one of them.

Method layer may contain four different method types: public, private, protected and default. A different colour can be assigned to each one of them.

All colours can be changed through *Colors* tab selected in Figure F.5.

Through *Layers* tab visible in Figure F.6 it is possible to change height of packages, classes or methods layers and gap size between different packages, classes or

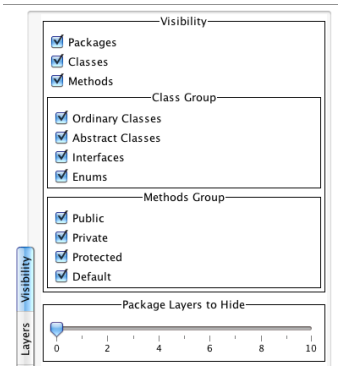


Figure F.4: Visibility tab

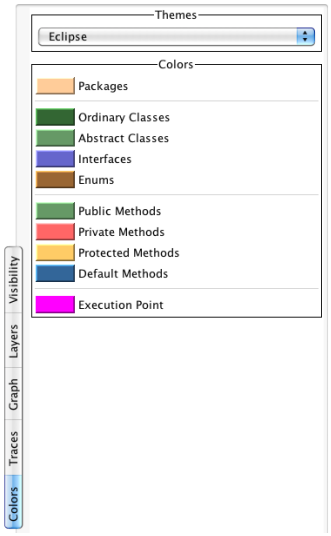


Figure F.5: Colours tab

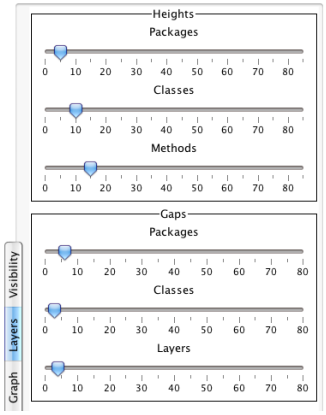


Figure F.6: Layers tab

individual layers.

Layers as well as their inner types can be hidden by toggling relevant checkboxes in *Visibility* tab displayed in Figure F.4. It is allowed to toggle checkboxes while being connected to a process or in playback mode. More information on connection mode can be found in Section F.3 and playback mode in Section F.4.

F.2.2 Connections/Traces

Once connection is established and trace is captured from a monitor connection between graph elements is drawn on the graph. Each connection is a coloured curve, that is green at one end and red on the other. A component at the green end is a caller and a component at the red end is a callee. This component might be a package, method or a class. Only connections for visible layers or components are drawn. More on layers can be found in Section F.2.1

When mouse is over a connection it is highlighted on the graph and a tooltip is displayed that includes details of the caller, callee and a thread id. Displayed details varies depending on whether the components are packages, classes or methods. If the mouse is over more than one connection all appropriate connections are highlighted and tooltip details are displayed for all of then in order of most recent calls.

Bundling strength of connections can be adjusted by changing a value of *Bundling Strength* slider located in *Graph* tab shown in Figure F.3. Value of 0 means bundling is not applied and as a result straight lines will be drawn, and value of a 100 means that each connection will be drawn through all control points.

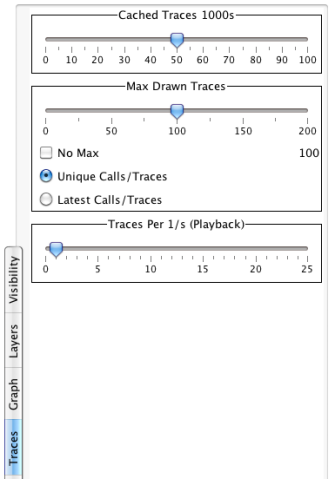


Figure F.7: Traces tab

Control points are parts of inner layout and can be made visible by selecting *Draw Inner Layout* checkbox in *Graph* tab. By default *Minimum Depth* is deselected as in some circumstance having *Minimum Depth* results in less readable connections.

Traces tab visible in Figure F.7 provides some control over drawn traces. Only connections from cached traces are drawn, so the more traces are cached the more can be drawn, this obviously impacts performance. The number of cached traces can be controlled by changing *Cached Traces 1000s* value. Sometimes it might be desired to change the number of drawn traces, this could be done be done in *Max Drawn Traces* section. In this section connection drawing strategy can be changed as well:

- *Unique Calls/Traces* means that only only one connection will be drawn between two components, i.e. if 10 calls were made from `methodA()` to `methodB()` they are treated as one call and only most recent is recorded.
- *Latest Calls/Traces* means quite the opposite to above. If 10 calls were made from `methodA()` to `methodB()` they are treated as separate calls.

Connections can be cleared through *Graph* → *Reset Drawn Traces* or by pressing CTRL (Command on Mac) + R.

It is allowed to change selected nodes in the project tree while connections are drawn. Connections between the nodes that are not selected in the tree are simply ignored. The same rule applies to the visible types of classes and methods, so disabling any checkboxes in *Visibility* tab, shown in Figure F.4, would result in either ignoring or including more connections.

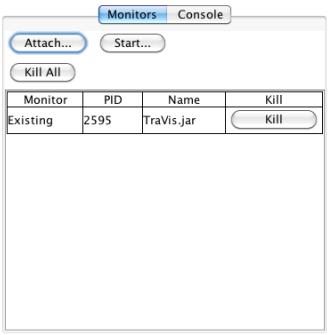


Figure F.8: Monitors tab

F.3 Establishing Connection

No trace collection can take place without establishing a connection to at least one process. The most important thing when creating a connection is currently selected nodes in the project tree. The script that is injected into a process is generated for **each selected node** in the tree. If a node was not selected at connection time it is not possible to include it in later stage without disconnecting and establishing a new connection.

Connecting to a process is safe. There are two different ways of creating a connection and monitoring a process. More than one connection can be created and active. The connections do not interfere with each other and can be killed or created at any time (other than in Playback Mode discussed in Section F.4). Being able to connect to more than one process at a time allows monitoring client and a server at the same time. Disconnecting from all processes at once can be done by pressing *Kill All* button in *Monitors* tab shown if Figure F.8.

F.3.1 Monitors

Monitors are responsible for script generation and establishing the connection to processes. The time taken to fully establish the connection with a process is dependent on the number of selected nodes and, if connecting to existing process, on how busy the process is.

It is possible to create a monitor for an existing processes as well as a new processes. Each individual monitor might be disabled by clicking *Kill* button in monitors table.

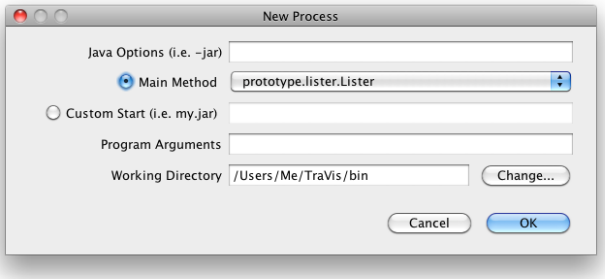


Figure F.9: New Process dialog window

Existing Process

Monitor connection to an existing process can be established by clicking *Attach...* button in *Monitors* tab, shown in Figure F.8, and selecting a process of interest from the list of running Java processes.

New Process

Monitor connection to a new process can be established by clicking *Attach...* button in *Monitors* tab and specifying all required options in dialog window presented in Figure F.9. The most important option in the dialog window is *Working Directory* as this is the path from which TraVis attempts to start the process.

F.4 Saving Traces and Playback Mode

Once the trace is collected it is possible to play it back by saving it. Traces can be saved through *File* → *Save As...* or by pressing CTRL (Command on Mac) + S. Saving trace can only happen when all monitors are closed, if this is not the case a dialog window is shown that upon confirmation closes all the monitors. Saved trace includes the tree project tree as well as the selected nodes, currently none of the settings are saved.

When opening a file project tree is rebuild, selected nodes are restored, project graph and trace graph, shown in Figure F.10 created.

Figure F.10: Trace Graph

F.4.1 Playback

Playback buttons are located next to a trace graph, seen in Figure F.10, these are only enabled in *Playback Mode*. Speed of the playback can be adjusted in *Traces* tab, shown in Figure F.7, by adjusting *Traces Per 1/s* slider. If there are less than 4 traces drawn per second current execution location is displayed next to execution point. The size of execution point can be adjusted in *Graph* tab, selected in Figure F.3.

The current position in file is displayed as vertical line on trace graph. This position can be manually adjusted by clicking anywhere on the graph. It is possible to select and playback a section of entire trace by clicking and dragging on the graph. The selection can be cleared by right clicking on the graph and selecting *Clear Selection*.

If the trace contains far too much information it is possible to save a sub-trace by selecting it in the manner displayed above and saving it through *File* → *Save Subtrace...* or by pressing CTRL (Command on a Mac OS) + T. This then can be opened. Saving a sub-trace might be used as a zoom-in function on a trace graph.

F.5 Console

For any newly created process the output is redirected to TraVis console, this can be found in *Console* tab. Redirect includes both `System.out` and `System.err`. *Note that if connected to existing process its output is not printed in the console.* Console also displays all the output, including error, from TraVis.

The console can be cleared by right clicking on it and selecting *Clear Console*.

Appendix G

Known Bugs

1. Deleting a selected node from a tree results in `ArrayIndexOutOfBoundsException` because `ProjectTree` is not removing the selected nodes from `CheckBoxTreeCellRenderer`.
2. Java `deleteOnExit()` method does not work properly on Windows and fails to delete temporary files created by TraVis.
3. When multiple items in the project tree are selected or deselected at once `stateChanged(ChangeEvent e)` in `ProjectTree` class is called once for every item. This causes graph to be repainted *n, n = number of items*, number of times instead of only once.

Appendix H

Program Listing

You should find a CD accompanying this report. The CD contains all of the electronic materials of this project.

The top level of this CD contains the following directories:

- `Executables` - contains executables for Mac OS X and Windows platforms as well as a runnable *jar* file and dependencies.
- `Report` - contains a copy of this report in *PDF* format.
- `SampleData` - contains a few sample trace files and binary directories for use with TraVis.
- `TraVis` - an Eclipse project folder containing all sources and dependencies of TraVis, it also includes early prototypes and tests.

The executables for Mac OS X and Windows automatically start TraVis with `-Xmx1024m` to increase Java heap space. If TraVis is started from a *jar* file it is advised to increase the amount of available heap space. In all cases it is necessary to have `dependencies` folder with all its files in the same directory as TraVis program.