FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Software Repository for Live Software Development

Gil Manuel Oliveira de Almeida Domingues

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ademar Aguiar

Co-Supervisor: Hugo Sereno

June 27, 2018

A Software Repository for Live Software Development

Gil Manuel Oliveira de Almeida Domingues

Mestrado Integrado em Engenharia Informática e Computação

June 27, 2018

Abstract

The concept of *Live Programming* exists since the genesis of some of the first programming environments such as Lisp machines, Smalltalk, among others. There are multiple characteristics that can be associated with it, specifically liveness, that is, the almost instantaneous nature of the feedback, as well as reflection, which allows for the software to modify its own implementation at runtime. For the new concept *Live Software Development* to be possible, the environment needs to allow the modification of the running software.

One way to facilitate *Live Software Development* would be to provide a visualization interface through which the developers could see and directly manipulate the running system. The visualization engine would have to receive detailed information about the system, in order to generate the visual representation of the system. As the required information is not only present in the static representation of the system - in source code files - but also generated dynamically at runtime, it becomes necessary to have an accessible software repository to store and allow the access to this information.

There have been many examples described in literature of software analysis techniques to improve software comprehension. These either analyze the structure of the software at several levels of abstraction or analyze the behavior of the software, ideally at runtime. There are also several works in literature comparing different database technologies, such as *SQL*, *graph based databases* and *time series databases*, how these differ in performance and which are the best fit for different types of data.

The goal of this thesis is threefold. The first goal is to determine which metrics best represent a running software system. Secondly, it is to implement software analysis tools which are most useful for the context of *Live Software Development*. The third goal is to implement a software repository which would be capable of storing the information obtained via these tools and provide access to that data, enabling its use for *Live Software Development* tools and environments.

The resulting framework from this thesis will then be used by a parallel dissertation which provides a virtual reality environment for visualizing this data.

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivation and Goals . . . 2
1.3 Project . . . 2
1.4 Dissertation Structure . . . 3

2 Literature Review **5**
2.1 Software Visualization . . . 5
2.1.1 Static Visualization . . . 6
2.1.2 Dynamic Visualization . . . 6
2.1.3 3D Visualization . . . 6
2.2 Software Structure . . . 7
2.2.1 Reverse Engineering . . . 8
2.2.2 Abstract Syntax Trees . . . 8
2.2.3 Design Patterns . . . 9
2.2.4 Tools . . . 10
2.3 Dynamic Analysis . . . 10
2.3.1 Instrumentation . . . 11
2.3.2 Virtual Machine Profiler . . . 12
2.3.3 Aspect Oriented Programming . . . 12
2.4 Database Technologies . . . 13
2.4.1 Real Time Data . . . 13
2.4.2 Structure Data . . . 14
2.5 Summary . . . 16

3 Proposed Solution **19**
3.1 Contextualization . . . 19
3.2 Goals and Research Questions . . . 20
3.3 Requirements . . . 20
3.3.1 Functional Requirements . . . 20
3.3.2 Non-functional Requirements . . . 22
3.4 Architecture . . . 23
3.5 Summary . . . 25

4 Static and Dynamic Software Analysis **27**
4.1 Problem . . . 27
4.2 Extracting Software Information . . . 28
4.2.1 Reverse Engineering . . . 28

4.2.2 Forward Engineering . . . 28
4.3 Assumptions . . . 28
4.4 Structural Analysis . . . 29
4.4.1 Structure . . . 30
4.4.2 Plug-in Generated Project Structure . . . 31
4.4.3 Communication . . . 32
4.4.4 Generating the Representation . . . 33
4.4.5 Live changes . . . 35
4.5 Runtime Analysis . . . 36
4.5.1 *AspectJ* . . . 36
4.5.2 Extracted data . . . 38
4.5.3 Communication . . . 39
4.6 Summary . . . 41

5 Repository for Software Metadata **43**
5.1 Model Structure . . . 44
5.2 Websockets . . . 44
5.2.1 Event information . . . 45
5.2.2 Structural changes . . . 46
5.3 Repository API . . . 46
5.3.1 Storing projects . . . 47
5.3.2 Reading projects . . . 47
5.4 Implementation details . . . 47
5.5 Summary . . . 48

6 Experiments & Results **49**
6.1 Case Studies . . . 49
6.1.1 *Maze* . . . 49
6.1.2 *JUnit4* . . . 50
6.2 Functional Requirements . . . 50
6.3 Performance Evaluation . . . 50
6.3.1 Structural Analysis . . . 51
6.3.2 Execution Analysis . . . 54
6.4 Visualization Engine Validation . . . 58
6.5 Summary . . . 58

7 Conclusions **61**
7.1 Main Contributions . . . 62
7.2 Future Work . . . 62

A Appendix A **65**
A.1 tese-repository . . . 65
A.1.1 Description . . . 65
A.1.2 Write Access . . . 65
A.1.3 Read Access . . . 65
A.1.4 Installation . . . 67

B Appendix B **69**
B.1 tese-static 69
B.1.1 Description 69
B.1.2 Installation 69
B.1.3 Usage 69

C Appendix C **71**
C.1 tese-runtime 71
C.1.1 Description 71
C.1.2 Installation 71
C.1.3 Usage 71

D Appendix D **73**
D.1 Projects 73
D.1.1 GET 73
D.1.2 POST 75
D.1.3 DELETE 76
D.2 Packages 76
D.2.1 GET 76
D.2.2 POST 77
D.2.3 DELETE 78
D.3 Classes 78
D.3.1 GET 78
D.3.2 POST 79
D.3.3 DELETE 80
D.4 Events 80
D.4.1 GET 80
D.4.2 POST 81
D.4.3 DELETE 82
D.5 WebSocket 82
D.5.1 Events 82
D.5.2 Structural Change Notification 82

References **85**

CONTENTS

List of Figures

1.1 Diagram of the idealized *Live Software Development Environment*. 3

2.1 Extract-Abstract-Present model for system structural information recovery. . . . 8
2.2 Generalization of a sequence diagram from a program's abstract syntax tree. . . . 9
2.3 Methods for software dynamic analysis. 11
2.4 Join point types and their definitions at runtime. 13
2.5 Software metamodel for object-oriented software. 15
2.6 Graph visualization generated by *Magnify* 16

3.1 Use case diagram for the framework. 21
3.2 Components of the software analysis framework. 23
3.3 Architecture of a full live software development system. 24

4.1 Screenshot of the eclipse plugin for structural analysis 29
4.2 Class diagram for the Java project model. 31
4.3 Execution analyzer aspect definition. 38
4.4 Sequence diagram describing the communication between the runtime analyzer and the repository during the former's lifetime. 40

5.1 Interactions between the software analysis tools and the repository. 44
5.2 Database structure diagram containing the projects' structure and runtime data. . 45

6.1 Timeline for the event set lifetime. 55
6.2 Scatterplot of the time values in relation to the event set size. 57
6.3 Timeline of the lifetime of an event. 57
6.4 Result of using the VR visualization tool on *JUnit* and *Maze* 59

LIST OF FIGURES

List of Tables

3.1 List of non-functional requirements 22

4.1 Relevant classes from the *Eclipse's Java Model* 30
4.2 Data extracted from each method call by the runtime analyzer. 39

6.1 Time monitoring points in the both the structural analysis tool and the repository, as well as their location in the structural analysis process. 52
6.2 Performance metrics related to the structural analysis. 52
6.3 Results of the first structural analysis experiment. 53
6.4 Results of the second structural analysis experiment. 53
6.5 Results of the third structural analysis experiment. 54
6.6 Location and description of each of the checkpoints for the performance analysis of the combined use of the runtime analyzer and repository. 55
6.7 Metrics generated from the checkpoint data regarding runtime analysis 56
6.8 Time intervals related with the lifetime of a single event. 56

LIST OF TABLES

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
IoT	Internet of Things
IDE	Integrated Development Environment
JVM	Java Virtual Machine
REST	Representational State Transfer
UML	Unified Modeling Language
VM	Virtual Machine
VR	Virtual Reality

Chapter 1

Introduction

The purpose of this first chapter is to introduce the context of this dissertation as well as the motivation behind it and its goals. The structure of the dissertation will also be described in detail.

1.1 Context

Software maintenance is a crucial part of the software development process. It is of paramount importance to follow the software performance after its deployment and take action to, for example, fix bugs or improve functionality. As it is such a core stage of the software development process, increasing the efficiency of this step is highly relevant.

A great part of the time spent maintaining a piece of software is used in the comprehension of the software in question. As such, it is of utmost importance finding ways of improving the efficiency of this process, decreasing the necessary time for a developer to understand the code one is about to modify. With the increasing complexity of software systems, tools that help improve the comprehension of a software system are a relevant investigation topic. One type of tools very suitable for this task of improving comprehension are visualization tools.

Visualization tools facilitate the understanding of the inner mechanics of a system by providing a tangible representation of the abstract concepts that compose the system. Both two dimensional and three dimensional software visualization tools have been built and explored in literature, the former more so than the latter, which leads us to believe there is still space for exploration in this field.

Liveness is closely related to software visualization tools. Tanimoto [Tan13] describes liveness as the ability to modify a running system, further detailing several degrees of liveness which vary in the quickness of the system's response to the change made to it. Ideally, the system will respond almost instantaneously to changes. Live programming is not a recent notion; LISP machines, the Smalltalk language and the Logo language are a few examples of uses of live programming in the earlier days of computing. In addition to software visualization, liveness is also closely related to

visual programming, which simply put provides a simpler, more intuitive interface to develop and modify software.

The novel concept of Live Software Development incorporates liveness. It provides the developer a means to visualize the system at various levels of abstraction during runtime which enhance software comprehension and allows the modification of software at runtime. It implies reflectiveness, that is, the software should be able to have knowledge of its own implementation and to change it during runtime.

In order to build a live software development environment, there are three main required components: tools to retrieve information from the system, a software information repository and a visualization engine. The tools used to collect information on the system should operate in two domains, both the static and the dynamic domain, to extract both structural information about the software (packages, classes, methods and others) and runtime information of the system (events, messages between components, method calls and others). The software repository should be able to efficiently receive and store this information, providing it to other tools which can use it. The visualization engine is responsible for converting this data to a tangible visual metaphor, so the users can observe the structure and behavior of the software during runtime.

1.2 Motivation and Goals

As mentioned in the previous section, increasing the efficiency in software comprehension during the maintenance phase of software development is necessary to reduce the time costs of this process. Live Software Development, being closely related to visualization tools and live programming, provides a way to achieve this.

The goal of this dissertation is to move closer to creating a live software development environment in two ways. The first one is the identification, adaptation and implementation of tools which extract meaningful information from software systems, both statically and dynamically. The second is to build a repository to collect the data gathered by these tools, store it and provide an interface so that the visualization engine can access it efficiently.

Therefore, the research questions this dissertation intends to answer can be described as such:

- What important information can be extracted from a software system to improve the developers' comprehension in the context of Live Software Development?
- How can that information be stored in a way that facilitates its fast storage, retrieval and processing?

1.3 Project

This dissertation is done in the context of a *Live Software Development* research project from the Software Engineering Research Group of FEUP. Concurrently with this dissertation there is another dissertation belonging to the same project which covers a complementary topic .

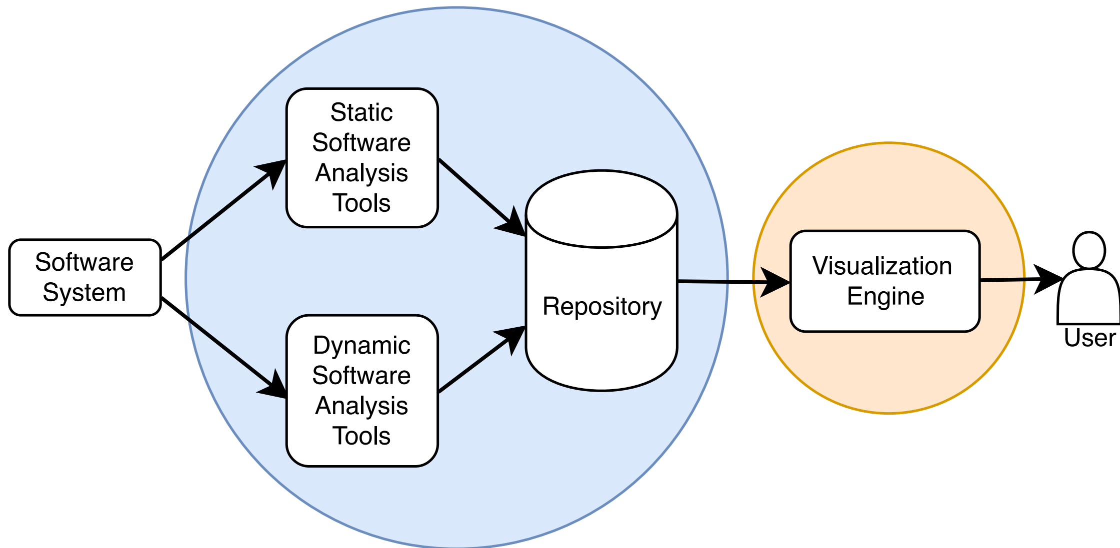


Figure 1.1: Diagram of the idealized *Live Software Development* environment. The blue highlighted part corresponds to the focus of this dissertation, while the orange highlighted part is the focus of the dissertation concurrent with this one.

This other dissertation has the goal of building a *Live Software Development* environment in virtual reality, and determining how useful it is to improve software comprehension. The combination of the work of these two dissertations will be a complete environment, from the data extraction to the data visualization.

A general view of the complete idealized system, as well as the components which are the focus of this dissertation can be seen in figure 1.1.

Through this dissertation, we intend to understand how a repository for *Live Software Development* should be developed by actively implementing and testing such a tool. In order to evaluate the proper functioning of the implemented tools, these will be tested on *JUnit*, as this is a commonly used Java system case study, and *Maze*, a smaller project built by the author of this thesis, providing a familiarity with it which will help in testing the implemented tools.

1.4 Dissertation Structure

This dissertation is composed of five chapters, including the present one. Chapter 2 will provide a literature review on the topics of statical analysis of software characteristics, dynamic analysis of software and database technologies.

In chapter 3 both the problem and the steps towards the proposed solution will be described in detail. Chapter 4 presents how the software information extraction tools were implemented to provide data to the repository and chapter 5 will describe how the repository prototype was build. Chapter 6 will discuss if and how the solution built matches with what was originally intended.

Finally, chapter 7 presents the conclusion and main contribution of the dissertation, discussing possible future work.

Chapter 2

Literature Review

2.1 Software Visualization 5

2.2 Software Structure 7

2.3 Dynamic Analysis 10

2.4 Database Technologies 13

2.5 Summary 16

This chapter of the dissertation will provide a description and short discussion on the current state of the relevant topics. It will present related work in each of the subjects, such as tools and methodologies correlated with the goals of this dissertation.

2.1 Software Visualization

Software visualization is an important tool to enhance comprehension. Software is inherently invisible, which does not help the task of understanding how a project functions. Visualization tools are necessary to associate a tangible representation to the code and the program execution. This is especially relevant in the maintenance, reverse engineering and re-engineering cases [Kos03].

The purpose of software visualization is not to create beautiful representations of systems but rather to create informative and useful aids to the process of software perception, especially for more complex systems [TC09].

A survey made by Bassil et al. determines functional aspects of software visualization tools and analyses how useful each of these functional aspects is perceived as useful for the software development process [BK].

2.1.1 Static Visualization

Static information is at the core of software comprehension. Even if a tool intends to provide visualization of execution traces, without information on the static representation of the software relating traces to specific parts of the system is not as intuitive.

Bassil et al. shows evidence that the most commonly used visualization methods are representations based on graphs. In fact, there are plenty of examples in literature [SJ15, BTDS13] which output graphs to represent the relationships between levels of a system [BK].

These static representations are not limited to the data a code crawler can extract from source code. Magnify [BTDS13] also attempts to extract metrics from components and present them in the visualization, such as how important an artifact is and the quality of the code.

DocTool [SJ15] simplifies the automatic analysis by only extracting basic structural information from the software in the form of a graph. To compensate for the lack of information resulting from this simple analysis, it also provides a web interface for the user to input information on higher level artifacts.

CodeCrawler [LD03, LDGP05] is a visualization tool which provides the possibility of visualizing data retrieved from other reverse engineering tools, offering a visual encoding that allows five metrics to be represented per entity. The visualization is deciding by choosing the layout, the five metrics out of a defined list [LD03] and the entities for which to represent those metrics.

2.1.2 Dynamic Visualization

Static representation of the software is necessary for software visualization, but it is not enough to fully understand the system. Not observing the behavior of a system would be wasting a valuable source of information.

Jinsight is an example of a tool created for the purpose of visualizing program runtime data, specifically geared towards multi-threaded Java applications. It provides multiple views to increase the probability of the user being able to detect existing performance issues, unexpected behavior or bugs. The information used by this tool is extracted from generated by a profiling agent in a standard *JVM* [DPJM+02].

In Orso et al., a different approach is taken in the sense of data collection. Instead of focusing in a single execution trace, it works best for already deployed software by analyzing a collection of traces. To avoid cluttering the visualization two methods are used: the traces can be aggregated and filtered, and the tool provides multiple levels of visualization (statement level, file level and system level) [OJH03].

2.1.3 3D Visualization

While the most commons software visualization methods are two-dimensional representations, there is also a valid reasoning behind using three-dimensional representations. Three dimensions provides the possibility of building visualization metaphors that are much more familiar to us.

For example, Wettel et al. presents a representation of the architecture of software as a city, where the user can freely move around and observe and interact with the system [WLR11].

Teyseyre et al. [TC09] discusses the use of 3D software representations and how they have been approach up until this point. Representations have mostly been in one of two ways: abstract visual or real world representations. Abstract visual representations are graphs, trees and other abstract geometric shapes, while an example of real world representations is a city metaphor.

It also describes three different types of user interactions with the visualization: directly manipulating the objects, user navigation so the user can view the system from multiple perspectives and system control through widgets.

In Okamura et al., a 3D tool for execution analysis is presented. Data is sourced from a combination of data monitoring, control flow monitoring and component testing. As the data collected is all time based, the tool builds a 3D animated scene, providing the user with navigation functions such as suspend, replay and rewind. For the purpose of debugging, the user is able to mark data and rules so it is highlighted throughout the animation and thus more easily identifiable [OST04].

There have also been cases where combined static and dynamic analysis is the source of information for the visualization. In Greevy et al., a combined static and dynamic data model merges information from both static and dynamic analysis, which is then displayed in multiple views. This is done in the context of feature centric reverse engineering, so the collected static data provides a single level of abstraction of the system [GLW06].

2.2 Software Structure

Source code is the software representation most familiar for a developer. It is how software is built and modified. However, it is not necessarily the best way to represent software when the goal is easier comprehension.

For that purpose, different and higher levels of abstraction are useful in order to increase the developers' understanding of the software, by elevating above the finer grained implementation details. *UML* is an example of a higher level representation of a system's structure and behavior [RJB04], being amongst the most popular for object-oriented systems.

To develop a higher level abstraction, it is required to obtain the existent structural information from the system. The focus of this section is to find how a software's structural information can be obtained.

Feijs et al. [FKO98] describes a model for analyzing architecture: the Extract-Abstract-Present model. Extraction consists on retrieving structural information from the system, abstraction is the derivation of new relationships between the components obtained in the earlier phase (that is, a further analysis of those components) and the presentation of that information through a graphical format. Figure 2.1 shows a diagram representation of this model.

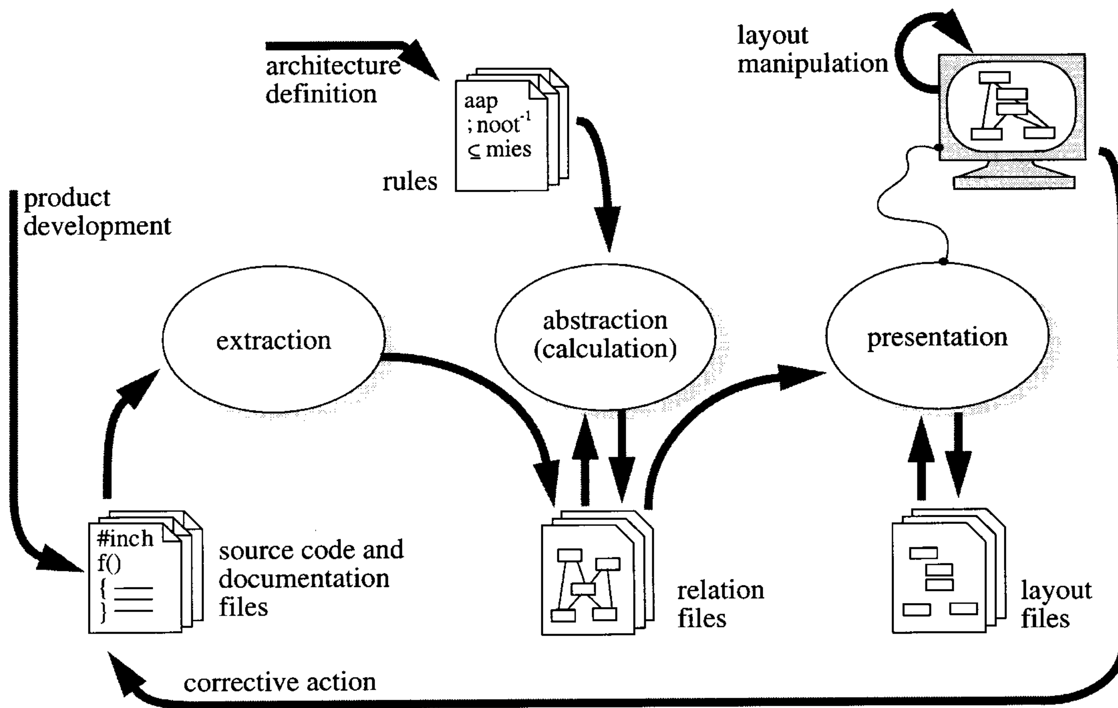


Figure 2.1: Diagram describing the process for the Extract-Abstract-Present model for system structural information recovery [FKO98].

2.2.1 Reverse Engineering

Reverse engineering can be described as the process through which design elements are recovered from the implementation. The abstraction resulting from reverse engineering is not a new or modified system. Rather, it is a process of examination [CC90].

The usefulness of this approach is demonstrated by literature on the subject. In Fauzi et al. [FHS16], reverse engineering is used to generate sequence diagrams that reflect a system's behavior, process that can be seen in figure 2.2. For this, the authors use a software's abstract syntax tree.

Although one may assume reverse engineering makes use solely of static representations, such as source code or byte code, this is not the case. There are instances in literature where static and dynamic analysis are combined. Guéhéneuc et al. [Gué04] demonstrates how a mixture of static and dynamic models allows for a more precise automatic generation of class diagrams.

Another kind of abstraction that is possible to retrieve from a system's implementation is design patterns. Shi et al. [SO06] describes *PINOT*, a tool to automatically detect design patterns from both the source code and the system's behavior.

2.2.2 Abstract Syntax Trees

Abstract syntax trees (*AST*) are one of the data structures used by compilers to create an intermediate representation of the software, thus becoming an interesting starting point for analyzing the

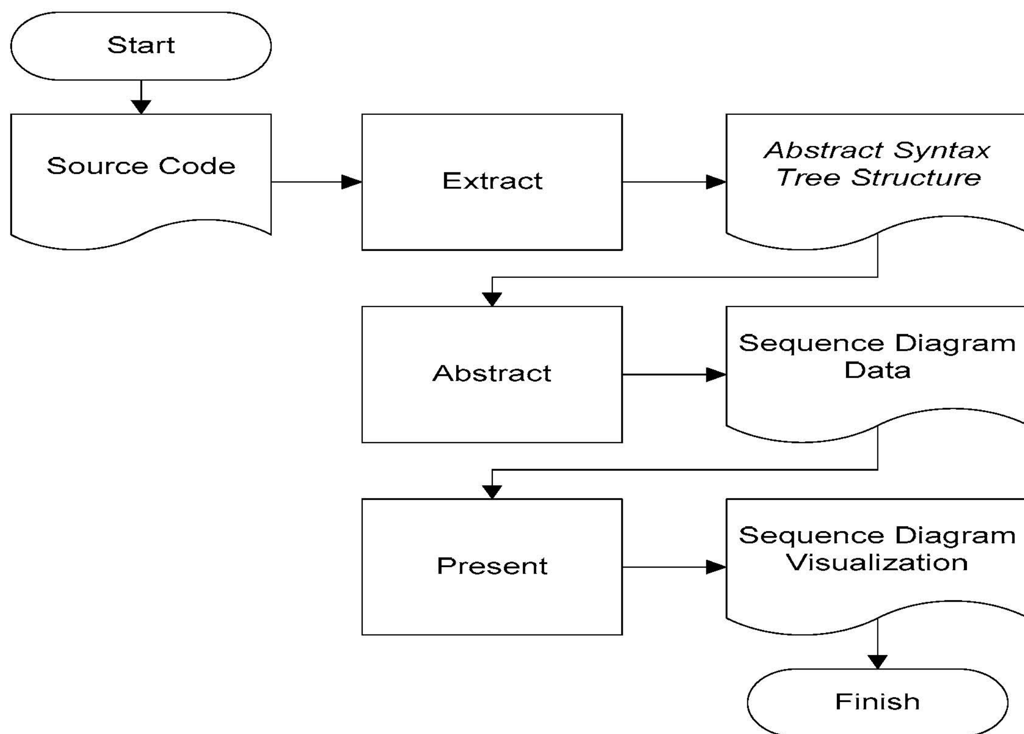


Figure 2.2: Diagram describing the generation of a sequence diagram from a program's abstract syntax tree [FHS16].

structure of a software system. It creates a structure from the input (source code) which ignores unnecessary syntactic details [Jon03].

The literature provides plenty of examples of the use of abstract syntax trees for code analysis. For instance, it has more than once been used to detect cloned code to prevent plagiarism [LB14, FCX13, TGHB13].

Baxter et al. [BYM+98] states that the reduction of duplicated code, which composes 5-10% of the code in large scale software, leads to reduced maintenance costs, and describes the application of a tool that uses abstract syntax trees to detect said duplicates. In this case, the *AST* provides a representation that is easier to compare, as opposed to lines of code, where changing variable names or inserting comments, for example, would increase the number of false negatives.

There have also been efforts in visualizing the evolution of a software project by analyzing the *AST* between commits, as opposed to the typical *file diffs* done by version control systems [FSWH16].

It is thus evident that this type of structure is incredibly relevant for the purpose of software structural analysis.

2.2.3 Design Patterns

When implementing a piece of software, design patterns provide a template for standard solutions known to work for certain problems. These patterns are not usually easy to perceive from the

source code, so developers rely on documentation to preserve the knowledge of the implemented patterns. However, documentation is commonly left behind when changing code in the software system during maintenance [FWG07].

Design pattern extraction from existing software artifacts becomes relevant to recover details from the design stage of the system.

In Nakayama et al. [NS14], an anchored *AST* is used to incrementally define patterns in code. Another method is used in Tsantalis et al., where class relationships mapped onto matrices and a similarity score is calculated to predefined matrices that represent patterns [TCSH06].

Antoniol et al. [AFC98] proposes an approach to recovering design patterns by extracting the software's representation in an intermediate language based on *UML* (in which there are already predefined pattern representations), extracting metrics from the software's classes and applying pattern recognition techniques to the results.

Another approach described in literature by Flores et al. [Flo06] involves the recovery of design patterns in frameworks by reverse engineering through multiple layers of high level abstraction. This is done by first recovering meta-patterns and then using these meta-patterns to recover design patterns, which will then be used to document the framework.

Finally, Shi et al. [SO06] claims to detect a majority of the design patterns defined in the *GoF* book [GHJV95] by reclassifying the patterns as either behavioral or structural, and making use of a mixture of static and dynamic analysis to identify the patterns.

2.2.4 Tools

For the purpose of structure analysis and reverse engineering, mature tools have already been developed and used. In this section an overview of some of these tools will be provided.

Reverse Engineering Environment

Rigi is a mature tool for research purposes, providing a complete environment for reverse engineering and analyzing large software systems. It includes parsers to extract information from the source code, a file-based central repository to aggregate the data, ways to analyze said data, a scripting language to automate these processes and its own graph visualization engine [KM10].

Structure Analysis

STAN is a tool aimed towards structural analysis of Java software. It uses the Java byte code instead of the actual source code. As an end result, it provides multiple views, such as dependency view or composition view, several established static metrics about the software and reports. It is available as a standalone app or integrates with the *Eclipse IDE* [STA09].

2.3 Dynamic Analysis

Obtaining a software system's structure is not sufficient to understand how it behaves. There are multiple sources of variability that cannot be taken into account during a static analysis. User

	Dynamic Analysis Technique			
	Instrumentation Based		VM Profiling Based	AOP Based
	Static	Dynamic		
Level of Abstraction	Instruction/Bytecode	Instruction/bytecode	Bytecode	Programming Language
Overhead	Runtime	Runtime	Runtime	Design and deployment
Implementation Complexity	Comparatively Low	High	High	Low
User Expertise	Low	High	Low	High
Re-compilation	Required	Not Required	Not Required	Required

Figure 2.3: Table with a brief characterization of each method for software dynamic analysis [GS15].

input, performance of shared resources and variable control flow paths all contribute to the fact that the source code does not predict the exact behavior during the execution of the program [GS15].

To compensate for this lack of information, the system should be observed during runtime. For example, logging is a very common practice in software development to record dynamic information of a program's execution [YPZ12].

Dynamic analysis can be implemented in three different ways. Gosain et al. describes the different approaches and tools associated. An overview can be seen in figure 2.3.

2.3.1 Instrumentation

A first approach would be to instrument the software by using a Java byte code rewriting library. This could be done at source level, by modifying the source code, at binary level, by rewriting compiled code, or at byte code level, by inserting new code either during compilation (statically) or at load time (dynamically).

Source-to-source transformation tools, originally used for the purpose of code generation, started being used for the purpose of maintenance and software evolution [BPM04]. This high level approach offers some other advantages, like the possibility of removing redundant source code and facilitating software comprehension [WY05].

Binary code transformation occurs after the compilation process. It can be done statically, using tools such as *EEL* [LS95] which provide a library for executable modification, or dynamically with tools like *MDL* [HNM+97], which perform an incremental instrumentation of the executable at runtime.

Finally, instrumenting byte code can be done with tools such as *Javassist*, which makes use of Java's reflective capabilities to provide a higher level *API* to instrument code [Chi00], or *BIT*,

which allows instrumentation at any point during byte code execution [LZ97]. In the case of *Javassist* this structural reflection is only possible before load time, being unable to compensate for dynamic loading of modified classes.

2.3.2 Virtual Machine Profiler

This method is dependent on the Virtual Machine used to run the compile code. In the case of the Java programming language, it would be the *JVM*.

By making use of the environment responsible for executing the code, it provides the possibility of having an in depth perspective of the behavior of the program. Tools like *JPDA*, whose original purpose is debugging, provide an interface (in this case *JMVTI*) through which one can apply profiling agents to extract information from the software running in the virtual machine [GS15].

One of the major advantages of this approach is that the analysis is independent of the software's code, avoiding complications which could stem from needing to merge the code responsible for the analysis with the code to be analyzed.

JMV-TI is capable of providing information to tools responsible for monitoring, profiling, debugging and other types of analysis [jvm].

2.3.3 Aspect Oriented Programming

Aspect oriented programming enables decoupling crosscutting concerns. It circumvents the need for instrumentation as the language already takes responsibility for that part of the implementation of the dynamic analysis.

This approach can be done in Java by using *AspectJ* [KHH+01]. *AspectJ* defines a series of concepts to establish how the results are produced: join points, pointcuts, advices and aspects.

Join points are the points of interest in the original source code, such as method calls, class definitions and others which are listed on figure 2.4. These are well defined points in the execution of a program. Figure 2.4 shows the join points *AspectJ* identifies.

Pointcuts are sets of join points which can be built using designators in order to provide matching of multiple join points at once.

Advices are the method-like mechanism which will be run at each join point of the specified pointcut. It provides execution *before*, *during* and *after* the join point.

Finally, aspects are the combination of all these components in a class like modular unit in order to specify the implementation of the crosscutting concern. Aspects can declare pointcuts and advices, as a class would be able to implement fields and methods [KHH+01, HH04].

Richters et al. uses aspect oriented programming to monitor a systems compliance with *UML* and *OCL* constraints [RG03]. In Gschwind et al., the usage of AspectJ allows the developed tool to obtain information about method calls and each argument used in said calls [GO03].

kind of join point	*points in the program execution at which...*
method call constructor call*	a method (or a constructor of a class) is called. Call join points are in the calling object, or in no object if the call is from a static method.
method call reception constructor call reception	an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method/constructor has been called.
method execution* constructor execution*	an individual method or constructor is invoked.
field get	a field of an object, class or interface is read.
field set	a field of an object or class is set.
exception handler execution*	an exception handler is invoked.
class initialization*	static initializers for a class, if any, are run.
object initialization*	when the dynamic initializers for a class, if any, are run during object creation.

Figure 2.4: Table containing the types of join point and the definitions of each join point at runtime [KHH+01].

2.4 Database Technologies

Developing a repository requires choosing which technology should be used for storing the data. There are several options from which to choose. File based repositories, relational databases, graph based databases, among others, are examples of databases that should be considered when making the design decision for what is one of the most important single points in the system.

The existing trade-offs should be analyzed, taking into consideration the goal of the system being developed. Volume of values inserted at once, volume of queries performed at once and whether it is necessary to enforce relationship constraints are some examples of requirements which affect the choice of database.

2.4.1 Real Time Data

Collecting and providing access to real time data in an efficient manner is one of the main challenges in database technologies. From infrastructure and system monitoring to the current trend of applying sensors for real world objects to report their status in real time, these applications require low latency for high volume data transfers, which continuously test the limits of the current state of database technologies.

Stonebraker et al. presents eight distinct requirements for storing and consuming real time data [ScZ05]. These requirements are:

Keep Data Moving: a system should process messages without having a costly write operation blocking the process. That is, ideally, messages should be processed *on-the-fly* by providing access to the data stream, avoiding a "*polling*" approach.

Query using SQL on streams: a high level query language with adequate stream-oriented capabilities should be supported, in order to provide the capability of finding messages of interest.

Handle Stream Imperfections: the system should be able to compensate for problems that typically occur in real time data streams, such as out-of-order, delayed or missing messages.

Generate Predictable Outcomes: the data processing method should be deterministic and thus provide a predictable outcome.

Integrate Stored and Streaming Data: a stream processing system should be able to efficiently store state, so that it is possible to combine past data with real time data.

Guarantee Data Safety and Availability: the system should have high availability and be able to ensure data integrity at all times.

Partition and Scale Applications Automatically: a real time data processing system should be able to split its workload between multiple processors to provide scalability. In the best case scenario, this split should occur automatically.

Process and Respond Instantaneously: the system should be able to match the speed of data retrieval, that is, it should process the data with minimal latency.

Establishing a parallelism to other applications, receiving data from system tracing and logging is akin to receiving real time sensor data, with the different of the inherent latency of the transmission of sensor data. In Veen et al., *SQL* and *NoSQL* databases are compared in terms of performance for high and low volume read and write operations [vdVvdWM12].

Traditional *SQL* databases are built using a fixed table structure and provide a query language mechanism to select data from these tables. *NoSQL* databases vary much largely in implementation. Two examples of *NoSQL* databases can be key value databases or time series databases. *SQL* databases are usually associated with more powerful query languages, while *NoSQL* databases are more usually associated with high performance read and write operations [HELD11].

The paper analyzes three databases: *PostgreSQL*, a relational database, *MongoDB*, a key-value database that stores its data in *JSON*-like files (*BSON*) and has a relatively powerful query language, and *Cassandra*, also a key-value store designed to cope with large volumes of data.

After analyzing the performance of these databases for the previously mentioned scenarios, the paper concludes that *Cassandra* is ideal for large scale critical sensor applications, *MongoDB* is a good fit for small to medium systems that require high write performance and *PostgreSQL* is the best choice when versatile query capabilities as well as reading performance are requirements [vdVvdWM12].

2.4.2 Structure Data

Software built using object-oriented programming languages has a well defined structure inherent to it. In a generic sense, all software artifacts created can be viewed as a vertex in a graph [DST11].

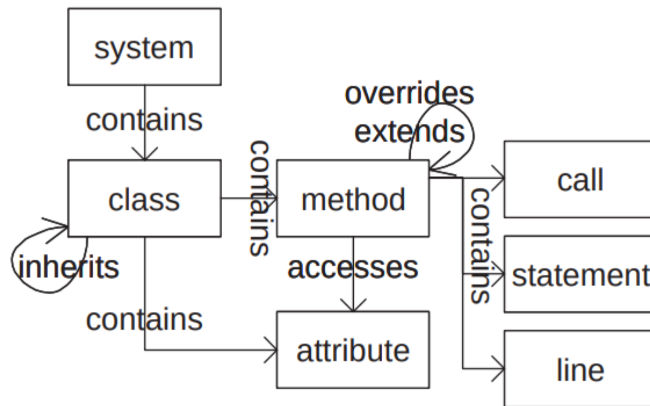


Figure 2.5: A software metamodel for object-oriented software [ML02].

Moving into source code level, one can see through the case of abstract syntax trees, the structure of the source code of a project closely resembles that of a graph.

Figure 2.5 shows an example of an object-oriented metamodel in graph form. *Mens et al.* built this metamodel and used it to produce low and high order metrics from the graph representation. Examples of low order metrics would be node count and path length, while high order metrics can be node count ratio and subsequent refinements [ML02].

As a result of this, many projects concerning software visualization and the software development project look towards using graph databases as the core of their implementations.

Magnify, is an attempt at creating a visualization system for software. It only provides a view of the source code and is limited to what it can extract from the code given to its parser. A graph database is used to store the structural information of a software in a language independent fashion, which is then used by a web interface to display that data. The nodes in this graph represent basic software components such as packages, classes and methods while the edges represent their structure and hierarchy. It uses *Tinkerpop*, which is provides an in-memory graph based database [BTDS13]. Figure 2.6 shows an example result of using this tool to visualize software structure.

Another tool for software project visualization, *DocTool*, uses a graph database along *json* files as its backbone. It splits its data collection in two: a server side and the client side. The server side corresponds to a crawler that extracts the basic structural information from the source code, such as methods and classes, while the client side serves the purpose of allowing the user to insert information about other concepts: entities, attributes, actions and pages.

The tool makes use of a code crawling *Eclipse plugin* to extract the information from the source code. It also provides a web interface for the user to input the client information and visualize the graph.

Each node in the graph database represents an element in the source code, such as a class or a method, while the edges between nodes represents relationships between these elements, such as *"class X implements method Y"*. The complementing *json* files serve as an intermediate step between the tools. Once a change is made to the data in the web client, for example, the

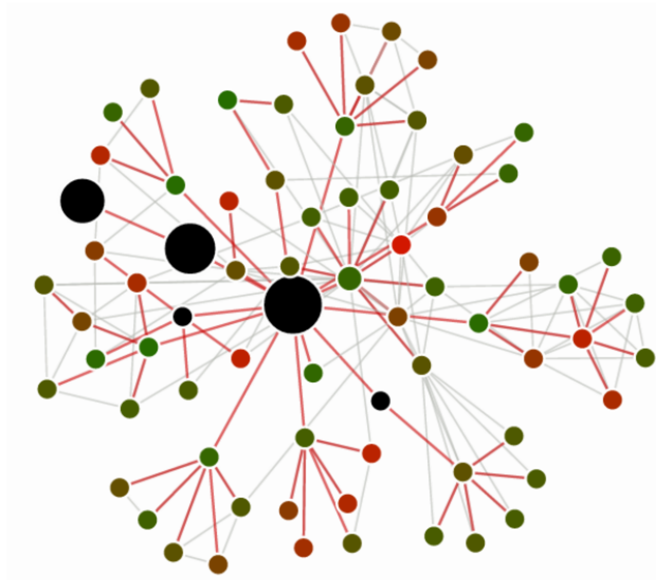


Figure 2.6: An example graph visualization generated by *Magnify* for Spring context 3.2.2 [BTDS13].

modification is first done in the *json* files and only then does it propagate to the graph database [SJ15].

Graph structures can also be stored in a relational database, albeit in a less intuitive fashion when compared to graph based databases [Cel17]. Perhaps for this reason most of the tools described in literature make use of graph databases.

2.5 Summary

Software visualization is a heavily explored area, especially in regards to 2D visualizations. While 3D visualizations have also been explored, it can be seen there is a lower focus on these. This could be due to the fact that three dimensional visualizations for non spatial data, such as software structure, is ill-advised.

An exception to this is real world metaphors, which is one of the more explored 3D visual representations of software. This requires generating meaningful spacial dimensions for the representation elements, making this representation adequate for three dimensions.

There is also not much literature in regards to 3D visualization of data resulting from software dynamic analysis, which leaves a gap in research ready to be explored.

The reverse engineering field for static analysis of software has plenty of tools described in literature, some more lightweight than others. For the context of this dissertation, the tool should be as decoupled as possible, so as to have a separation of responsibilities between components.

The abstract syntax tree in the context of Java provides an easy way to access structure from source code by abstracting minor syntactic details. This facilitates the retrieval of the software's structure.

As for dynamic analysis, the ideal approach would have a low overhead and should be as decoupled from the source code as possible, which should rule out source-to-source instrumentation. Some experimentation should be done between bytecode instrumentation, *VM* profiling and aspect oriented programing approaches to decide which is a better fit.

The database technology will be an impactful decision on this dissertation. Nonetheless, literature already shows us that *SQL* and *NoSQL* databases have points in favor and against.

Chapter 3

Proposed Solution

3.1 Contextualization . . . 19
3.2 Goals and Research Questions . . . 20
3.3 Requirements . . . 20
3.4 Architecture . . . 23
3.5 Summary . . . 25

In this chapter, the problem will be explored. It will start with the contextualization of the dissertation on the overarching project which contains another dissertation which was developed in parallel. The goals and research questions description follows that section, leading up to the requirements where we discuss what the developed framework should accomplish and how it should do so.

The section that follows contains the architecture of the frameworks and how its components connect with each other, as well as discussing how this framework could be connected with a specific external tool to create a live software visualization environment.

Finally, the case studies which will be used to help develop, test and verify the correct functionality of the framework will be presented.

Further implementation details will be left for the two following chapters, chapter 4 and chapter 5.

3.1 Contextualization

This project was created in the context of an investigation group on *live software engineering*, started in the present school year, at Faculdade de Engenharia da Universidade do Porto.

This dissertation was realized in parallel with another one, named "Towards a Live Software Development Environment", which aims at building a *virtual reality* environment for improving

software comprehension, through the visual representation of the software structure and runtime behavior. In order to obtain said data from a *Java* project, it will use the framework described in this work, and display it through a city metaphor by representing packages as blocks and classes as buildings, among other representations. Despite being two separate projects, these can be used in tandem to allow for a complete software comprehension environment, as one project should provide an easy to access *API* to the obtained data analysis and the other one should be able to use this data and display it in real-time.

Given the *live* context these two thesis were based upon, they share requirements regarding the availability of data and how the system as a whole should handle live events such as runtime information or structural updates. As the *virtual reality* environment is the direct interface with the user, the data should be provided to it immediately, as a way to provide *liveness*.

This dissertation is responsible for the extraction, storage and distribution of information regarding a software project. The information should be extracted from a development environment and affect the analyzed source code as minimally as possible. It should then be possible to access this information and be notified of any live modifications.

3.2 Goals and Research Questions

The main goal of this dissertation is to build upon the concept of live software development and explore the necessity of obtaining information from the software and enable the access to that information in a simplified manner. Two research questions can be derived from this goal:

- What important information can be extracted from a software system to improve the developers' comprehension in the context of Live Software Development?
- How can said information be stored in a way that facilitates its fast storage, retrieval and processing?

This dissertation will explore these questions individually at first and arrive at a framework that combines the tools which were built that achieve the goals of these questions. In this context, a framework corresponds to a set of tools and support systems which provide extension points where external tools can connect to access the software metadata collected by the tools framework.

3.3 Requirements

In this section, both functional and non-functional requirements of this framework will be described.

3.3.1 Functional Requirements

Functional requirements describe what the system should be able to accomplish and what external actors want to accomplish in the system. Defining functional requirements and use cases therefore

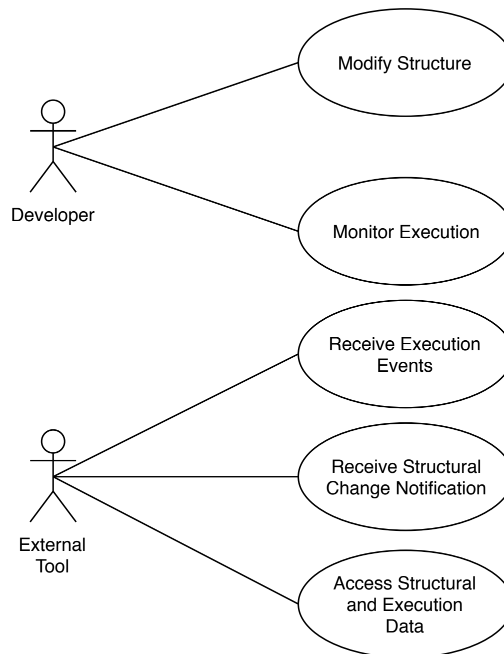


Figure 3.1: Use case diagram detailing the functional requirements of the framework.

implies the definition of actors. In the case of this framework, we are presented with two different actors: the software developer and external tools. The requirements are shown in figure 3.1 and described in further detail the following list.

1. **Software developer.** The software developer will want his work to be processed through the repository.

    1.1 **Project Modification.** The developer should be able to have the project be processed upon modifying its source code.

    1.2 **Execution Analysis.** The developer should be able to have the project's execution be monitored and logged.

2. **External Tool.** An external tool should be able to retrieve data from the repository.

    2.1 **Obtain Structure.** An external tool should be able to obtain the structure of any analyzed project.

    2.2 **Obtain Execution.** An external tool should receive details about the execution of a project.

    2.3 **Structural Change Notification.** An external tool should be informed when a change to the analyzed project's structure occurs.

Non-functional requirement	**Description**
Source Change Performance	Changes to the source code of the project should be propagated quickly enough so the developer has a sense of cause-effect between the modification and the information update in the repository.
Execution Event Performance	Events generated during the execution of a project in analysis should be propagated to the server quickly enough to provide a sense of liveness but avoid clogging the server with too many requests.
Structural Analyzer Performance	The analysis of the source code structure should not impact or hinder the development process (by affecting the performance of the IDE or in other ways).
Execution Analyzer Performance	The analysis of a project's execution should have little to no impact on the performance of the actual project, that is, it should have no added latency.
Structural Consistency	The structural analyzer should provide a way to restore the correct software structure if it ever reaches an inconsistent state.
Space Constraints	The execution analyzer should avoid saturating the server with execution data overtime.

Table 3.1: List of non-functional requirements along with their description

3.3.2 Non-functional Requirements

The system's non-functional requirements describe in what way the functional requirements should be accomplished. This project in particular has a special concern on performance requirements as it needs to interact with both actors (developers and external tools) in the way that will induce the perception of *liveness*.

The two core non-functional requirements that apply to this project have to do with time bounds (time performance) and reliability (data consistency).

Even when considering the different types of data stored in the repository, there are different performance requirements associated with the retrieval of each one. For instance, it is of extreme importance to provide the external tools with the execution events in real-time while on the other hand, structural changes may take a while to propagate to the framework and then the external tools, as it reflects the process of modifying and saving the source code.

That being said, source code changes should still be processed in a reduced timeframe so that the developer has the sense of *liveness* and cause-effect, from the changes in the project to the update in the information on the external tool.

The non-functional requirements are therefore listed in table 3.1.

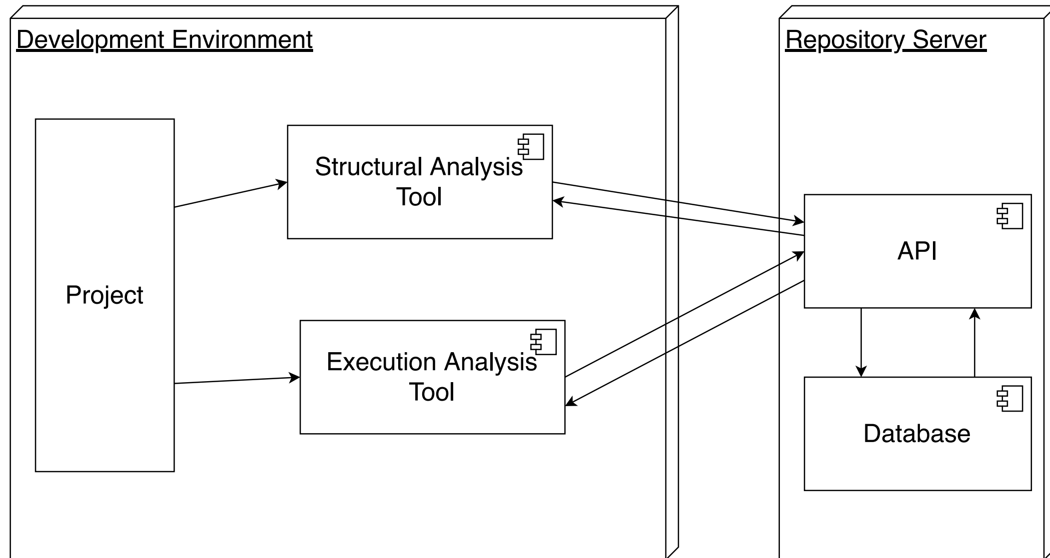


Figure 3.2: Diagram displaying the components of the software analysis framework: the analysis tools and the repository.

3.4 Architecture

There are two levels of architecture that need to be described. First, it is necessary to describe the architecture of the framework which is described in this thesis: its components and how they will interact, what external interfaces it will provide, among other details. Finally, it is important to understand how a live software environment should work and how it would be implemented, using the framework described earlier.

The focus of this thesis, as mentioned earlier, will be on these two concerns: how and what information on software projects can be extracted and how it can be stored as to enable its distribution to external tools which wish to use it.

The proposed solution for these problems is twofold. First, a pair of structural and execution analysis tools should be able to obtain information from the software project. Secondly, a repository should be able to store said information quickly and make it easily available to any external tool.

These two components (analysis tools and repository) will build the framework which will allow other tools to use information on a software without concerning themselves with extracting the information and guaranteeing the communication of that information to it in a specific format.

These components will communicate via a *REST API*, allowing the repository to serve as a tool which will separate the analysis concerns from the concerns of external tools (whether visualization or others). Figure 3.2 shows the interactions between the components of the framework. It is worth mentioning that due to the server-client nature between the components, the development environment can be on the same machine as the repository server or even on separate machines.

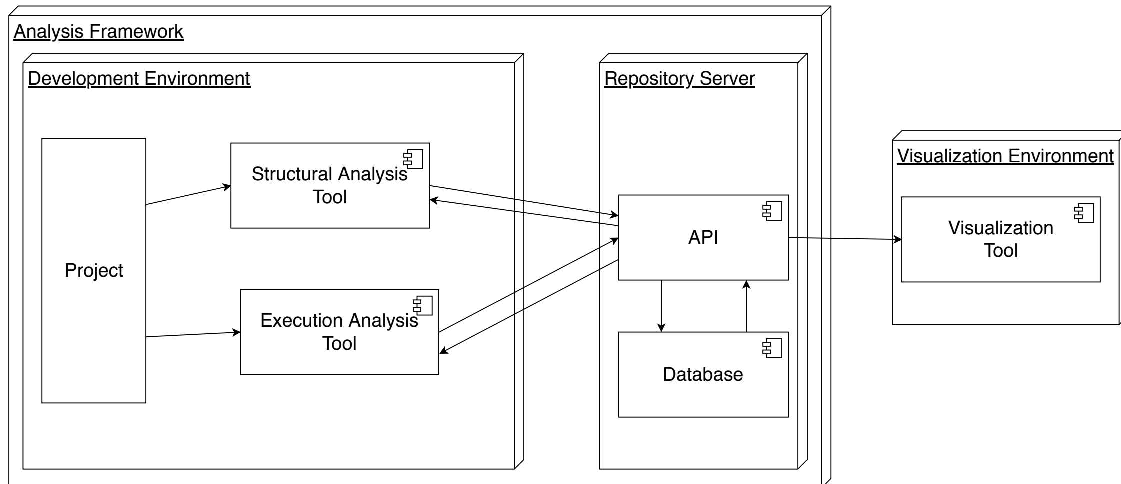


Figure 3.3: Diagram displaying the overall architecture of a full live software development system, complete with the analysis framework and visualization environment components.

Live Software Visualization Environment

The live software development environment envisioned in these projects can be defined as the conjunction of two large, decoupled components: the visualization environment and the software analysis framework. Despite being focused specifically on the software analysis component, and the fact that this component can work on its own to provide information to a variety of tools, it is still relevant to describe how these two components can be combined to create an environment to empower development and software comprehension.

Given the opportunity provided by the "Towards a Live Software Development Environment" thesis which was developed at the same time as the present dissertation, we have an example of how an external tool can use of the information provided by the repository described here.

An overview of the system composed of the combination between the framework described in this thesis and the visualization environment described in "Towards a Live Software Development Environment" can be seen in figure 3.3. As will be mentioned later, external tools will have no direct contact with either the project or the software analysis tools, and is therefore only aware of the repository.

This visualization tool will communicate with the repository, having no direct interaction with the software analysis tools. The interactions with the repository will be done through a *REST API* which will be specified in D. It will then use this information to build a 3D representation of the project structure and allow the user to navigate and interact with it.

In figure 6.4 we can see the results of a project (in this case the *Maze* project) after going through the full pipeline: analysis, information storage and distribution, and visualization.

This figure fails, however, to represent another channel to represent information which was also implemented. As this visualization system was implemented using virtual reality, it was possible to place extra information about inner class structure next to each controller. This means

that when a user looks at his own hand, it can read extra information which is not present in the model representation.

3.5 Summary

In this chapter, the project was described and contextualized within the other relevant project in the *live software development* group. The way these two projects should be able to interact was also detailed to build the concept of the full live software development environment.

Both functional and non-functional requirements were discussed for the framework described. The functional requirements were presented as use cases for the two possible actors: the software developer and external tools. Non-functional requirements were described as particularly focused on the performance concerns of both the analysis tools and the repository, as well as data consistency and space constraints.

Finally, the architecture of the framework, as a combination of the analysis tools and the repository, was discussed, more specifically the manner in which these components communicate. A system using this framework and the visualization environment was also lightly presented as a more concrete example of a full environment.

Proposed Solution

Chapter 4

Static and Dynamic Software Analysis

4.1	**Problem**	27
4.2	**Extracting Software Information**	28
4.3	**Assumptions**	28
4.4	**Structural Analysis**	29
4.5	**Runtime Analysis**	36
4.6	**Summary**	41

This chapter will describe the problem of selecting and extracting the system representative metrics in further and the proposed solution. The resulting set of tools should be able to adapt its output to whatever repository implementation desired.

First, a higher level overview of the tools developed for this environment will be provided, followed by a more in-depth explanation of how they work together with the repository to provide information about the piece of software in question to users and tools.

4.1 Problem

Source code is the most common representation of software. It is through modification of the source code that the basic evolutionary process of software development takes place.

Creating a *Live Software Development* environment implies using a visualization engine for improved software comprehension. This visualization engine, however, requires structural and behavioral data from the software in order to create a useful representation of the system through a viable visual metaphor.

Therefore, it is necessary to determine what is the most important information to retrieve from the software and how it can be obtained. Structural data is required to understand what are the

components of the system and how they depend on each other, while behavioral information is necessary to observe when and how do the components actually interact.

4.2 Extracting Software Information

The proposed solution for this process will begin by determining which are the metrics that best represent the structure and behavior of a *Java* system.

As the point of origin of this analysis is the source code, there are two main paths that can be followed: reverse engineering and forward engineering.

4.2.1 Reverse Engineering

Through reverse engineering higher level representations of the software can be extracted. This will be the basis of the static structural analysis.

We will make use of two representations of a *Java project*. First, the *Java Model* used by the *Eclipse IDE*, which contains information about the Java elements such as compilation units, packages, methods, among others. Secondly, the abstract syntax tree of the software, which is a data structure used by the Java compiler and can be used to overlook minor syntactic details of the code and arrive at an easier to understand representation of the source code structure, from package level down to method level. The combination of these two representations will provide the information on how the system is composed as well as empower the next process.

4.2.2 Forward Engineering

Forward engineering, as opposed to reverse engineering, leads to the lower level representations of the system. This will be the process through which we will observe the system's behavior.

Several possible approaches were mentioned in chapter 2. Between instrumentation, virtual machine profiling and aspect oriented programming. After an overall analysis of how straight forward it is to implement these approaches to the selected case studies, the best fit will be used.

The selected approach will then be used for execution tracing, through event logging, at a granularity that will be decided further during development. One possible option would be event logging every method call, registering the calling class, the called method and the used arguments.

Monitoring would also be a viable option for relevant behavior information. We would need to define resource usage or function execution time thresholds so that an event is logged when one of those thresholds is violated.

4.3 Assumptions

As software comprehension is inherently tied to the maintenance and development process, we assume the tools which will request information from this framework will do so from a development environment.

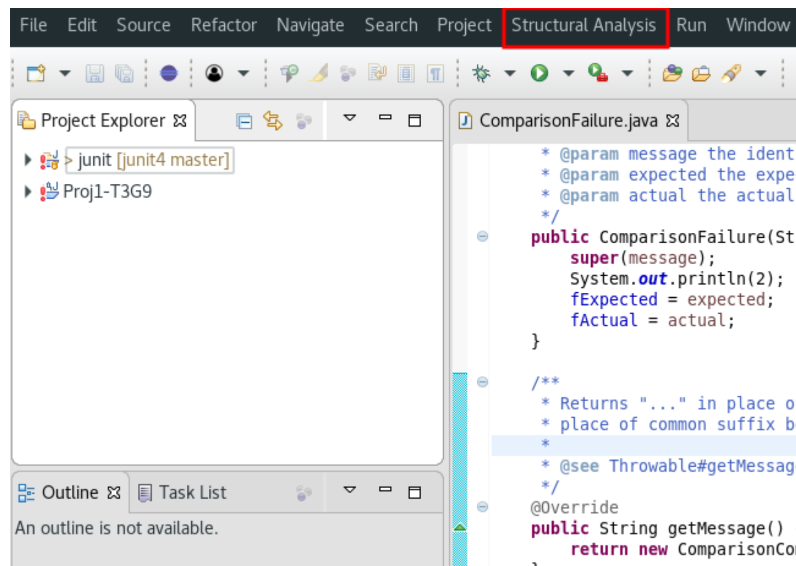


Figure 4.1: Screenshot of the menu added by the structural analysis plugin. This tool mostly runs on background but this button allows the reset of the project structure on the repository.

Furthermore, the focus of these tools will be software projects developed in *Java*. Limiting these tools to a single language, despite reducing versatility, will enable a more detailed and focused analysis of the projects, avoiding the need for extreme abstraction of structural concepts so that it could be adaptable to other languages.

4.4 Structural Analysis

First and foremost, a tool whose purpose is to increase software comprehension will require information of how the software in question is composed structurally. As stated before, in the specific case of a *Java* project, the *abstract syntax tree* can be used to abstract from syntactic details from the language and provide a structure of the elements generated from the source code which is considerably easier to interpret.

In order to have easier access to the abstract syntax tree as well as some other structural details of a *Java* project, and given the assumption of development environment mentioned earlier, the software structure analysis component was envisioned as an *IDE plug-in*, which can be seen in figure 4.1, implemented using the *Eclipse Plugin Development Environment*.

Open source *IDE*'s such as *Eclipse* provide a set of libraries to develop extra components for the development environment. More specifically, a *Java* oriented *Eclipse IDE* contains the *Java development Tools* which among other things, such as access to the *UI* of the environment or even the debugger, allows a *plug-in* developer to access to the structures *Eclipse* uses to represent *Java* projects of all the projects in the workspace.

In order to facilitate the access to these structures, the structure analysis tool was developed as a *Eclipse plug-in*.

Java Model Interface	Interface Description
IJavaModel	The Java workspace currently opened in this Eclipse instance. It is the parent of all projects.
IJavaProject	Represents a Java project.
IPackageFragment	Represents an entire Java package. It is contained in a project and contains compilation units.
ICompilationUnit	Represents a .java file.
IType	A source type in a compilation unit.
IMethod	A method or constructor declared inside a source type.
IField	An attribute/field inside a type.

Table 4.1: Relevant classes from the *Eclipse's Java Model*

4.4.1 Structure

Before deciding what the internal representation of the workspace for the *plug-in* will be, it is necessary to understand the structures *Eclipse JDT* provides access to: the *AST* and the *Java Model*.

Abstract Syntax Tree

An *abstract syntax tree* is composed of *ASTNodes*, which can also be composed of other *ASTNodes*. Each *ASTNode* represents a *Java* language source code construct, such as name, type, expression, statement or declaration. Several other classes exist that extend *ASTNode* in order to include attributes and methods specific to the source code construct they represent.

Given its proximity with the source code, the *AST* the allows fine grained information on where some elements are located in a source file. For example, through the *AST* it is possible to know what is the index of the starting character of method declaration and its length in characters.

Nevertheless, the fact that the *AST* is a powerful representation of a project comes with the significant drawback. Due to its fine grained structural nature, it is considerably more complex to navigate than the *Eclipse Java Model*.

Eclipse's Java Model

The *Java Model* is composed of the classes which model the elements that compose a *Java* program. These classes range from *IJavaModel*, which represents the workspace in question, *IJavaProject*, which represents the project itself, to *IMethod* and *IType*, which represent methods and classes respectively. The *Java Model* elements which were relevant to the structure used by the *plug-in* to represent a workspace are described in table 4.1.

As the *Java Model* structure is considerably easier to traverse than the *abstract syntax tree* due to its coarser granularity, it was used as the main source of information about the project to build the internal model.

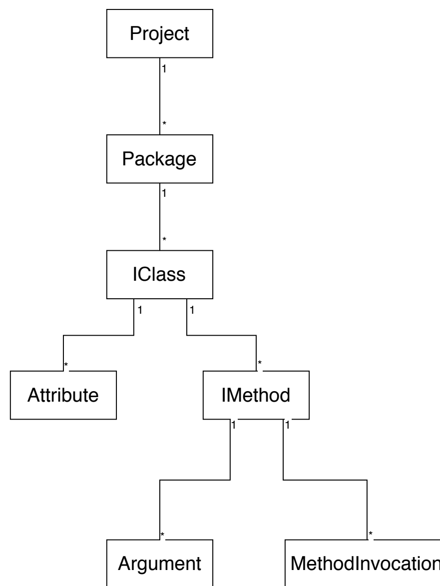


Figure 4.2: Class diagram for Java project model in the Eclipse plug-in.

4.4.2 Plug-in Generated Project Structure

The structure generated by the *plug-in* is mostly based upon the structure provided by the *Eclipse Java Model* classes described in table 4.1. Additionally, the structure also includes attributes which are either not directly extracted from existing attributes in the Java model (derived from one or more attributes in the Java model) or obtained from the *AST*.

To facilitate the comprehension of the elements and their attributes, a small description will now be provided for each one. An high level view of these elements is also present in figure 4.2

- **Project**

  Project Name: the name of a project in the workspace.

  Packages: all the packages in the project, including the parent packages which do not contain any classes, only subpackages.

- **Package**

  Package Name: Name of the package. It contains the name of the parent packages in it, separated by '.'s .

  Has subpackages: Boolean indicating whether this package contains subpackages or not.

  Classes: all the classes in the package.

- **Class**

  Class name: Simple name of the class.

  Hash: a hash generated from the classes canonical name. This hash's relevance will become clear in the section describing the runtime analysis process.

Lines of code: the number of lines in which the class is defined.

Qualified name: Canonical name of the class. That is, besides the class name, it also includes the hierarchy of parent packages in it.

Attributes: All the attributes/fields defined in this class.

Methods: All the methods declared in this class.

- **Attribute**

  Attribute name: Name of the attribute.

  Type: type of the attribute.

- **Method**

  Method name: Name of the method.

  Return type: type of the returned value of the function.

  Lines of code: number of lines of code this method's declaration takes.

  Start of method: index of the first character of the method declaration in the source code file (obtained from the *AST*).

  Length of method: length in characters of the method declaration in the source code file(obtained from the *AST*).

  Key: Resolved binding key of this method. The method binding needs to be generated so this method is considered resolved. For the method to be resolved, the IMethod object for this method has to be generated from the *MethodDeclaration* (a node of the *AST*) through *resolveBinding()* and then getting the resulting binding's associated JavaElement.

  Arguments: the method's arguments.

  Method invocations: all the invocations to methods that occur inside the body of this method.

- **Argument**

  Argument name: Name of the argument.

  Type: type of the argument.

- **Method Invocation**

  Key: Same as the Method's key, but for the invoked method. The resolved binding key guarantees that if the invoked method was declared in the project, the keys will match.

4.4.3 Communication

Given the context within which this *plug-in* is encompassed, it was implemented in a way that allows it to communicate with a server, either hosted remotely or locally. In the case of this framework, this server corresponds to repository.

The *plug-in* provides the repository with the structural information of the workspace and all its projects by sending a POST request in which the body contents correspond to the *JSON* structure represented in excerpt 4.1.

```
1  {
2    "projectName": string,
3    "packages": [
4      {
5        "packageName": string,
6        "hasSubpackages": boolean,
7        "classes": [
8          {
9            "className": string,
10           "hash": string,
11           "linesOfCode": integer,
12           "qualifiedName": string,
13           "attributes": [
14             {
15               "attributeName": string,
16               "type": string
17             }
18           ],
19           "methods": [
20             {
21               "methodName": string,
22               "key": string,
23               "startOfMethod": integer,
24               "lengthOfMethod": integer,
25               "linesOfCode": integer,
26               "returnType": string,
27               "arguments": [
28                 {
29                   "argumentsName": string,
30                   "type": string
31                 }
32               ],
33               "methodInvocations": [
34                   string
35               ]
36             }
37           ]
38         }
39       ]
40     }
41   ]
42 }
```

Excerpt 4.1: Template in *JSON* format for the body of the HTTP POST request of the static analyzer

By sending all the information in one large *HTTP* request, instead of sending multiple small requests, that is, one request per element to its corresponding endpoint in the server, the latency of the structural information upload process is significantly reduced.

4.4.4 Generating the Representation

The actual process of extracting the structure of the projects in the workspace is based on a progressive descent through the *Java Model*. Before the *Java Model* can be analyzed, it has to be generated from the *IWorkspace* class, which represents the workspace in a language agnostic manner. This is done by invoking *JavaCore*'s create method with the current *IWorkwspace* as an argument.

Once the *Java Model* is obtained, we analyze each Java project in the workspace. The analysis of an element of a certain level in the *Java Model* implies the analysis of all their child elements. For example, analyzing a projects implies analyzing that project's package fragments, which further implies analyzing each package fragment's compilation units, and so on.

Although this process may seem trivial, there are some points worth noting in regards to the extraction of the lower level elements in the model. There are cases in which obtaining the child elements of a specific parent element is not as linear as calling a *getChildElements* method which returns an array of said child elements. This is the case when obtaining both the classes' methods and the method invocations within them.

The complexity in obtaining these two types of structural elements arises due to the fact that, in both cases, it is necessary to obtain information about them from the *AST* to be used in conjunction with the information from the *Java Model*. The process through which this is accomplished will now be described in further detail.

- **Converting compilation unit from *Java Model* to *ASTNode***

  The first step consists in converting the compilation unit, in form of a *ICompilationUnit* object from the *Java Model*, to its corresponding node in the *AST*. In order to achieve this, a *parse* function was created, which uses an *ASTParser* to generate an *abstract syntax tree* from the compilation unit and returns this newly created *CompilationUnit* instance.

- **Visitor overview**

  For *AST*'s compilation units, the way to process its descendant nodes is through implementations of *ASTVisitor*. By creating a class which extends *ASTVisitor* and overriding the method *visit*, this method will be called once for each node of the type chosen as argument of the method which descends from the compilation unit. For our case, we implemented two *ASTVisitors*: one for *MethodDeclaration*'s and one for *MethodInvocation*'s.

  These visitors will be instantiated and afterwards the compilation unit in question will accept these visitors so they can analyze it. While a visitor is analyzing a compilation unit it will store the *ASTNodes* it finds which are relevant for said visitor.

- **Resolving bindings**

  This step is crucial for both the *MethodDeclaration* and the *MethodInvocation* visitors. One of the most important attributes in the *plug-in*'s model is the *key*, as it allows the cross reference of invoked methods and methods declared in the project. Although this *key* is provided by the Java model, it is only guaranteed to uniquely represent a method if the Java model element is resolved, that is, its binding has to be resolved.

  In order to *resolve* the method's binding, we have to call the method *resolveBinding* of the relevant *AST* node, and from the resulting *IMethodBinding* object we obtain the resolved *IMethod* object, which we store. From this *IMethod* element we can now get the binding key to identify the method in future invocations.

- ***MethodDeclaration* visitor**

  Method declarations in a compilation unit could be simply obtained from the Java model. However, gathering them this way would not resolve the method bindings. As such, an *ASTVisitor* is required to obtain all the method declarations of a compilation unit.

  The method declaration *AST* nodes are stored and their corresponding Java model element is generated and stored as well. From these two arrays of elements we can obtain most of the method information described in the previous section, including the method arguments' information. The visitor class allows access to the resulting element arrays, so they can be analyzed.

  It is not possible, however, to obtain the method invocations from the method's body using these elements, which is why we need a *MethodInvocation* visitor.

- ***MethodInvocation* visitor**

  This visitor is slightly more complex than the previous one. As it acts on the compilation unit and not on each method from the compilation unit separately, we need to find a way to determine which method body contains which method invocations.

  Upon visiting a method invocation, we crawl up its *AST* until its parent method declaration is found (note: as a simplification, we do not consider invocations from outside method bodies). Once the parent method declaration is found, both bindings are resolved (the method declaration and the method invocation. Finally, the original method invocation node is stored and so are the two resolved Java model methods (invoked method and containing declaration).

  The visitor implements a *getInvocationsFrom* method, which receives an *IMethod* object and returns an array of the binding keys corresponding to methods which were invoked from the given method. This array is then added to the resulting information of the method being analyzed.

Packages also rely on an extra processing step. In *Java*, packages are conventionally named hierarchically, which implies that the packages of a project compose a tree structure. In order to facilitate the understanding of the concept of packages and subpackages, the list of packages obtained through the *Java* model is complemented by generated "parent" packages.

This is implemented by checking each package's name: if it contains a dot somewhere in its name, it means there is a higher level package. This higher level package's name is the child package's name without the substring after the last dot. The packages generated in this manner are then appended to the list of packages in the project.

4.4.5 Live changes

One of the crucial features of the *plug-in* developed for the statical analysis is the ability to detect changes to the source code in real time and reanalyzing the changed elements.

The *Eclipse JDT* provides the mechanism to implement an element change listener, which calls a predefined function once there is a change to a Java element inside the *Eclipse IDE*. The callback function will receive as an argument the *ElementChangedEvent*, from which we can obtain the *IJavaElementDelta* which contains information about which element was changed.

As *IJavaElementDelta* informs us of which element was changed, the representation of the project in the *plug-in* does not have to be rebuilt from the start. Processing time is thus saved by only analyzing the affected elements, from the *Project* level to the *Compilation Unit* level.

Despite the fact that it would be interesting to allow modifications to the *Method* level, *Eclipse JDT* does not provide a notification of change in an *IMethod* element when the method body is changed, only a *ICompilationUnit* level notification. The lowest change listener implemented was therefore at the *Compilation Unit* level.

When communicating the result of this partial analysis to the repository, the *JSON* data sent is the part of the aforementioned *JSON* structure relevant to the element level analyzed. The request is then sent to the endpoint corresponding to said element: "/projects", "/packages" or "/i-classes".

Another important factor to guarantee consistency is the analysis of the workspace when the *IDE* is launched. This compensates for any changes that may have been done to the source code from an external tool, as well as establishing a mechanism to restore the projects' representations to a safe state if any inconsistency issues occur during the detection of live changes.

It is also important to note that if there is any issues with the analysis as a result of incorrect source code (invoking inexistent functions), the model will not be generated and the changes will not be propagated.

4.5 Runtime Analysis

Alongside structural analysis, the analysis of the software's behavior upon execution is extremely important for one to know how a piece of software functions. However, analyzing runtime behavior elicits a myriad of problems.

Depending on the amount of data the analyzing tool extracts from the software at runtime, the data throughput can be massive. Some measures have to be taken in order to mitigate this issue.

Another concern for a runtime analyzer is that it should be minimally invasive, that is, the logging concerns should be as decoupled from the software to be analyzed as possible. For the runtime analysis we built an *AspectJ* project which would weave the generated logging code into the target project with minimal impact to it.

The main focus of this analyzer is method invocations. However, it could easily be extended to monitor other events such as constructor calls, exception handling, among others.

4.5.1 *AspectJ*

As mentioned earlier, one of the requirements for this analyzer is the separations of concerns in the source code, that is, the source code of the original software should not have to be modified in

order for it to be analyzed. This excludes the case of simply implementing a logger as a class in the project and then calling a *log* method whenever it is relevant, adapting it to whichever context it is called from.

Luckily, *AspectJ* provides a way to achieve this, by weaving *advices* into the original code. For the analyzer code to be weaved into the project in question, we need to choose the relevant *join points*, define the *pointcuts* and the *advices*. These steps will now be explained in further detail:

- **Join points**

  The first concern is to choose the relevant joint points. These are the points in a *Java* project in which *AspectJ* allows us to introduce advices. Examples of *join points* are method calls, method executions, constructor calls, field reference, field set and exception handlers, among others.

  For our analyzer, however, we chose to only focus on method calls.

- **Pointcuts**

  Secondly, it is necessary to define exactly what instances of the *joint points* will be weaved with the *advice*. As the goal is to build a generic method call logger, the conjunction of *pointcuts* will have to include calls for any method except for calls which occur in the source code of the analyzer project.

  The *pointcuts* used by the analyzer are the *call pointcut*, which gathers all method calls, and the *within pointcut*, to select all method calls from within the classes of the runtime analyzer and avoid analyzing them.

  Given the fact that the analyzer is provided as an *AspectJ* project, the user can add pointcuts to the existing advice. One possible application for this would be to select method calls originating from a specific class or package by using the *within* pointcut. Besides allowing for a more targeted analysis, it would help the communication process run more smoothly, since the amount of information being sent would be reduced.

- **Advice**

  Finally, we need to define the aspect *advice*. The *advice* specifies the code that will be weaved into the original source code on compilation, at each *pointcut*.

  An *advice* can be set to run before, after or between the joint point. As we want to have a notion of the order of method calls, the advice will be weaved before the method calls.

  For the analysis of a method call, *AspectJ* provides a multitude of variables. In the specific case of the runtime analyzer we use two of these variables: *thisJoinPoint* and *thisEnclosingJoinPointStaticPart*. Most of the data will be extracted from the *thisJoinPoint* variable, while the *thisEnclosingJoinPointStaticPart* will only provide the name of the class where the method call occurs. The data obtained for each method call will be described in a more detailed manner in section 4.5.2.

```
public aspect MethodInvocation {
    pointcut methodInvocation() :
        call(* *(..)) && (!within(MethodInvocation)) &&
            (!within(communication.Logger)) && (!within(communication.RepositoryInterface)) &&
            (!within(communication.Startup)) ; //&&
            //( insert other calls here || call);

    before() : methodInvocation(){
        System.out.println("NOW\n");

        Startup.getInstance();

        JSONObject event = new JSONObject();

        event.put("this", thisJoinPoint.getThis() == null ? "static" : "instance");
        event.put("target", thisJoinPoint.getTarget() == null ? "null" : "exists");
        event.put("kind",  thisJoinPoint.getKind());
```

Figure 4.3: Screenshot of a code segment of the aspect which monitors the execution.

Two hashes are also generated, one from the name of the class where which the method call (the origin class) and the other from the name of the class where the called method is declared (the destination class). These hashes are generated in the same way as the class hash generated by the static analyzer.

Figure 4.3 shows the partial definition of the aspect used to monitor method calls (missing the rest of the advice). The join point corresponds to *call*, while the rest of the pointcut specifies that the advice should not be weaved into method calls of the execution analyzer. Finally, the advice recovers information from the method call and hands it over to the communication interface to send the method call to the repository.

Though an interface was not built for this, as the analysis tool is implemented as an *AspectJ* project, a developer could modify the aspect where the comment *"insert other calls here"* is done in figure 4.3, and add *within* pointcuts to specifically identify the classes or packages where he wants the method calls to be obtained. This reduce the toll on the repository and allow the developer to focus specifically on the method calls in a small set of classes.

After defining the aspect, upon compiling the project, *AspectJ* instruments the resulting code by inserting the code defined in the advice in the points specified by the advice.

4.5.2 Extracted data

The explanation of the data extracted from each method call is done in table 4.2. The main goal of this process is to extract the most valuable information without compromising the dimension of each *event*, considering how there is a massive amount of method calls in a normal piece of software and that these *events* will have to be handled by the repository.

Besides the data present in table 4.2, the analyzer also obtains an array of the arguments used in the method call and for each one stores its type in a field called *type* and a whether it is null or not in a field called *value*.

Event Field	Field Description
this	"instance" if there is an executing object or "static" if the method call occurs from a static method.
target	"exists" if there is a target object or "null" if the method call does not have a target.
kind	Indicates the kind of jointpoint found (in the current implementation it only appears with the value "method-call").
signature	the called method's signature.
class	name of the executing object's class or "null" if there is no executing object (static method).
sourceLocation	a string in the format "ClassFileName.java:<line_of_call>".
originClass	name of the class where the method was called. As opposed to the class field, this is never null. Generated from thisEnclosingJoinPointStaticPart
destinationClass	name of the class which declares the called method.
originHash	hash created from the value of the field originClass
destinationHash	hash created from the value of the field destinationHash

Table 4.2: Data extracted from each method call by the runtime analyzer.

The first implementation stored the string representation of the argument in the *value* field. However, it quickly became apparent that this was not a good idea, as there were cases of long and cumbersome string representations that increased the amount of data which needed to be sent to the repository significantly.

4.5.3 Communication

The communication component of this analyzer is of utter importance given the large amount of data it will transmit. The mechanisms used to avoid slowing down the execution of the software in analysis will be discussed in this subsection, as well as the sequence of communications with the repository and the structure of the messages.

4.5.3.1 Buffering and asynchronous requests

In order to reduce the impact of the analysis and the latency with which *events* arrive at the repository, two approaches are adopted: asynchronous requests and buffering.

Asynchronous requests are the most simple improvement that can be implemented. Especially taking into account that we do not have to process any sort of returning information from the repository, since we favor reduced latency over the guarantee that all *events* are received, asynchronous requests avoid stopping the execution of the original software to send a request and await the server's response. This reduces the performance impact of the analyzer to the original software significantly.

The second mechanism is buffering *events*, that is, storing events in an array and sending a request with all the stored events, clearing the array afterwards, and repeating this process at a fixed time interval. The reasoning behind using buffering is to minimize the impact of the inherent

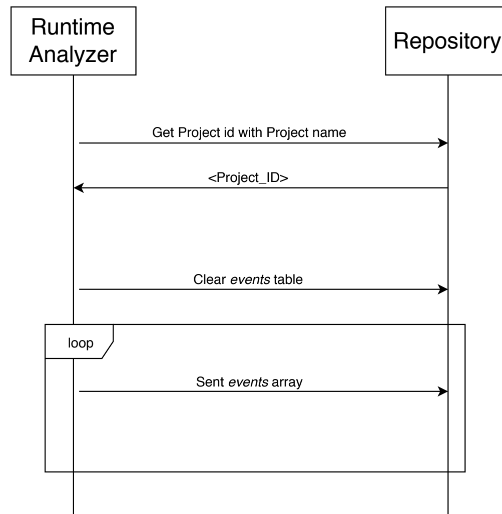


Figure 4.4: Sequence diagram describing the communication between the runtime analyzer and the repository during the former's lifetime.

latency of communicating with the server. Similarly to the reasoning behind sending the whole project structure in a single request, it is better to send one large request and allow the server to process it than to send a large batch of smaller requests.

Though buffering may affect the notion of *liveness*, it prevents unordered *events* and avoids, or at least reduces the likelihood of overwhelming the communication channel with massive amounts of small requests.

4.5.3.2 Analyzer communication sequence

From the startup of the analyzer to when it is actually analyzing the software, the communications with the server are described in figure 4.4.

The first step of the startup process is to preemptively obtain the project's id in the repository. This prevents the need for the repository to constantly query what project has the given name. It does this by obtaining the project's name and using it to query the repository for that project's id.

Secondly, it requests the repository to clear all the information originated from runtime analysis. Given the possibly massive amount of data a runtime analysis can generate, the repository assumes only one project is being analyzed at a time.

Finally, at a fixed interval of time, the analyzer sends the buffered array of *events* to the repository for storage and distribution.

4.5.3.3 Request structure

Although the data extracted by the analyzer has already been described, it is still relevant to describe how it should be sent to the repository. Excerpt 4.2 provides a template of the *JSON* structure of the data to be sent in the body of a *HTTP POST* request to the *"/events"* endpoint of the

repository.

```
{
  events: [
      {
          "this": string,
            "target": string,
            "kind": string,
            "signature": string,
            "class": string,
            "sourceLocation": string,
            "originClass": string,
            "destinationClass": string,
            "originHash": string,
            "destinationHash": string,
            "projectName": string,
            "projectId": integer,
            "arguments": [
              {
                  "value": string,
                    "type": string
                }
            ]
            "timestamp": string (in timestamp format)
        }
    ]
}
```

Excerpt 4.2: Template in *JSON* format for the body of the HTTP POST request of the runtime analyzer

4.6 Summary

In this chapter, the concern of software analysis, both static and dynamic, was discussed. First, the problem of how software analysis is a requirement for *live software development* was described.

A discussion on how software information could be extracted, either through forward or reverse engineering, each method with its separate possibilities. Some assumptions were then detailed in order to describe the scene for the implementation of the analysis tools: the fact that they would be used in a developed environment and how the tools would focus on the *Java* language to allow for more specificity.

The topic of static analysis was then approach through the description of the implementation of a structural analyzer. This analyzer will use both the *AST* and the *Eclipse JDT*'s *Java* model. The plugin's internal representation of a workspace was then described, along with the attributes associated with each type of element, which transitioned to the description of the communication

component of the plugin. Finally, details of how the representation of the workspace is generated were discussed, as well as how the plugin was prepared to detect and propagate changes to a single element, as long as it was whether a project, package or class.

The final topic was the dynamic analysis, where an execution analyzer was described. The choice of *AspectJ* as the core technology for this analyzer was explained and basic information was provided about how aspect oriented programming with *AspectJ* is performed. The data extracted from each method call was described, followed by how that information was sent to the server.

Chapter 5

Repository for Software Metadata

5.1 Model Structure 44
5.2 Websockets 44
5.3 Repository API 46
5.4 Implementation details 47
5.5 Summary 48

This chapter will be dedicated to the implementation of the repository which will be responsible for containing and distributing the information resulting from the analysis discussed in the previous chapter.

The data obtained from the previously described tools will have to be combined in the repository in spite of their differing nature: while the structural information is in a tree format, the runtime information is received in the form of a simple array of *events*.

Given the type of data the repository will have to handle, the database technology selected as the basis of the repository was *PostgreSQL*. This way we can build a structure to represent the software which is very similar to the structure of the output of the static analyzer, and associate the events generated by the dynamic analyzer to the corresponding elements of the software structure, without having to handle different database technologies simultaneously just to have a more specific fit to the data.

Another advantage of using a *SQL* database is the ability to have an *key-value* table to link structural elements to another type of analysis and the value associated with it. That is, with the structural information as the most crucial data in the repository, it would be possible to offer *plug-and-play* capabilities to other analysis plug-ins, such as profilers.

To facilitate the development of an *API* for the plug-ins and the tools which will use the data in the repository, *Ruby on Rails* was used to build the server. Using this technology also provides some other facilities which are important to the performance of the repository, namely websockets.

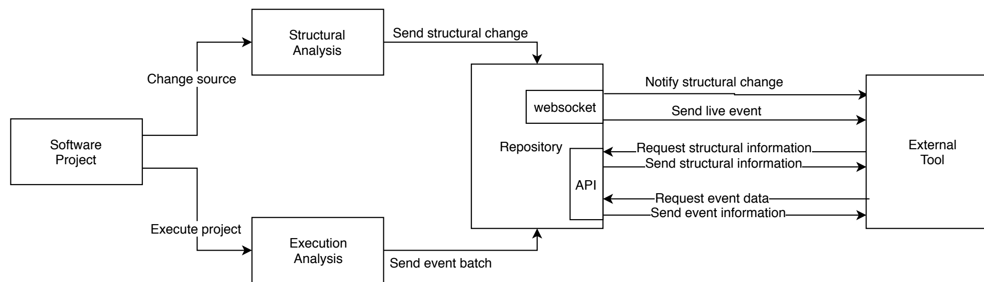


Figure 5.1: Interaction diagram between the software analysis tools and the repository, as well as the developers and external tools.

The overview of the interaction between the software analysis tools and the repository, as well as the developer and external visualization tools, can be seen in figure 5.1.

5.1 Model Structure

Firstly, it is important to define the structure of the model the repository will store. This structure has to contain the information from the static and dynamic analyzers and combine it. The structure is described in the diagram 5.2.

The fields of each table correspond to what was described as the fields in the output of each one of the tools, in chapter 4.

Rails eases development by generating the database tables related with each model in this structure, as well as providing a simple way to create a controller for each model which will handle requests received through the *API*.

The repository database is built so that multiple projects can be stored at once. However, due to the large amount of events the runtime analysis can generate, we assume the repository only contains events one execution of a single project at one point in time.

5.2 Websockets

Storing information in the database has an associated delay, especially when it is necessary to perform constraint checks, such as foreign keys. One of the requirements of this repository, however, is to have a reduced latency so that external tools can receive the static and runtime information in real-time.

By using *Rails'* implementation of websockets (*Action cable*), the repository allows external tools to connect to a websocket and receive two different types of information from it, which will now be described in further detail.

Websockets allow the implementation of a *publish-subscribe* patterns for the prioritized information that needs to be sent in real time, such as notification of structural changes and new execution events.

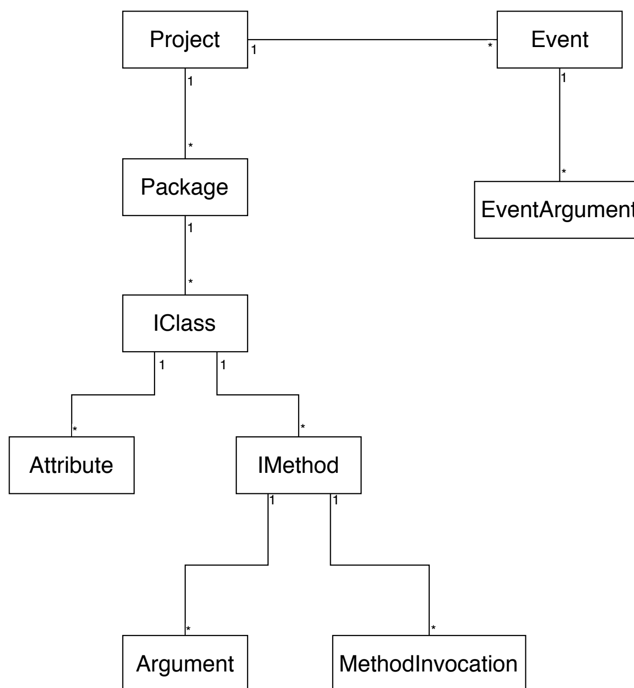


Figure 5.2: Database structure diagram containing the projects' structure and runtime data.

Rails provides three different subscription adapters for websockets: async, which is only recommended for development and testing environments, *PostgreSQL* and *Redis*. The subscription adapters were configured for *PostgreSQL* due to the advantage of not having to setup *Redis* and requiring barely any configuration.

A channel was build to transmit said time sensitive information, called *ControlChannel*. In order to subscribe to that channel, a client application has to open a websocket client and send the message described in excerpt 5.1 to the */event_stream* endpoint. It is then able to receive the stream of messages.

```
1 { "command": "subscribe", "identifier": "{\"channel\": \"ControlChannel\"}" }
```

Excerpt 5.1: Subscription message to be sent through the websocket client to the */event_stream* endpoint

5.2.1 Event information

Upon receiving an array of events, the repository sends these events to the websocket, so that any external tool listening to it can receive this information in real time.

Considering the previously mentioned requirement, there is the assumption that for the sake of liveness it is not crucial to guarantee consistency with the project's structural information at real-time. Therefore, the events are sent through the websocket before being inserted into the

database, which allows external tools to both receive the events in real time as well as consulting the database for the events which occurred inside a specified time interval.

The structure of the event data sent through the websocket matches the structure of the event data obtained through the *REST API*.

5.2.2 Structural changes

The other type of message that is sent through the websockets is a control message warning any listening external tool that a project's structural information has been updated. This message has a specific structure that allows tools to discern the type of element which was changed (project, package or class), if the element was changed or deleted, and the id's in the database of the parent elements so that the tool can easily have access to the updated information.

The structure of the message is described in the excerpt 5.2.

It is the external tool's responsibility to decide whether it will fetch the entire structure of the project again or only the modified element and apply the changes to the local model itself.

```
1 {
2   "fetch_structure": string,
3   "operation": string,
4     "project_id": integer,
5     "package_id": integer,
6     "class_id": integer
7 }
```

Excerpt 5.2: Template in *JSON* format for the *structure update* message sent through the websocket

It is important to point out some particularities in these messages. First off, the *fetch_structure* field can have as value "class", "package" or "project". Secondly, the *operation* field can have the value "change" or "delete".

Lastly, the fields *project_id*, *package_id* and *class_id* only exist from the higher level element to the element which was modified. For example, if a package was modified, the *class_id* field would not make sense, but both *package_id* and *project_id* would.

5.3 Repository API

The repository was build on the notion that external tools, whether analysis, visualization or with another purpose, should be able to interact with it. To achieve this, the repository provides an *API* which gives *CRUD* (create, read, update and delete) access to the data model.

Given the extensive nature of the *API* documentation, it will be described in a document which will be present in appendix D. However, a couple of specific cases will be specified here.

5.3.1 Storing projects

As mentioned earlier, the *API* provides *CRUD* access to every level of the structural model. However, as was discussed in the previous chapter, it is important to provide a way for the project information to be sent to the repository in a single request.

For this purpose, the */projects* endpoint will check for the existence of the *packages* field in the request before inserting the information in the database. If this field is not present, it means that the request should only affect the *projects* table in the database.

However, if this field is present, it implies the whole project information is present in this request and it is therefore necessary to insert the nested elements of the project (packages, classes, methods, etc.).

5.3.2 Reading projects

In similar fashion to inserting information in the repository, it is important to provide the user with the distinction between simply reading the top level information related to a project and reading all the nested information in that project, all the way down to the method invocations.

In order to provide this control, the */projects* endpoint will look for a specific *URL* encoded variable. If the variable *deep* exists and is set as true, the *GET* request will return all the nested information of the given project.

5.4 Implementation details

A few other implementation details are worth pointing out, as they have impact in the operations of the repository. As basic as these details may seem, their impact makes their discussion worthwhile. These details will now be listed out and lightly discussed.

- **Silencing the *ActiveRecord* logger**

  When inserting small batches of data, this does not appear to have any performance impact. However, there is always an added delay with each extra task the server has to do. The amount of logging output that is generated when inserting a project of the scale of *JUnit*, for instance, into the database, results in added processing time, which can be avoided by silencing the logger.

- **Filtering large *POST* request fields**

  Similarly to what happens with the *ActiveRecord* logger, the rails server may attempt to output large amounts of information to the console. In this case, it happens on every request to the *API*. If the contents of the request are large enough, the console output may cause some performance issues. By filtering out these fields (mainly nested fields, such as "packages" in a */projects* POST request) we force the rails server to not print this data.

- **Using *activerecords-import***

This useful ruby gem provides a way to reduce the number of inserts to the database. *ActiveRecord* provides a wrapper to the database, and as is the case with *HTTP* requests, there is a inherent delay to each request. Therefore, minimizing the number of inserts will have a positive impact on performance.

The *activerecords-import* gem aggregates every element that is to be inserted into a given table in an array and inserts them using a single query instead of one query per element.

5.5 Summary

In this chapter, the software metadata repository was described in detail. First, the inner model structure was detailed. It was built in such a way that the data originating from the analysis tools easily fit the model. The choice of *PostgreSQL* as the database technology was justified by the adaptability of this technology to a variety of possible data structures, as well as the easier integration with the *Rails* server, which acts as the core of the repository.

Secondly, the main mechanism to provide the liveness characteristic to the repository was described. ActionCable's websockets were used for this purpose, implementing a *publish-subscribe* component to the repository. It was explained what kind of information would be sent through this websocket: meta information about structural changes and execution events generated by the execution analyzer. The protocol for clients to connect to this channel was also described.

The repository's *API* was not described in detail as that is the responsibility of the *API* documentation present in D. Nonetheless, some particularities, such as the *URL* encoded argument for retrieving the nested elements of a project or how a project is inserted in the database in a single POST request were discussed.

Finally, some implementation details that affect the performance of the repository were examined: silencing the *ActiveRecord* logger, which had a slight impact in performance, similarly to filtering *API* request fields, and the use of *activerecords-import*, a much more significant addition which reduces considerably the number of queries to the database when inserting data.

Chapter 6

Experiments & Results

6.1	Case Studies	49
6.2	Functional Requirements	50
6.3	Performance Evaluation	50
6.4	Visualization Engine Validation	58
6.5	Summary	58

In this chapter the framework described in the previous chapters will be verified. Firstly, the case studies which were used to help develop, test and verify the correct functionality of the framework will be presented. There will be a section focusing on how the tool achieves the described functional and non-functional requirements. Afterwards, a section will be dedicated to the validation work done by the parallel thesis "Towards a Live Software Development Environment" and how it helps indirectly validating this framework.

6.1 Case Studies

In order to test and validate that the framework developed is working as originally intended, two *Java* projects were used. These projects are both developed in *Java* and were chosen as they provide a different set off differing characteristics, namely the structural complexity.

6.1.1 *Maze*

The *Maze* project was the first project developed in the context of the *Object-Oriented Programing Laboratory* course of the *Integrated Masters on Informatics Engineering and Computing* degree. It is a project developed in *Java* which was developed with a focus on the projects architecture and correct separation of concerns in packages and classes.

This project was developed in 2014, over the course of half of a semester, by groups of 2 elements. The projects.

The reason for choosing this project was threefold. Firstly, as this project was developed partially by the author, there's an inherent familiarity with the project, its structure and inner workings. Secondly, its a small project which allows easier verification of whether the analysis and data storage is being done correctly or not. Lastly, despite its small size, it still has some degree of architectural complexity, that is, it contains a reasonable number of each one of the structural elements (packages, classes, etc.).

6.1.2 *JUnit4*

JUnit is a unit testing framework widely used in *Java* projects. This software is commonly used in literature as a case study.

It was selected to be one of the case studies as there was a necessity to validate the functioning of the framework on a larger scale project. This larger scale should push the capabilities of the structural analysis tools to build the representation quickly, as well as the repository's capability to receive and distribute this volume of information in a timely manner.

6.2 Functional Requirements

During the development process of the software analysis tools and the repository, there was a consistent confirmation that the functional requirements were met. The functionality if obviously a core part of the framework and the development process would only be finished once the functionalities earlier proposed were met.

From the developer point of view, as expected, upon modifying source code of a project in analysis, the structural changes are reflected in the repository. In similar fashion, once the project in analysis is executed, a feed of execution events starts being sent to the repository.

From the external tool point of view, that is, from the reading endpoints of the *API* and the server's live feed *websocket*, it is possible to obtain the structural information of every analyzed project (*API*), the events generated be a project's execution (*API* and *websocket*) and be notified of any structural changes in a project, including what specific element was changed (*websocket*).

6.3 Performance Evaluation

As for non-functional requirements, the methodology used for verifying the correct functioning of the framework and validating that it actually conforms to the requirements defined was through experimental runs using the selected case studies.

The steps taken towards validating the framework can be separated into two main segments: the experimentation associated with the structural analysis and the experimentation related to the execution analysis. Despite the division being made along the two software analysis tools, the validation experiments for both also concern the repository.

Also worth noting is that these experiments will only involve the framework, not including any external tool that may eventually use the framework for the information it provides.

6.3.1 Structural Analysis

The focus of these experiments was to study the performance of the framework when analyzing the structural components of a software project. Though it is not as time sensitive as the execution analysis, it is nonetheless important for the framework to reflect changes to a software in a timely manner, as was mentioned in the non-functional requirements in chapter 3.

In order to do so, we obtained metrics which provide information of the time it took to process the structure change, segmenting this timeframe into smaller relevant segments. This was done by obtaining time stamps at specific points along the path responsible for handling the structural information.

6.3.1.1 Experimental Scenarios

The experimental scenarios were composed of the structural analysis tool and the repository. The execution analyzer was not included as it will be the focus of other experiments.

The structural analysis plugin was installed in an instance of the *Eclipse IDE*. On the other hand, the repository was deployed in an external machine present in the same local area network as the development machine. This was to provide a "worst case scenario" given the presence of network latency, which will be noticeable on the results.

Time monitoring points were set up both in the analysis tool and the repository to monitor the time each step of the data retrieval took. The points in which a time monitor was placed are described in table 6.1. A unique identifier was associated to each one to facilitate the mention of the relevant monitors in each experiment.

These timestamps extracted from the execution can be combined to create performance metrics, which are described in table 6.2. Two additional metrics could be created that relate to the impact of the data transmission through the network, namely *Project Transmission Time* and *Class Transmission Time*. As these metrics are very dependent on the network topology, they do not really provide much information on the performance of the tools in separate or in conjunction, and so they will be ignored. Nevertheless, as the network topology used in the experiments represents a very likely development environment for this framework, the impact of the data transmission on the *Full Project Analysis Time* and *Full Class Analysis Time* metrics will not be removed.

The goals of these experiments were to verify the performance of three core functionalities related to both the repository and the structural analysis, which are the following:

- **Experiment 1 - Project insertion into the database:** the first scenario tested was inserting a project into the database for the first time. This was done with both case study projects: *Maze* and *JUnit*. Relevant metrics for this experiment are the *Full Project Analysis Time*, *Tool Project Analysis Time* and *Project Creation Time*.

Time Monitoring Location	Description
Structural Analysis Tool	The start of the analysis of a given project in the workspace.
	The conclusion of the analysis of a given project in the workspace.
	Sending the request to the repository API with the project information.
	Receiving response from the server to the project creation request.
	The start of the analysis of a given class in a project.
	The conclusion of the analysis of a given class in a project.
	Sending request to the repository API with the class information.
	Receiving response from the server to the class creation request.
Repository	Receiving the project creation request sent by the analyzing tool.
	After inserting the project's structure in the database.
	Receiving the class creation request from the analyzing tool.
	Responding to the class creation request from the analyzing tool.

Table 6.1: Time monitoring points in the both the structural analysis tool and the repository, as well as their location in the structural analysis process.

Metric	Name
Time for project analysis and creation in database.	Full Project Analysis Time
Time for tool to analyze project.	Tool Project Analysis Time
Time for server to insert project data into the database.	Project Creation Time
Time for server to update existing project data in the database.	Project Update Time
Time for class analysis and modification in the database.	Full Class Analysis Time
Time for tool to analyze modified class.	Tool Class Analysis Time
Time for server to insert/modify class into the database.	Class Creation/Update Time

Table 6.2: Performance metrics related to the structural analysis.

	Metrics		
	Full Project Analysis Time	**Tool Project Analysis Time**	**Project Creation Time**
Maze	1642ms	340ms	1054ms
JUnit	5737ms	1854ms	3146ms

Table 6.3: Table displaying the results of the first structural analysis experiment.

- **Experiment 2 - Project information update:** the second scenario was inserting a project which was already present in the database. As with the previous experiment, both *Maze* and *JUnit* were used for this experiment. Given that *Tool Project Analysis Time* was already taken into account in the previous experiment for the same projects, this experiment will be evaluated through the *Full Project Analysis Time* and the *Project Update Time*.

- **Experiment 3 - Class creation/update:** the final scenario for structural analysis was altering a class file and observing the propagation of the modifications. This experiment only involves the *JUnit* project, as the dimension of the project itself no longer is a concern. The metrics resulting from this experiment are the *Full Class Analysis Time*, *Tool Class Analysis Time* and *Class Creation/Update Time*.

6.3.1.2 Results and Discussion

The results of experiment 1 are shown in table 6.3. The first thing to notice is how the dimension of the piece of software actually impacts both the analysis and the storage performance. This was already expected, as a larger structure has to be analyzed and then converted into the repository's representation of a project (more elements have to be created). Another interesting conclusion is the fact that the bottleneck when inserting entire projects into the database is on the repository itself.

Table 6.4 shows the results of experiment 2. The *Tool Project Analysis* times are very similar to the experiment one, which can be explained by the fact that, when updating the whole project, it is fully analyzed as if it were the first time. However, as the database has to clear its records of the projects to be updated, the insertion in the database takes considerably longer, roughly twice as long as simply inserting the project in the database for the first time.

Finally, the results of structural analysis experiment number 3 are represented in table 6.5. The dimension of the project does not have an impact on this experiment, as only a class will be updated. The class file modified in this case contains 171 lines of code, which is a reasonable

	Metrics	
	Tool Project Analysis Time	**Project Update Time**
Maze	353ms	2310ms
JUnit	1839ms	6640ms

Table 6.4: Table displaying the results of the second structural analysis experiment.

	Metrics		
	Full Class Analysis Time	**Tool Class Analysis Time**	**Class Creation Time**
JUnit	40ms	22ms	7ms

Table 6.5: Table showing the results of the third structural analysis experiment.

size for a class. The whole process of analyzing a class takes a very small amount of time when comparing to the full project analysis.

Through these experiments we can draw several conclusions. The first one would be that the repository consumes a larger amount of the total analysis time. This was already expected given the fact that the repository has to build the projects representation in its internal model and then store it in the database. These results could be worse, however, had *activerecords-import* not been used to reduce the amount of queries needed to insert the project in the database.

Despite this clear drawback at first glance, the repository provides a unique capability that outputting to a file would not offer, which is writing a partial analysis. This is a great advantage when one compares the results of experiments 2 and 3. If modifying a project source file required analyzing and inserting the entirety of the project once again in the repository, the repository would quickly become impracticable. However, by reducing its analysis to the scope of the modified file and inserting it alone into the database reduces the processing time required to analyze a modification by two orders of magnitude in the specific case of classes, which are the most commonly modified structures.

6.3.2 Execution Analysis

Complementing the previous section, the focus of this set of experiments was to verify the performance of the framework when analyzing the runtime behavior of a software project, more specifically detecting method calls. Timeliness is a crucial aspect of the runtime analysis, as the framework has to make this information available in real time to enable the concept of liveness, as was mentioned in the non-functional requirements in chapter 3.

In similar fashion to the previous set of experiments, we obtained metrics which informed of the time it took to events representing method calls, allowing the segmentation of this large timeframe into different pieces. This was done by obtaining time stamps at specific points along the execution analyzer and the repository's events controller.

6.3.2.1 Experimental Scenarios

The goal of the experimental scenarios described in this section is to analyze the performance of the execution analyzer in conjunction with the software repository.

In this set of experiments, only the *Maze* project was used. The amount of data generated no longer depends solely on structural complexity, but on the processing capabilities of the machine the program is running on.

Time Monitoring Location	Description
Execution Analysis Tool	Event buffer is sent to the repository.
	Specific event was generated and sent to the buffer.
Repository	Event set was received by the repository.
	Event set started being imported into the database.
	Event set finished being imported into the database.
	Specific event was received by the repository.
	Model for the specific event has been built.
	Specific Event was sent through the websocket.
	Specific event was imported into the database.

Table 6.6: Table containing the location and description of each of the checkpoints for the performance analysis of the combined use of the runtime analyzer and repository.

Similarly to the environment of the past experiments, the repository was deployed in an external machine present in the same local area network. The runtime analyzer was imported into the workspace and linked to the *Maze* project. No specific classes or methods were selected to be monitored, so that the analyzer would be able to collect every method call and generate an event from it.

The time monitoring points defined for this set of experiments is described in the table 6.6 and the metrics derived from it are detailed in table 6.7.

The experiments performed can be described as follows:

- **Experiment 1 - Generating and storing a set of events:** this scenario consists of running the *Maze* project with the execution analyzer linked to it, producing several sets of events, sequentially sent to the repository, at a fixed timed interval. The generated metrics associated with each of the event set transmitted are the *Full Event Set Time*, *Event Set Size*, *Repository Event Set Time*, *Event set Modeling Time* and *Event Set Import Time*.

- **Experiment 2- Following the lifetime of an event:** the concept behind this experiment is following a specific event from its creation to the final moment when it is imported into the database. The generated metrics associated with this experiment are the *Specific Event Life Time* and the*Specific Event WebSocket Time*.

6.3.2.2 Results and Discussion

The timeline shown in figure 6.1 describes the timestamps and time intervals relevant for the first experiment.

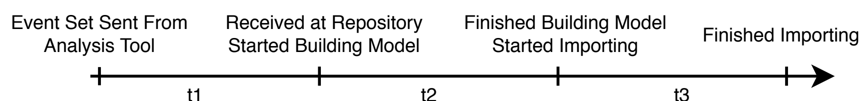


Figure 6.1: Timeline for the event set lifetime.

Metric	Name
Time taken to fully process a set of events	*Full Event Set Time*
Size of the set of events processed	*Event Set Size*
Time the set of events takes to arrive at the repository	*Event Set Transmission Time*
Time taken to convert the event set to the repository's inner model	*Event Set Modeling Time*
Time taken fore the event set to be imported to the database	*Event Set Import Time*
Time taken for a specific event to be imported to the database since it was generated	*Specific Event Life Time*
Time for a specific event to be sent through the websocket	*Specific Event WebSocket Time*

Table 6.7: Table containing the metrics generated from the checkpoint data regarding runtime analysis.

In this first experiment, the buffer timer length does not affect the values as the start of the timeline in figure 6.1 is the moment the event set is transmitted.

We gathered information about four buffer transmissions. Given the largest size out of these buffers (5690) and the smallest (1420), the event sets will be grouped into intervals three: [0-2000[, [2000-4000[, [4000-6000[. For each of these intervals, we will aggregate the grouped event sets by the average value of each time performance metric.

Figure 6.2 show the scatter plot and linear regressions associated with these time values and event set sizes.

The second experiment involved the lifetime of a event, from the moment this event is sent to the buffer to the moment it enters the database. Figure 6.3 describes the timeline of this series of events.

As opposed to the previous experiment, this one is affected by buffer timers. We experimented with the buffer timer at five and two seconds. Table 6.8 shows the values associated with the time intervals in 6.3.

From the analysis of a singular event, one can see that it will arrive more quickly at the websocket and the database. This is most likely due to the combination of the longer average waiting time in the buffer, the transmission of a larger event set to the repository and the larger number of elements to be converted into the repository's inner model before it can be imported to the database.

		t1	**t2**	**t3**	**t4**
Buffer Timer	5 seconds	3700ms	1ms	0ms	3694ms
	2 seconds	1402ms	1ms	0ms	2193ms

Table 6.8: Time intervals related to the lifetime of a single event from the moment it was inserted in the buffer until it is inserted in the database.

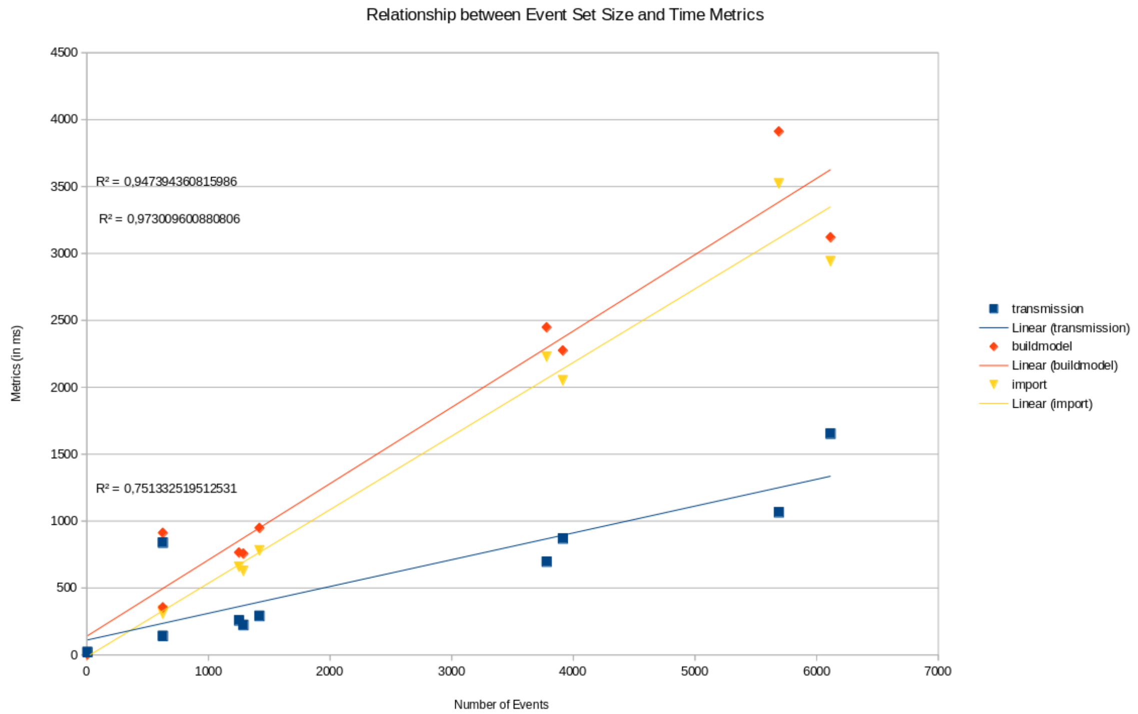


Figure 6.2: Scatterplot and linear regression of the time values (transmission, model building and importing) in relation to the event set size.

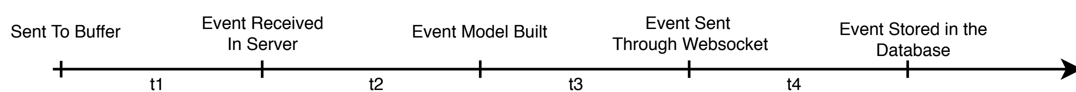


Figure 6.3: Timeline of the lifetime of an event, from being sent to the buffer to being sent throught the websocket and into the database.

We can conclude that a two second buffer timer allows for smaller event set requests, more frequently without saturating the server, providing a good sense of liveness.

6.4 Visualization Engine Validation

"Towards a Live Software Development Environment", as mentioned in chapter 3, is a dissertation pursued in parallel with this dissertation. This project resulted in a visualization environment which uses data gathered and made available by this framework.

The proper functioning of this framework was paramount to the correct functioning of the visualization environment and as such, this visualization environment's validation can be correlated with this frameworks validation.

In order to validate the effectiveness of a live visualization environment, a survey took place where users were tasked with exploring the *Maze* case study in the virtual reality visualization environment. The surveyees were then asked to identify two possible issues (not necessarily errors) injected into the software, which do not appear as issues in a regular *IDE*: an infinite loop and an invocation with a number of null arguments greater than zero.

The experiment with the visualization environment was performed by 25 different surveyees, having received overall positive results towards the usage of the virtual reality visualization engine.

This in turn helps validate the usefulness data this framework extracts from the static representation of software, as well as from its runtime behavior, and the manner in which the repository makes this data available to external tools, such as this visualization environment.

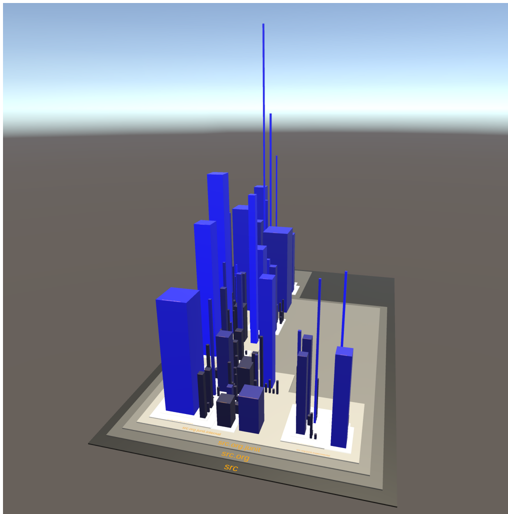
Figure 6.4 shows the resulting visualization from the virtual reality environment built in this dissertation. In figure (a) we can see the structural representation of the *JUnit4* project while in figure (b) we can see the *Maze* project running.

6.5 Summary

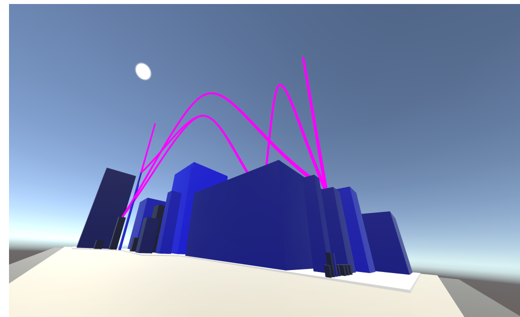
In this chapter we described the *Maze* and *JUnit* projects and why they were chosen to be the case studies. The different structural complexities between the two, the fact that *JUnit* is a recurring project in literature when *Java* project analysis is required and the familiarity with the *Maze* project.

It was discussed how the constant development and testing process with the case studies project allowed the verification of wether the framework being developed matched the functional requirements.

We also described the experiments we delineated to provide a performance evaluation and test if the framework conformed to the non-functional performance requirements. The description of each experiment was followed by the discussion of the results obtained, where we concluded that the framework is in fact, in line with the performance and liveness requirements, and ready to be used with an external tool.



(a) Top view of the visual representation of the *JUnit* project's structure.



(b) View of the visual representation of some *Maze* project's execution events.

Figure 6.4: Two figures from the visual representation generated from the analysis of the *JUnit* and the *Maze* projects respectively. Figure (a) shows a good perspective of the structure of *JUnit* while figure (b) shows some events between structural elements of *Maze*.

Finally, the validation process of the dissertation "Towards a Live Software Development Environment" was lightly discussed, as well as how the good results in this process served as a validation for this framework, since it's correct functioning was required for the environment developed in that dissertation to work properly.

Chapter 7

Conclusions

7.1 **Main Contributions** 62
7.2 **Future Work** 62

Through the literature review process, the technologies and approaches to use in the project were identified and molded the development process.

In the problem overview, this problem was put in context with the live software development concept and two research questions were built from the problem. Both functional and non-functional requirements were described for a framework that conformed to these research questions. Furthermore, the architecture of this framework was described, and it was put into context within a possible full live software development environment with an external visualization tool. Finally, the case studies used in testing and validating this framework were documented.

In chapter 4, the static and dynamic software analysis tools were described in depth, explaining how these tools allow for the analysis of the structure and execution of a *Java* project. From a high level overview of what each of these tools should do, to low-level details of how these tools were implemented, the components which interact directly with the software in this framework were discussed.

Chapter 5 provided the high level and implementation details of the repository which stores and provides the information recovered by the analyzers. The usage of a *REST API* for the *CRUD* access to the model as well as the websocket made available to receive real-time notifications and events was discussed in detail.

Finally, chapter 6 focused on how we tested the performance of the framework and how it conformed with the predefined functional and non-functional requirements.

7.1 Main Contributions

The field of software analysis is a rich one in regards to literature. There was however a large focus on full analysis, with tools which fully self contained from the analysis to the displaying of information. The main contributions of this thesis are therefore listed as follows:

- A structural analysis tool for *Java* projects which can be included into any *JDT* enabled *Eclipse IDE* as a plug-in, capable of recognizing changes to several levels of the *Java Model* tree.
- An execution analysis tool for *Java* projects which can be included in the relevant workspace and added to a project with minimal modifications to the concerning project required.
- A software repository ready to receive information from the previously mentioned analysis tools, and provide it in real-time to any external tools through an *API*.

To complement the contributions mentioned, the *API* is detailed in appendix D, and the instructions to install and use the repository server, structural analysis tool and execution analysis tool are in A appendixes B and C respectively.

7.2 Future Work

Unfortunately the time allowed for the development of this project is limited and as a result several other features which would be interesting to have in this framework could not be implemented. This section will serve the purpose of describing ideas for what could be advancements and improvements in this toolset.

- **Complement model with missing types of *Java* elements and relationships between them:** this is the case for abstract classes, interfaces, inheritance, among others. This was not possible to implement in the current state of the framework due to time constraints, but would make for a good addition, as it would make the data model used more complete.
- **Introduction of other levels of structural abstraction:** it would also be interesting to provide more levels of abstraction in the structural representation of the software. This could be the case for obtaining design patterns and architectural patterns from the software.
- **Plug additional runtime analysis tools to the repository:** there are plenty of other tools which can provide useful information about how a piece of software executes, for instance, profilers. Additional analysis tools could be adapted to output these metrics to the repository so that external tools could access that information in conjunction with the data already provided by the current state of the framework.
- **Build an improved interface for the execution analysis:** One of the advantages of the runtime analyzer described here is the ability to specify which classes we wish to log, instead

of logging all the classes in a project. This, however, has to be done through changes in the aspect file itself. A more user-friendly solution could involve building a *GUI* to allow the user to introduce the classes he wishes to analyze and then inject that specification into the aspect template file already produced for the analyzer in this dissertation.

- **Database optimizations:** The database is one of the core elements in this framework. It would be extremely useful to improve the performance of this database. Aside from *PostgreSQL* performance optimizations, it would also be interesting to experiment with other database technologies, for example graph based databases and time series databases, and understand which setup is the most adequate for this framework.

As one can see, there is still a large margin for further improvement of this framework. Nevertheless, the work accomplished in this dissertation sets a foundation upon which to develop this advances.

Appendix A

Appendix A

A.1 tese-repository

Repository for Java projects' structural and execution data.

A.1.1 Description

This is one of the three components necessary for the correct functioning of the framework developed in "A Software Repository for Live Software Development". This component is responsible for aggregating and distributing the information recovered by the two other analysis tools. It is composed of a Rails server, along with a PostgreSQL database and nginx.

The two other analysis tools are the structural analyzer and the execution analyzer.

A.1.2 Write Access

Write access is provided for every component of the structure and execution data. Writing structural data to the repository assumes the existence of the nested attributes. For example, when inserting a new *i_class*, it expects the class' attributes and methods.

A.1.3 Read Access

A.1.3.1 Database

The repository provides read access to every element (structural or execution) in the database, as well as the possibility of listing all the elements of that type. In the specific case of project, there are two possibilities: either retreiving the shallow project metadata or, using the `deep=true` GET field, retreiving the full information of the project, with all nested attributes included.

For the events, there is a `from` and a `to` field that can be set as two timestamps to retreive all execution logs between those two points in time.

A.1.3.2 WebSocket

The repository provides access to live data through a websocket. In order to connect to it, the interface has to create a websocket and send the following subscrtiption message:

```
{ "command": "subscribe", "identifier": "{\"channel\": \"ControlChannel\"}"
}
```

If the structural and execution analyzers are correctly configured and installed, you should be able to receive two types of messages through the websocket.

1. Events

```
{
  "this": string,
  "target": string,
  "kind": string,
  "signature": string,
  "class": string,
  "source_location": string,
  "origin_class": string,
  "destination_class": string,
  "origin_hash": string,
  "destination_hash": string,
  "project_name": string,
  "project_id": integer,
  "event_arguments": [
    {
      "argument_value": string,
      "argument_type": string
    }
  ]
  "timestamp": string (in timestamp format)
}
```

2. Structural change

```
{
  "fetch_structure": string,
  "operation": string,
  "project_id": integer,
  "package_id": integer,
  "class_id": integer
}
```

Field	Value
fetch_structure	*project/package/class*
operation	*delete/change*
project_id	Always present
package_id	Present when change is on package or class
class_id	Present when change is on class

A.1.4 Installation

The following steps will setup the repository.

1. Clone the repository to the intended server machine.
2. `cd tese-repository`
3. Run the following sequence of commands.

```
docker-compose build

docker-compose run app rails db:create RAILS_ENV=production

docker-compose run app rails db:migrate RAILS_ENV=production

docker-compose up -d
```

Appendix A

Appendix B

Appendix B

B.1 tese-static

Repository for the java static analyzer used in the dissertation "A Software Repository for Live Software Development"

B.1.1 Description

This is one of the three components necessary for the correct functioning of the framework developed in "A Software Repository for Live Software Development". This component is responsible for the extraction of a structural representation of the Java projects in the workspace. It was developed as a Eclipse Plugin and be installed in any Eclipse IDE with the Java Development Tools (JDT) installed.

This component is intended for communication with the repository in Repository and its information complemented by the tool in Execution Analyzer.

B.1.2 Installation

These are the steps to run/install this tool correctly.

1. Install the repository as instructed in Repository.

2. Set a system environment variable for LIVESD_SERVER with the server url. This is done by modifying the */etc/environment* file, adding a line like the following:

```
LIVESD_SERVER="http://0.0.0.0/"
```

3. Copy the *.jar* file in *exports/plugin* to the *dropins* folder in your eclipse root folder.

B.1.3 Usage

The plugin builds the projects' representations when the Eclipse IDE starts, and detects changes in the workspace automatically, sending them to the repository as they occur. That being said, if

there are any issues with the representation of the projects in the repository noticed by the user, use the button *Sample Menu* $\rightarrow$ *Process Source* to flush the structure of the whole project into the repository.

Appendix C

Appendix C

C.1 tese-runtime

Runtime metadata extractor for java projects

C.1.1 Description

This is one of the three components necessary for the correct functioning of the framework developed in "A Software Repository for Live Software Development". This component is responsible for the extraction of the execution logs of a Java project. It was developed as an *AspectJ* project to be linked to any project to analyze.

This component is intended for communication with the repository in Repository and its information complemented by the tool in Structural Analyzer.

C.1.2 Installation

These are the steps to run/install this tool correctly.

1. Install the repository as instructed in Repository.

2. Set a system environment variable for LIVESD_SERVER with the server url. This is done by modifying the */etc/environment* file, adding a line like the following:

```
LIVESD_SERVER="http://0.0.0.0/"
```

3. Install *AspectJ* into your Eclipse IDE.

4. Import this project to the workspace.

C.1.3 Usage

In order to use this to analyze a given project, follow these steps.

1 Add *AspectJ* capabilities to the project to analyze (Right click in project to be analyzed in Project Explorer → Configure → Convert to AspectJ Project).

2. Include analyzer project in the other project's aspect path (Right click in project to be analyzed in Project Explorer → Properties → AspectJ Build → Aspect Path tab → Add Project and select analyzer).

3. Run project to analyze.

It is also possible for a user to specify which classes/packages he wants to analyze. For this, the user should modify the MethodInvocation.aj file and insert the following snippet to the end of the pointcut:

`&& (<INSERT_WITHINS>)`

With `<INSERT_WITHINS>` being `within(<PACKAGE>.<CLASS>)` pointcuts, separated by `||` operators.

Example of possible added pointcuts:

```
1 \\ (within(maze.cli.CLInterface) || within(maze.logic))
```

Appendix D

Appendix D

D.1 Projects

D.1.1 GET

D.1.1.1 */projects/*

```
1  [
2    {
3      "id": integer,
4      "project_name": string,
5      "num_packages": integer,
6      "created_at": string,
7    "updated_at": string
8    },
9    ...
10 ]
```

D.1.1.2 */projects/:id*

With URL parameter *deep=false* or not set,

```
1  {
2    "id": integer,
3    "project_name": string,
4    "num_packages": integer,
5    "created_at": string,
6    "updated_at": string
7  }
```

With URL parameter *deep=true*

Appendix D

```
1 {
2 "allProjectData": [
3   {
4     "id": integer,
5     "project_name": string,
6     "num_packages": integer,
7     "packages": [
8       {
9         "id": integer,
10        "package_name": string,
11        "class_count": integer,
12        "has_subpackages": boolean,
13        "package_path": string,
14        "i_classes": [
15          {
16            "id": integer,
17            "class_name": string,
18            "qualified_name": string,
19            "method_count": integer,
20            "attribute_count": integer,
21            "lines_of_code": integer,
22            "class_hash": string,
23            "class_attributes": [
24              {
25                "id": integer,
26                "attribute_name": string,
27                "attribute_type": integer
28              },...
29            ],
30            "i_methods": [
31              {
32                "id": integer,
33                "method_name": string,
34                "start_of_method": integer,
35                "length_of_method": integer,
36                "lines_of_code": integer,
37                "return_type": string,
38                "argument_count": integer,
39                "key": string,
40                "arguments": [
41                  {
42                    "id": integer,
43                    "argument_name": string,
44                    "argument_type": string
45                  },...
46                ],
47                "method_invocations": [
48                  {
49                    "id": integer,
```

```
50                      "invocation": string
51                    },...
52                  ]
53                },...
54              ]
55            },...
56          ]
57        },...
58      ]
59    }
60  ]
61  }
```

D.1.2 POST

D.1.2.1 Request Body

```
1   {
2     "projectName": string,
3     "packages": [
4       {
5         "packageName": string,
6         "hasSubpackages": boolean,
7         "classes": [
8           {
9             "className": string,
10            "hash": string,
11            "linesOfCode": integer,
12            "qualifiedName": string,
13            "attributes": [
14              {
15                "name": string,
16                "type": string
17              }
18            ],
19            "methods": [
20              {
21                "methodName": string,
22                "key": string,
23                "startOfMethod": integer,
24                "lengthOfMethod": integer,
25                "linesOfCode": integer,
26                "returnType": string,
27                "arguments": [
28                  {
29                    "name": string,
30                    "type": string
```

```
                }
              ],
              "methodInvocations": [
                string
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

D.1.3 DELETE

D.1.3.1 */projects/:id*

HTTP code 204

D.2 Packages

D.2.1 GET

D.2.1.1 */packages/:id*

```
{
    "id": integer,
    "package_name": string,
    "class_count": integer,
    "has_subpackages": boolean,
    "package_path": string,
    "i_classes": [
      {
        "id": integer,
        "class_name": string,
        "qualified_name": string,
        "method_count": integer,
        "attribute_count": integer,
        "lines_of_code": integer,
        "class_hash": string,
        "class_attributes": [
          {
            "id": integer,
            "attribute_name": string,
            "attribute_type": integer
          },...
```

```
        ],
        "i_methods": [
          {
            "id": integer,
            "method_name": string,
            "start_of_method": integer,
            "length_of_method": integer,
            "lines_of_code": integer,
            "return_type": string,
            "argument_count": integer,
            "key": string,
            "arguments": [
              {
                "id": integer,
                "argument_name": string,
                "argument_type": string
              },...
            ],
            "method_invocations": [
              {
                "id": integer,
                "invocation": string
              },...
            ]
          },...
        ]
      },...
    ]
}
```

D.2.2 POST

D.2.2.1 */packages*

```
{
  "packageName": string,
  "hasSubpackages": boolean,
  "projectName": string,
  "classes": [
    {
      "className": string,
      "hash": string,
      "linesOfCode": integer,
      "qualifiedName": string,
      "attributes": [
        {
          "name": string,
```

```
          "type": string
        }
      ],
      "methods": [
        {
          "methodName": string,
          "key": string,
          "startOfMethod": integer,
          "lengthOfMethod": integer,
          "linesOfCode": integer,
          "returnType": string,
          "arguments": [
            {
              "name": string,
              "type": string
            }
          ],
          "methodInvocations": [
            string
          ]
        }
      ]
    }
  ]
}
```

D.2.3 DELETE

D.2.3.1 */packages/:id*

HTTP response code 204

D.3 Classes

D.3.1 GET

D.3.1.1 */i_classes/:id*

```
{
  "id": integer,
  "class_name": string,
  "qualified_name": string,
  "method_count": integer,
  "attribute_count": integer,
  "lines_of_code": integer,
  "class_hash": string,
```

```
  "class_attributes": [
    {
      "id": integer,
      "attribute_name": string,
      "attribute_type": integer
    },...
  ],
  "i_methods": [
    {
      "id": integer,
      "method_name": string,
      "start_of_method": integer,
      "length_of_method": integer,
      "lines_of_code": integer,
      "return_type": string,
      "argument_count": integer,
      "key": string,
      "arguments": [
        {
          "id": integer,
          "argument_name": string,
          "argument_type": string
        },...
      ],
      "method_invocations": [
        {
          "id": integer,
          "invocation": string
        },...
      ]
    },...
  ]
}
```

D.3.2 POST

D.3.2.1 */i_classes*

```
{
  "className": string,
  "hash": string,
  "linesOfCode": integer,
  "qualifiedName": string,
  "projectName": string,
  "packageName": string,
  "attributes": [
    {
```

```
10        "name": string,
11        "type": string
12      }
13    ],
14    "methods": [
15      {
16        "methodName": string,
17        "key": string,
18        "startOfMethod": integer,
19        "lengthOfMethod": integer,
20        "linesOfCode": integer,
21        "returnType": string,
22        "arguments": [
23          {
24            "name": string,
25            "type": string
26          }
27        ],
28        "methodInvocations": [
29          string
30        ]
31      }
32    ]
33  }
```

D.3.3 DELETE

D.3.3.1 */i_classes/:id*

HTTP response code 204

D.4 Events

D.4.1 GET

D.4.1.1 */events*

URL parameter *from* and *to* can be used to an interval using timestamps

```
1   [
2     {
3       "this": string,
4       "target": string,
5       "kind": string,
6       "signature": string,
7       "class": string,
8       "source_location": string,
```

```
      "origin_class": string,
      "destination_class": string,
      "origin_hash": string,
      "destination_hash": string,
      "project_name": string,
      "project_id": integer,
      "event_arguments": [
        {
          "argument_value": string,
          "argument_type": string
        }
      ]
      "timestamp": string (in timestamp format)
    }
  ]
```

D.4.2 POST

```
{
  "events": [
    {
      "this": string,
      "target": string,
      "kind": string,
      "signature": string,
      "class": string,
      "sourceLocation": string,
      "originClass": string,
      "destinationClass": string,
      "originHash": string,
      "destinationHash": string,
      "projectName": string,
      "projectId": integer,
      "arguments": [
        {
          "value": string,
          "type": string
        }
      ]
      "timestamp": string (in timestamp format)
    }
  ]
}
```

D.4.3 DELETE

D.4.3.1 */events*

HTTP Response code 204

D.5 WebSocket

Subscribe by sending

```
{ "command": "subscribe", "identifier": "{\"channel\": \"ControlChannel\"}"
}
```

to */event_stream*.

D.5.1 Events

```
{
  "this": string,
  "target": string,
  "kind": string,
  "signature": string,
  "class": string,
  "source_location": string,
  "origin_class": string,
  "destination_class": string,
  "origin_hash": string,
  "destination_hash": string,
  "project_name": string,
  "project_id": integer,
  "event_arguments": [
    {
      "argument_value": string,
      "argument_type": string
    }
  ]
  "timestamp": string (in timestamp format)
}
```

D.5.2 Structural Change Notification

```
{
  "fetch_structure": string,
  "operation": string,
  "project_id": integer,
```

```
5   "package_id": integer,
6   "class_id": integer
7 }
```

Appendix D

References

[AFC98] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 153–160, Jun 1998.

[BK] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *Proceedings of the 9th International Workshop on Program Comprehension*.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[BTDS13] C. Bartoszuk, G. Timoszuk, R. Dąbrowski, and K. Stencel. Magnify - a new tool for software visualization. In *2013 Federated Conference on Computer Science and Information Systems*, pages 1485–1488, Sept 2013.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998.

[CC90] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

[Cel17] Joe Celko. Handling graphs in sql. https://www.red-gate.com/simple-talk/sql/t-sql-programming/handling-graphs-sql/, Feb 2017.

[Chi00] Shigeru Chiba. Load-time structural reflection in java. In Elisa Bertino, editor, *ECOOP 2000 — Object-Oriented Programming*, pages 313–336, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[DPJM+02] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In Stephan Diehl, editor, *Software Visualization*, pages 151–162, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[DST11] Robert Dąbrowski, Krzysztof Stencel, and Grzegorz Timoszuk. Software is a directed multigraph. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, pages 360–369, Berlin, Heidelberg, 2011. Springer-Verlag.

[FCX13] J. Feng, B. Cui, and K. Xia. A code comparison algorithm based on ast for plagiarism detection. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 393–397, Sept 2013.

[FHS16] E. Fauzi, B. Hendradjaya, and W. D. Sunindyo. Reverse engineering of source code to sequence diagram using abstract syntax tree. In *2016 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, Oct 2016.

[FKO98] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Software: Practice and Experience*, 28(4):371–400, 1998.

[Flo06] Nuno Flores. Engenheria reversa de padrões em arquiteturas reutilizáveis. Master's thesis, MEI, FEUP, Porto, January 2006.

[FSWH16] M. D. Feist, E. A. Santos, I. Watts, and A. Hindle. Visualizing project evolution through abstract syntax tree analysis. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 11–20, Oct 2016.

[FWG07] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79, Oct 2007.

[GHJV95] 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.

[GLW06] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3d. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06, pages 47–56, New York, NY, USA, 2006. ACM.

[GO03] T. Gschwind and J. Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 259–268, March 2003.

[GS15] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, editors, *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122, Cham, 2015. Springer International Publishing.

[Gué04] Yann-Gaël Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, pages 28–41. IBM Press, 2004.

[HELD11] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM.

[HNM+97] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. Mdl: a language and compiler for dynamic program instrumentation. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 201–212, Nov 1997.

[Jon03] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

[jvm] Java™ virtual machine tool interface (jvm ti).

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[KM10] Holger M. Kienle and Hausi A. Müller. Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247 – 263, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[Kos03] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[LB14] F. M. Lazar and O. Banias. Clone detection algorithm based on the abstract syntax tree approach. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 73–78, May 2014.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, September 2003.

[LDGP05] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler: An information visualization tool for program comprehension. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 672–673, New York, NY, USA, 2005. ACM.

[LS95] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. *SIGPLAN Not.*, 30(6):291–300, June 1995.

[LZ97] Han Bok Lee and Benjamin G. Zorn. Bit: A tool for instrumenting java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, pages 7–7, Berkeley, CA, USA, 1997. USENIX Association.

[ML02] Tom Mens and Michele Lanza. A graph-based metamodel for object-oriented software metrics. In *Electronic Notes on Theoretical Computer Science*, volume 72, 202.

[NS14] K. Nakayama and E. Sakai. Source code pattern as anchored abstract syntax tree. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pages 170–173, June 2014.

[OJH03] Alessandro Orso, James Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 67–ff, New York, NY, USA, 2003. ACM.

[OST04] T. Okamura, B. Shizuki, and J. Tanaka. Execution visualization and debugging in three-dimensional visual programming. In *Proceedings. Eighth International Conference on Information Visualisation, 2004. IV 2004.*, pages 167–172, July 2004.

[RG03] Mark Richters and Martin Gogolla. Aspect-oriented monitoring of uml and ocl constraints. In *In AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML*, 2003.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition).* Pearson Higher Education, 2004.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005.

[SJ15] A. Sadar and Vinitha Panicker J. Doctool - a tool for visualizing software projects using graph database. In *2015 Eighth International Conference on Contemporary Computing (IC3)*, pages 439–442, Aug 2015.

[SO06] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[STA09] STAN. Stan: Structure analysis for java, 2009.

[Tan13] Steven L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press.

[TC09] A. R. Teyseyre and M. R. Campo. An overview of 3d software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, Jan 2009.

[TCSH06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, Nov 2006.

[TGHB13] G. Tao, D. Guowei, Q. Hu, and C. Baojiang. Improved plagiarism detection algorithm based on abstract syntax tree. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 714–719, Sept 2013.

[vdVvdWM12] J. S. van der Veen, B. van der Waaij, and R. J. Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 431–438, June 2012.

[WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference*

on Software Engineering, ICSE '11, pages 551–560, New York, NY, USA, 2011. ACM.

[WY05] Daniel G. Waddington and Bin Yao. High-fidelity c/c++ code transformation. *Electronic Notes in Theoretical Computer Science*, 141(4):35 – 56, 2005. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

[YPZ12] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 102–112, June 2012.