Scalability Solutions for Program Comprehension Through Dynamic Analysis

Andy Zaidman

Universiteit Antwerpen

Promotor: prof. dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van

Doctor in de Wetenschappen

Acknowledgements

First and foremost, the support of my family has been instrumental. I could not have done it without every one of you. A very special thank you goes to Wendy, my wife and best friend, for always being there and supporting me.

Over the years prof. Demeyer has given me carte blanche when it comes to doing my research, yet was always there to provide me with precious advice and the necessary support at times when there were more questions than answers. Serge, thank you very much!

Every day, Filip Van Rysselberghe, my office mate, was there to cheer me up, provide answers to unanswerable questions and... provide music. For all other questions, Bart Du Bois, Hans Stenten, Niels Van Eetvelde, Pieter Van Gorp, Hans Schippers, Bart Van Rompaey, Matthias Rieger and Olaf Muliawan were always ready to provide their opinions and insights. Many other people from the department were always in for a chat when research was going slow and helped me keep my spirits high. Thank you *all* for providing such a pleasant atmosphere.

I had the opportunity to be able to enjoy a number of collaborations with amongst others Bram Adams, Toon Calders, Kris De Schutter, Orla Greevy, Wahab Hamou-Lhadj and Kim Mens. To them I say: keep up the good work!

A sincere word of gratitude goes out to the members of my doctoral jury, who, through their extensive reviews, have helped this dissertation become significantly better. Thank you Chris Blondia, Serge Demeyer, Theo D'Hondt, Stéphane Ducasse, Dirk Janssens, Jan Paredaens and Tarja Systä.

To all my friends that kept spurring me on: I hope I can one day repay you guys.

And finally to everyone who over the years has played a role — small or big — in this dissertation and the work leading up to it: I will not forget and you will not be forgotten...

Abstract

Dynamic analysis has long been a subject of study in the context of (compiler) optimization, program comprehension, test coverage, etc. Ever-since, the scale of the event trace has been an important issue. This scalability issue finds its limits on the computational front, where time and/or space complexity of algorithms become too large to be handled by a computer, but also on the cognitive front, where the results presented to the user become too large to be easily understood.

This research focusses on delivering a number of dynamic analysis based program comprehension solutions that help software engineers to focus on the software system during their initial program exploration and comprehension phases.

The key concepts we use in our techniques are *frequency of execution* and *runtime coupling*. Both techniques deliver a solution which can help the software engineer bring focus into his or her comprehension process by annotating parts of the trace that contain similarities or by bringing out the key concepts (classes) of a system. To validate our techniques we used a number of open-source software systems, as well as an industrial legacy application.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

In the beginning the Universe was created. This has made a lot of people very angry and has been widely regarded as a bad move.

—Douglas Adams

Greenfield software development is fun! Building software systems from scratch allows you to be your creative self. You and your team are able to define a whole range of parameters, whether it be the global architecture of the system, some fancy design quirks, the choice of technologies, etc. Sometimes even, the programming language to be used can be your favorite one.

However, in a world were everything changes at a seemingly continually increasing rate, software has to keep up with the changing environment in order to stay useful and keep fulfilling the expectations one has of it. When software that is already in place needs to change, the software development team is confronted with a whole new set of challenges when compared to the greenfield software development situation.

One of these challenges — and probably the most time-consuming one — is trying to understand the existing piece of software, a discipline which is termed "program comprehension". Typically, this process is aided by the available documentation, however, often this documentation is either non-existent or out-dated. This dissertation is about providing solutions to be used in this situation, specifically to be used during the initial phases of the program comprehension process. The solutions we present are based upon dynamic analysis, or the analysis of data gathered during the execution of a software system.

1.1 Context

Legacy software is software that is still very much useful to an organization – quite often even *indispensable* – but the evolution of which becomes too great a burden [Bennett, 1995, Demeyer et al., 2003]. Legacy software is omni-present: think of the large software systems that were designed and first put to use in the 1960's or 1970's; these software systems are nowadays often the backbone of large multinational corporations. For banks, healthcare institutions, etc. these systems are vital for their daily operations. As such, failure of these software systems is not an option and that is why these trusted "oldtimers" are still cared for every day. Furthermore, they are still being evolved to keep up with the current and future business requirements.

We propose to use this definition of Brodie and Stonebraker [Brodie and Stonebraker, 1995], which gives an apt description of a legacy system:

> "*Any information system that significantly resists modification and evolution to meet new and constantly changing business requirements.*"

Note that this definition implies that age is no criterion when considering whether a system is a legacy system [Demeyer et al., 2003].

As an example from the world of banking, we still see data formats in use today which have been defined decades ago. Access to that data is being provided through special applications or modules. These have now become legacy systems, but if any of these were to fail, a downtime of a day or two can mean bankruptcy for the company in question.

To make things worse, evolving a system can exaggerate the legacy problem. To paraphrase Lehman, an evolving system increases its complexity, unless work is done to reduce it [Lehman and Belady, 1985]. This increase in complexity is further enlarged when the original developers, experienced maintainers or up-to-date documentation are not available [Sneed, 2005, Brodie and Stonebraker, 1995, Moise and Wong, 2003, de Oca and Carver, 1998, Demeyer et al., 2003]. A number of solutions to cope with evolution have been proposed in the field of software reengineering [Chikofsky and Cross II, 1990, Bennett, 1995, Sneed, 1996].

1.2 The modalities of change

When one wants to apply countermeasures to stabilize or reduce complexity, the software engineer would ideally like to have a *deep insight* into the application when starting his/her reengineering (or better still refactoring)

operation [Sneed, 2004, de Oca and Carver, 1998, Lehman, 1998]. Yet this understanding is often found lacking as, over time, legacy applications become something of magical black boxes. For one, there is the "if it ain't broke, don't fix it" attitude which often gets in the way. Secondly, there is the problem of out-of-sync documentation, which hinders program comprehension for maintainers and new developers alike [Chikofsky and Cross II, 1990, Moise and Wong, 2003].

Nevertheless, this insight is certainly necessary to be able to apply these countermeasures reliably, economically and promptly. Going beyond applying countermeasures to stabilize or reduce complexity, is the need to integrate software systems that were originally not conceived to work together. The 1990s were characterized by an increase in spending on information technology, partly due to the so-called "dot com bubble". During this period, the problem on integrating standalone applications became known under the flag of *Enterprise Application Integration* or *EAI* [Linthicum, 1999]. During the early 2000s, the community shifted its focus towards more loosely connected components and the problem of integrating software systems became known as building up a *Service Oriented Architecture* or *SOA* [Gold et al., 2004].

As such, apart from a status-quo scenario, in which the business has to adapt to the software that resists change, a number of scenarios are frequently seen [Bennett, 1995]:

1. Rewrite the application from scratch, from the legacy environment, to the desired one, using a new set of requirements.
2. Reverse engineer the application and perform a rewrite of the application from scratch, from the legacy environment, to the desired one.
3. Refactor the application. One can refactor the old application, without migrating it, so that change requests can be efficiently implemented; or refactor it to migrate it to a different platform.
4. Often, in an attempt to limit the costs, the old application is "wrapped" and becomes a component in, or a service for, a new software system. In this scenario, the software still delivers its useful functionality, with the flexibility of a new environment. This works fine and the fact that the old software is still present is slowly forgotten. This leads to a phenomenon which can be called the *black-box syndrome*: the old application, now component or service in the new system, is trusted for what it does, but nobody knows – or wants to know – what goes on internally.
5. A last possibility is a mix of the previous options, in which the old application is seriously changed before being set-up as a component or service in the new environment.

Intuitively, it is clear that for scenarios 2 through 5, a certain level of

insight into the existing application is necessary before reengineering can safely begin. This is where the discipline of program comprehension comes in.

1.3 Program comprehension

When programming a piece of software, the programmer has to build a mental bridge that connects the code he/she is writing and the program behavior he/she is trying to accomplish [Renieris and Reiss, 1999]. Conversely, when a programmer is trying to gain an understanding of a system, he is actually trying to get the reverse mapping: from the functionality that is present in the system to the code that is performing that very functionality.

Depending on the source, literature suggests that between 30 and 60% of a software engineer's time is spent in the program comprehension phase, where one has to study existing code, documentation and other design artifacts in order to get an adequate understanding of a software system [Spinellis, 2003, Wilde, 1994, Corbi, 1990]. Adequate being the level where the programmer feels comfortable that his change operation(s) will not harm the system's architecture, design or functionality in a bad way.

The manner in which a programmer builds up his understanding of a software system varies greatly. It is mostly dependent on the individual, but can be influenced by the magnitude of the system under study, the level of understanding needed for the task at hand, the programming language, the familiarity with the system under study, previous experiences in the domain, etc. [Lakhotia, 1993, von Mayrhauser and Vans, 1995]. While in theory it is necessary to understand the entire system before making any changes, in practice it is essential to use a *goal oriented* or *as-needed strategy*: you want to get an understanding of the part of the system that you are specifically interested in with regard to the task at hand. Furthermore, due to economical constraints this should happen both quickly and thoroughly [Lukoit et al., 2000].

Realizing that program comprehension is such an important part of every software engineer's life, we wonder how we can provide stimuli to make the comprehension process more efficient.

1.4 Lack of documentation

When focussing on delivering a program comprehension solution, we do make the assumption that the program comprehension process happens without

adequate documentation being available. A number of factors make us believe that for many software systems this assumption is more often than not a reality:

- A lot of the knowledge of an application is not written down in documentation, but is present in the heads of the developers. The information technology (IT) business, being a sector with a high degree of personnel volatility, is certainly at risk of losing a lot of this implicit knowledge about their software systems when personnel is leaving the project or leaving the company altogether [Chikofsky and Cross II, 1990].
- As we already mentioned, software systems have to evolve. This evolution can cause a drift away from the original architecture [Mens, 2000]. Furthermore, keeping the documentation synchronized with those evolutionary changes does not always happen in a structured way [Chikofsky and Cross II, 1990, Moise and Wong, 2003].
- From our own experiences with industry we also know that the software system's documentation is often not up to date. More on this can be found in Chapter 9.

1.5 Dynamic analysis

Dynamic analysis is the study of running software with the aim of extracting properties of the software system. Typically, software is run according to an execution scenario and run-time information is stored in a so-called *execution trace*. Opposite to this dynamic approach stands the concept of static analysis, which extracts software system properties from artifacts such as source code, documentation, architectural diagrams or design information. Within this research, dynamic analysis is the basic means by which we want to stimulate the program comprehension process.

Dynamic analysis has long been a subject of study in the context of (compiler) optimization, test coverage, etc. It has also been extensively researched for program comprehension purposes, sometimes in a pure dynamic analysis context, sometimes in a mixed static-dynamic context [Ball, 1999, Eisenbarth et al., 2001, Richner, 2002, Systä, 2000a, Sayyad-Shirabad et al., 1997, El-Ramly et al., 2002, Jerding and Rugaber, 1997, Gschwind et al., 2003, Greevy and Ducasse, 2005, Hamou-Lhadj et al., 2005, Hamou-Lhadj et al., 2004]. Although results have been encouraging, the problem of scalability has been recognized as an important stumbling point [Larus, 1993, Smith and Korel, 2000]. In the context of using dynamic analysis for program comprehension purposes, this problem of scalability has three major components, namely:

- A computational component, where the scalability of the underlying algorithm for the program comprehension tool is important in order to make sure that the results (1) can be computed and (2) can be computed in a reasonable amount of time [Larus, 1993].
- A visual component, where the resultset has to be scalable to make it easy for the user to interpret the results [Jerding and Rugaber, 1997, Jerding and Stasko, 1998, Jerding et al., 1997].
- A cognitive component, where the resultset presented to the end user should be of an acceptable size so that an information overload of the end user can be avoided [Zayour and Lethbridge, 2001].

As such, within the research presented in this dissertation, we will put an emphasis on the scalability of the techniques we propose. This scalability is mainly concentrated around developing scalable algorithms and providing concise resultsets.

1.6 Hypothesis

> *Run-time coupling measures* and *relative frequency of execution* are two axes in the run-time information-space that can help us build heuristics for program comprehension purposes. Furthermore, these two axes allow to build in scalability, both at the level of the algorithm and at the level of the resultset.
>
> - Run-time coupling measures allow us to identify must-be-understood classes in the software system.
> - Frequency of execution allows us to identify regions of the trace that are highly repetitive.

1.7 Solution space

As we have already mentioned, there is a clear emphasis on scalability for the techniques that we have developed. To be more precise, we have defined two heuristical techniques that allow a huge event trace to be reduced to a more abstract representation that is presented to the user. We will now briefly introduce these two heuristics.

1.7.1 Run-time coupling based heuristic

The basic idea behind using coupling is the fact that structural dependencies between modules of a system can indicate modules that are interesting for initial program comprehension [Robillard, 2005]. As a measure we use run-time export coupling, which — provided we have a well-covering execution scenario — gives us all actual dependencies that happen at runtime. Modules which exhibit a high level of export coupling request other modules to do work for them (delegation) and often contain important control structures.

Coupling measures however are typically between two classes or modules, whereas we want to take into consideration the complete structural topology of the application. To overcome this strict binary relation between modules, we add a transitive measurement for reasoning over the topology. We use webmining techniques for this [Zaidman et al., 2005]. *Webmining*, a branch of datamining research, analyzes the topological structure of the web trying to identify important web pages based solely on their hyperlink structure. By interpreting call graphs as web graphs, we port this technique so that we are able to retrieve *important* classes. An important class being a class that needs to be understood early on in the program comprehension process in order to understand other classes and the interactions between these classes.

The resultset obtained from this heuristic is a list of all the classes/modules of which containing procedures were executed during the scenario. These classes/ modules are ranked from being important to being irrelevant during early program comprehension phases. To validate our approach we used two open source case studies, namely *Apache Ant 1.6.1* and *Jakarta JMeter 2.0.1*. The actual validation was done by comparing the results obtained to extensive design documentation that was publicly available.

Furthermore, we applied this heuristic on an industrial legacy C system. In contrast to the open source case studies where we had to rely on documentation available on the internet, we were now able to validate the results we obtained with the original developers and current maintainers of the application. The results of this industrial experiment confirm the value of this technique.

We expand upon the aforementioned techniques in Part II of this dissertation. In Part IV we report on our industrial experiences with this approach.

1.7.2 Frequency based heuristic

Thomas Ball [Ball, 1999] introduced the concept of *"Frequency Spectrum Analysis"*, a way to correlate procedures, functions and/or methods through their relative calling frequency. The idea is based around the observation that

a relatively small number of methods/procedures is responsible for a huge event trace. As such, a lot of repeated calling of procedures happens during the execution of the program. By trying to correlate these frequencies, we can learn something about (1) the size of the inputset, (2) the size of the outputset and —most interesting for us— (3) calling relationships between methods/procedures.

We build further upon this idea, by proposing a visualization of the trace that allows for visual detection of parts of the event trace that show tightly collaborating methods. We applied this technique on two open source case studies, namely *Apache Tomcat 4.1.18* and *Fujaba 4.0*.

The visualization we propose resembles a "*heartbeat*" as seen on an *ECG* or electrocardiogram and should be interpreted in a similar way. For regions in the trace where tightly collaborating methods are executed, the visualization shows a very regular pattern, like a ECG of a heart that is "in rest". Regions in the trace where the collaboration between methods is less tight are visualized much more erratically. This distinction can help the software engineer concentrate on those parts of the trace that he is particularly interested in.

We expand upon this visualization in Part III of this dissertation and show a variation of this approach that we used in the industrial application that is discussed in Part IV.

1.7.3 Research contributions

The major research contributions of this thesis are:

- A technique to identify key classes during early program comprehension phases [Zaidman et al., 2005, Zaidman et al., 2006b].
- A technique to visualize execution traces and identify similar parts in the execution [Zaidman and Demeyer, 2004].
- A large-scale industrial case studies to evaluate the scalability of the aforementioned techniques [Zaidman et al., 2006a].

1.8 Academic context

The research presented in this dissertation has been performed within the *Lab On ReEngineering* (LORE) research group, part of the University of Antwerp. In a broader sense, this research has been carried out in the context of the ARRIBA project. ARRIBA is short for *Architectural Resources for the Restructuring and Integration of Business Applications* and its aim is

to provide a methodology and tools to support the integration of business applications that have not necessarily been designed to coexist[1].

The ARRIBA project team consists out of a team of researchers from the Free University of Brussels (*Vrije Universiteit Brussel — VUB*), the Ghent University and the University of Antwerp. Furthermore, a number of companies are involved in ARRIBA. These industrial partners are (1) responsible for checking whether the research that is carried out by the academic partners is relevant in an industrial context and (2) they are able to provide case studies to the academic partners in order to validate the research prototypes. Although the group of companies has changed during the duration of the ARRIBA project (2002 – 2006), the following companies form the backbone of the industrial committee:

- Inno.com
  An ICT expertise center dedicated to advise and assist its clients and partners to cope with their most challenging technology and business issues (www.inno.com).
- Banksys
  Manages the Belgian network for debit card transactions (www.banksys.be).
- Anubex
  An expert in application modernisation through software conversion and application migration (www.anubex.com).
- Christelijke Mutualiteit
  A Belgian social security provider (www.cm.be).
- KAVA
  A non-profit organization grouping over a thousand Flemish pharmacists (www.kava.be).
- KBC
  A banking and insurance company (www.kbc.be).
- Toyota Motor Europe
  European branch of the Toyota Motor company (www.toyota.be).

The ARRIBA project is sponsored by the IWT Flanders[2] within the 2002 call of the GBOU program.

1.9 Organization of this dissertation

The organization of this dissertation is as follows.

[1]For more information about this project, please visit: http://arriba.vub.ac.be

[2]Institute for the Promotion of Innovation by Science and Technology in Flanders. For more information, see: http://www.iwt.be

Chapter 2 gives a view on program understanding and introduces a number of theories and models pertaining to program comprehension. We also position our research within the existing program comprehension frameworks. Chapter 3 talks about advantages and disadvantages of dynamic analysis. We provide an overview of techniques that enable dynamic analysis and discuss the ones that we have used during our research in some more detail.

Part II of this dissertation deals with a program comprehension solution that is based on coupling. In Chapter 4 we introduce some of the concepts concerning coupling and we relate coupling to program comprehension. Chapter 4 also introduces the dynamic coupling framework we use for our research, while in Chapter 5 we explain why it is important to also take into account indirect coupling for the purpose of program comprehension. Here we also explain how webmining can help us take into account this indirect coupling. Chapter 6 introduces the experimental setup we use to validate our hypothesis and presents the results we have obtained from applying our techniques on a set of open source case studies. Chapter 7 then describes a control experiment in which we compare the results we have obtained through dynamic analysis with results from a similar experiment carried out with static analysis.

Part III deals with frequency analysis. Chapter 8 describes our experiences of retrieving clues that can speed up the program comprehension process when taking into account the relative frequency of execution of methods or procedures.

Part IV deals with our industrial experiences regarding the techniques we have developed. As such, Chapter 9 showcases our experiences with applying both the coupling based and frequency based program comprehension solutions in an industrial legacy C context. We present the results we have obtained and also discuss a number of typical pitfalls that occur when applying dynamic analysis in a legacy context.

Chapter 10 then gives an overview of related work and Chapter 11 concludes this dissertation with a discussion about our contributions and some pointers to future work.

Chapter 2

Program comprehension

All truths are easy to understand once they are discovered; the point is to discover them.

— Galileo Galilei

This chapter tries to capture what program comprehension is. We provide a definition and determine in which circumstances a software engineer needs to understand a program. Furthermore, we discuss a number of program comprehension models and we discuss which factors can influence the choice of program comprehension model.

2.1 What is program comprehension?

When a person starts to build up an understanding of a previously unknown software system or a portion thereof, he/she must create an informal, human oriented expression of computational intent. The creation of this expression happens through a process of analysis, experimentation, guessing and puzzle-like assembly [Biggerstaff et al., 1993].

When it comes to a definition of what program comprehension means, we adhere to the definition introduced by Biggerstaff et Al [Biggerstaff et al., 1993]:

> *"A person understands a program when able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."*

As such, from this definition we learn that the program comprehension process is closely related to the *concept assignment problem*. This is the problem of discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given program [Biggerstaff et al., 1993]. From this point of view, it becomes clear that the comprehension process is a highly individual process, where results can vary from person to person, while still understanding the software system in the same way.

In this chapter we will first try to explain why program understanding is necessary in the field of reengineering as a whole (Section 2.2), after which we will discuss a number of program comprehension theories in Section 2.3.

2.2 Program understanding as a prerequisite

Program understanding is a necessary prerequisite to many software engineering activities. Von Mayrhauser and Vans have made a compilation of software maintenance specific scenarios in which program comprehension is a necessary prerequisite [von Mayrhauser and Vans, 1995]. Table 2.1 provides an overview of these maintenance activities.

From Table 2.1 it becomes clear that most day-to-day software maintenance activities require a certain level of insight into the application to be maintained. Being aware of the fact that almost all software evolution activities require understanding of the software system, makes the link between software evolution and program understanding become very clear.

Furthermore, knowing that most reengineering operations require a program comprehension phase and knowing that up to 60% of the software engineer's time can be spent in this phase (see Section 1.3) [Spinellis, 2003,Wilde, 1994,Corbi, 1990], improving the efficiency of this phase can mean a significant overall efficiency gain.

2.3 Program comprehension models

In the introduction we have already mentioned that program understanding is a highly individual activity. A number of factors influence *how* a software engineer goes about his/her program understanding process, i.e. which strategy he/she will follow. Some of these — sometimes very subjective — factors are [von Mayrhauser and Vans, 1995]:

- the level of experience of the software engineer
- the level of familiarity with the problem domain

Maintenance tasks	Activities
Adaptive	Understand system Define adaptation requirements Develop preliminary and detailed adaptation design Code changes Debug Regression tests
Perfective	Understand system Diagnosis and requirements definition for improvements Develop preliminary and detailed perfective design Code changes/additions Debug Regression tests
Corrective	Understand system Generate/evaluate hypotheses concerning problem Repair code Regression tests
Reuse	Understand problem, find solution based on close fit with reusable components Locate components Integrate components
Code leverage	Understand problem, find solution based on predefined components Reconfigure solution to increase likelihood of using predefined components Obtain and modify predefined components Integrate modified components

Table 2.1: Tasks and activities requiring code understanding.

- the level of familiarity with the solution space
- the complexity of the software system's structure
- the amount of time available

Studies that lie on the border between psychology and computer science have shown that many strategies exist for the program comprehension process. These strategies can roughly be divided into three models of program comprehension, namely: the top-down model, the bottom-up model or a mix of the previous two, the so-called integrated model [von Mayrhauser and Vans, 1995]. The next few sections will discuss each of these models.

2.3.1 Top-down program comprehension models

Top-down understanding typically applies when the code, problem domain and/ or solution space are familiar to the software engineer [von Mayrhauser

and Vans, 1995]. This stems from the idea that when a software engineer has already mastered code that performed the same or similar tasks, the structure of the code will have parallels. These similarities in code structure are easier to recognize in a top-down understanding process.

When a software engineer goes about his program understanding in a top-down strategy, he/she usually already has a hypothesis or a number of hypotheses about the structure of the system. These hypotheses can come from previous experiences relating to software in the same domain, using similar technologies, etc. or from *beacons* in the software's code, documentation or other artifacts. In program comprehension terminology a beacon is a cue that indexes into knowledge, e.g. triggering a memory from a previously seen construct and associating it with the current solution. In software engineering terminology a good example of such a beacon can be a design pattern [Gamma et al., 1995], e.g. an MVC (Model View Controller) pattern, that would give an indication as to how the GUI layer is structured.

When using this top-down program comprehension strategy, a mental model is built throughout the process that successively refines hypotheses and auxiliary hypotheses about the software system. Hypotheses are iteratively refined, passing several levels until they can be matched to specific code in the program (or a related document, e.g. a configuration file) [von Mayrhauser and Vans, 1995].

2.3.2 Bottom-up program comprehension models

When the code and/or problem domain are not familiar to the software engineer, bottom-up understanding is frequently chosen. This section describes the models that are used in this situation.

Program model Pennington [Pennington, 1987] found that when code is completely new to programmers, the first mental representation they build is a control-flow program abstraction called the program model. This representation, built from the bottom up via beacons, identifies elementary blocks of code in the program. The program model is created via the *chunking* of microstructures into macrostructures and via *cross-referencing*. Chunking is about creating larger entities from small blocks to reason with, while cross-referencing relates these larger entities to a higher level of abstraction. As an example we could say that all the classes that work together to create a linked list can be chunked together, while then actually designating it as a "linked list" and understanding its purpose (i.e. being a container structure) is cross-referencing it to a higher level of abstraction.

Situation model A second model that Pennington identified is the situation model [Pennington, 1987]. This model also operates in a bottom-up fashion and creates a dataflow/functional abstraction. The application of this model requires knowledge of the real-world domains that are present in the software system. An example of this type of bottom-up comprehension is relating `clothesInventory = clothesInventory - itemsSold` to "reducing the inventory of clothes by the number of items sold". Again, lower order situation knowledge can be chunked into higher order situation knowledge. The situation model is complete once the program goal is reached.

2.3.3 Integrated model

An integrated model for code comprehension involves the top-down strategy, bottom-up strategy (both the program and the situation model) and a knowledge base. The knowledge base, which typically is the human mind, stores (1) any new information that is obtained directly from the application of either of the two program comprehension strategies or (2) information that is inferred.

Intuitively, one would think that in practice the integrated model is most commonly used when trying to understand large scale systems. The reasoning behind this is that certain parts of the code may be familiar to the software engineer because of previous experiences and other parts of the code may be completely new. Experiments by Von Mayrhauser confirm this intuition [von Mayrhauser and Vans, 1994].

2.4 On the influence of comprehension tools

Storey et al describe an experiment in which they study the behavior of 30 participants when using program comprehension tools [Storey et al., 2000]. More precisely, they observe the factors that influence the participant's choice of program comprehension strategy.

Their conclusion was that, ideally, the tools supporting the program understanding process should facilitate the programmer's preferred strategy or strategies, rather then enforce the use of a fixed strategy [Storey et al., 2000]. Approaches missing features to optimally use a strategy often meant switching to another strategy, hindering the comprehension process. Being able to seamlessly switch between strategies was seen as a bonus.

Based on these observations, we will keep a serious eye on whether the techniques that we propose do not force the user to use a specific program comprehension strategy.

Chapter 3

Dynamic Analysis

It requires a very unusual mind to undertake the analysis of the obvious.

— Alfred North Whitehead

Choosing a basic means to reach a goal usually implies that the means adds some interesting properties to reach your goal. The means we propose is dynamic analysis. As such, we advocate the use of our means in the light of object-oriented software, where polymorphism and late-binding make a program hard to understand statically. Another benefit of using dynamic analysis is that you are able to follow an as-needed strategy during program comprehension. We also provide an overview of a number of dynamic analysis enabling technologies.

3.1 Definition

> In software engineering, dynamic analysis is the investigation of the properties of a software system using information gathered from the system as it runs.

We propose to use the above definition of *dynamic analysis*. It purposely remains quite vague as to not put a bias on the type of dynamic information that is collected or the kind of dynamic analysis that is executed. In other words, it remains sufficiently broad so that it can be used for program comprehension purposes, to collect design or performance metrics, etc.

Opposite to dynamic analysis stands the concept of static analysis, which collects its information from artifacts such as the source code, design documents, configuration files, etc. in order to investigate the system's properties.

Again, we remain quite vague on what kind of properties we want to investigate, as in the most general sense, these properties can for example be structural, behavioral, quality or performance oriented.

Enabling dynamic analysis in a software (re)engineering context requires the generation of an execution trace of the software system under study; the *execution trace* being the structure in which the gathered information is stored. To obtain this execution trace, one needs to execute the software system according to a well-defined execution scenario. An execution scenario being an instance of one or several use cases [Jacobson, 1995].

3.2 Why dynamic analysis?

The choice of dynamic analysis with regard to this research is inspired by two factors, firstly because dynamic analysis allows a very *goal-oriented* approach, meaning that we only have to analyze those parts of the application that we are really interested in and, secondly, because dynamic analysis is much more succinct at handling polymorphism, which is abundantly present in modern object-oriented software systems.

Within this section, we will briefly touch both factors.

3.2.1 Goal oriented strategy

Dynamic analysis allows to follow a very goal oriented (or as-needed) strategy when it comes to dealing with unknown software systems. When one only has end-user knowledge of the system available, it becomes relatively easy to only exercise those scenarios from the use cases that pertain directly to the knowledge that one wants to gather. This results in a smaller, more to-the-point execution trace and as a consequence it can also lead to better analysis-results.

Example. One wants to build up knowledge of how a word-processor like Microsoft Word functions internally when changing the properties of some piece of text that is selected. When one uses dynamic analysis for this, one could execute only those use cases that directly involve the selection of text and the subsequent property-change of that text, e.g. put the text in bold. If one were to use a less goal-oriented strategy, e.g. by a very broadly defined execution scenario or through static analysis, one would need to understand

a lot more of the application before knowing which parts exactly are related to the functionality one is trying to understand.

3.2.2 Polymorphism

Polymorphism is the ability of objects belonging to different classes of the same class-hierarchy to respond to method calls of methods with the same name, in a type-specific way, i.e. with different behavior. Furthermore, the programmer does not have to know the exact class of the object in advance, so the class and its resulting behavior can be decided at run time. This gives rise to the notion of *late binding*, deciding at runtime which behavior will be executed for a certain object.

The mechanism of polymorphism allows to write programs more efficiently. Furthermore, it also should allow software to be more easily evolvable. However, for program comprehension purposes, polymorphism can complicate matters as it becomes difficult to grasp the precise behavior of the application, without witnessing the software system in action. This is because one possibly polymorphic call is a variation point that can give rise to a great number of different behaviors (the number of possible behaviors is equal to the number of classes present in the class-hierarchy below the statically defined class type plus one). To illustrate this, we know of a class hierarchy in Ant, a Java build tool, where the class `Task` has more than 100 child-classes, each portraying specific command-line tasks that can possibly be executed.

In contrast, when looking at a software system with the help of dynamic analysis however, the view obtained from the software is precise with regard to the execution scenario. The behavior that is called upon is specific to the functionality exercised and as such, the number of possible variations is diminished up to the point that all variations are actually executed (and not only theoretically possible).

3.3 Modalities of dynamic analysis

In this section we will give you an example of an execution trace and we will briefly touch a number of technologies that enable the extraction of an execution trace from a running software system. Furthermore, we will discuss some details of the implementations we used during our experiments.

3.3.1 Example execution trace

Figure 3.1 shows an example of a small piece of trace obtained from running JHotDraw[1], a small paint-like application written in Java.

```
CALL org.jhotdraw.application.DrawApplication::<clinit> ( ()V )
EXIT org.jhotdraw.application.DrawApplication::<clinit> ( ()V )
CALL org.jhotdraw.samples.javadraw.JavaDrawApp::<clinit> ( ()V )
EXIT org.jhotdraw.samples.javadraw.JavaDrawApp::<clinit> ( ()V )
CALL org.jhotdraw.samples.javadraw.JavaDrawApp::main ( ([String;)V )
CALL org.jhotdraw.samples.javadraw.JavaDrawApp::<init> ( ()V )
CALL org.jhotdraw.contrib.MDI_DrawApplication::<init> ( (String;)V )
CALL org.jhotdraw.application.DrawApplication::<init> ( (String;)V )
...
```

This fragment of an execution trace contains all non-library methods that happen during a typical run of JHotDraw. To be more precise, each *call to* and *return from* a method is recorded, which allows us to retrieve all calling relations and nesting depth of calls. Consider for example the last entry in the execution trace above: we record the originating package (e.g. `org.jhotdraw.application`), the classname (e.g. `DrawApplication`), the methodname (e.g. `<init>`, which stands for the constructor) and its parameters and return type (e.g. parameter `String` and return type `void`).

Figure 3.1: Example execution trace

3.3.2 Trace extraction technologies

Profiler or debugger based tracing. A *profiler* is typically used to investigate the performance or memory requirements of a software system. A *debugger* on the other hand is frequently used to step through a software system at a very fine grained level in order to uncover the reasons for unanticipated behavior.

Typically, profiler and debugger infrastructures of virtual machines or other environments (sometimes even the operating system itself) send out events at certain stages of the execution. One can then write a plugin to the virtual machine or the environment in order to capture these events and act upon them, e.g. store them in an execution trace. Typical events that can be caught with a profiler or debugger are the invocation of a method/procedure, the return from a method/procedure, access to variables, fields, etc.

[1]For more information, see: http://www.jhotdraw.org/

Aspect-oriented based tracing. Aspect-oriented programming (AOP) introduces a new program entity, called an *aspect* [Kiczales et al., 1997]. This aspect can be used to isolate a so-called cross-cutting concern, or a concern that is present in many classes or modules and does not strictly belong to any of the classes or modules concerned. The code that is responsible for such a concern can be captured in the *advice* part of the aspect, while the *pointcut* part of the aspect specifies where to insert the particular piece of code.

As such, AOP allows to insert a piece of code at the beginning or at the end of a method/procedure. This makes it possible to write a so-called tracing aspect, an aspect that generates an entry in the execution trace every time a method call or a method return takes place.

AST rewriting based tracing. When parsing the source code of a software system, alterations can be made to the abstract syntax tree (AST) before outputting the AST again as normal source code. To our knowledge, no standard approach exists for this AST rewriting. An example of such an approach is the work of Akers [Akers, 2005]. Please also note that some AOP implementations work in a very similar way, where the aspect weaver is built on top of an AST rewriting mechanism [Zaidman et al., 2006a].

Method wrapper based tracing. Method wrappers allow to intercept and augment the behavior of existing methods in order to wrap new behavior around them [Brant et al., 1998, Greevy and Ducasse, 2005]. In the present context, this new behavior can be tracing functionality.

Ad-hoc based tracing. The previous mechanisms we have mentioned all have a structured way at going about the tracing operation. However, sometimes, when only a very limited amount of points of interest exist within a software system, manual or script-based instrumentation can be a (short-term) solution.

3.3.3 Implementation

For the experiments that we will describe in subsequent chapters we used two different trace extraction technologies, namely a profiler-based solution for the Java experiments we carried out and an aspect-based solution for the industrial experiment we carried out in a legacy C context. We will now give a brief overview of the technologies we used for extracting the execution traces from these Java and C systems.

Java For our Java experiments we chose to use the *Java Virtual Machine Profiler Interface* or short *JVMPI*[2]. This interface allows you to write a plugin in C/C++ that connects with the Java Virtual Machine (JVM). The JVMPI sends out events at such moments as a method entry, a method exit, an activation of the garbage collector, etc. The plugin that you can write yourself then, can be programmed to capture these events and handle them. In our case, we programmed a plugin that captured each method entry and exit and outputted this information to a trace file, similar to the one that can be seen in Figure 3.1.

The JVMPI was introduced with the release of Java 1.1 and has always been labeled as an experimental technology, which in a sense, it has always remained, up until its successor, the JVMTI[3] interface was presented with the release of Java 1.5.

A definite drawback to the JVMPI interface is the fact that it becomes unstable when used in combination with the HotSpot technology that was introduced in Java 1.3. HotSpot JVM's use the principle of *just-in-time* compilation to improve performance. This instability manifested itself through events that were never thrown by the JVMPI interface and as such are missing in the resulting trace. To overcome this problem we explicitly switched off the HotSpot technology when performing tracing operations. This resulted in lower performance, but in a more stable virtual machine and thus in better quality traces.

C When it comes to our experiments with software written in C, we made use of an aspect-oriented solution. We used *Aspicere*, a tool built by members from the ARRIBA team of the Ghent University, to instrument the application under study and generate an execution trace. Chapter 9 gives more details about our choice for Aspicere.

3.4 The observer effect

In many disciplines of exact science, the observer effect refers to changes that the effect of observing has on the phenomenon being observed. A classical example of this comes from the discipline of quantum physics, where the observation of an electron will change its path because the observing light or radiation contains enough energy to disturb the electron under study. In social sciences also, a similar effect has been reported, where people will start

[2]More information on this technology can be found at: http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/

[3]Java Virtual Machine Tool Interface

to behave differently when being observed. In social sciences this effect is called the *Hawthorn effect*.

In the field of software engineering, a similar effect has been observed, namely the *probe effect* [Andrews, 1998]. In the context of using dynamic analysis, this effect can manifest itself in different ways:

- Because the software system being traced is less responsive when executing the software system according to the pre-defined execution scenario, the user is likely to respond to this unresponsiveness by clicking on a button multiple times. As such, the actually executed scenario can possibly diverge from the pre-defined execution scenario.
- A second, possibly more serious threat can be the influence of the tracing operation on thread interactions that happen within the program being traced.

As such, it is advisable to generate as little as possible overhead when extracting the trace from a running software system in order to minimize the observer effect. A first step towards minimizing the overhead can be the *post-mortem* analysis of the trace, i.e. analyzing the trace after the program (and its accompanied tracing operation) has finished. This solution stands opposite to an *online* analysis.

3.5 Threats to using dynamic analysis

When performing dynamic analysis, one wants to generate a high-quality execution trace of the executed scenario. High-quality, meaning that the trace we obtain is an actual reflection of what happened during the execution of the software. A number of situations however are typically problematic when performing dynamic analysis. In this section we will briefly discuss these and we will indicate which precautions we have taken to minimize their effects.

- In many software systems, lots of threads interact with each other to make the functionality of the system happen. These threads can interact in parallel (when more than one processor is available) or in sequence (when only one processor is available). Just storing all the actions of each of the threads in one execution trace can lead to a confusing image, because the execution trace would make one believe that two methods were executed sequentially, whereas actually they were executed by two completely different threads. To overcome this situation, an execution trace should be made for each thread that is active during the execution of the scenario.
- More and more systems make use of classloading functionality or re-

flection mechanisms to load classes dynamically. When using a profiler or debugger based dynamic analysis solution, this often leads to a situation where the resulting execution trace contains entries from the classloader of reflection mechanism whenever calls to methods from the loaded class are made. Currently, we have not taken any countermeasures to prevent this from happening. A possible consequence of this is that some method calls are not correctly recorded, meaning that the execution trace contains an entry of a call to the reflection mechanism or the classloader, but not of the actual method that is executed through these mechanisms. To the best of our knowledge, the execution scenarios we used for our experiments did not contain any of these situations.

3.6 Strengths and weaknesses

Table 3.1 gives an overview of the strengths and weaknesses of using dynamic analysis for program comprehension purposes.

	Strength	Weakness
Polymorphism	√	
Goal-oriented	√	
Overhead		√
Observer effect		√

Table 3.1: Strengths and weaknesses of dynamic analysis

Part II

Coupling based solutions for program comprehension

Chapter 4

How coupling and program comprehension interact

To manage a system effectively, you might focus on the interactions of the parts rather than their behavior taken separately.

—Russell L. Ackoff

It has been observed that software engineers who are trying to become familiar with a software system follow structural dependencies that are present in the system to navigate through the system. Coupling is a direct consequence of these structural relationships. This chapter describes how runtime coupling measures provide an indication for classes that need to be understood early on in the program comprehension process.

4.1 Introduction

Program comprehension is an inherently human activity, as such intuition and a dose of luck are essential ingredients to complete this mission successfully. Recent empirical studies have shown that when effective developers have to identify high-level concepts associated with the task at hand in the source code, they have a tendency to follow structural dependencies [Robillard et al., 2004]. Novice developers however, working on an unfamiliar system may easily get stuck in irrelevant code and fail to notice important program functionality, leading to low-quality software modifications [Robillard et al., 2004] or non-optimal time-management.

Within this research track it is our goal to provide the end-user with a number of starting points, from which they can start following structural dependencies in order to familiarize themselves with the system under study. The basic means by which we want to identify these starting points is *coupling*.

4.2 Coupling

Coupling was introduced by Constantine et al in the 1960s as a heuristic for designing modules [Yourdon and Constantine, 1979]. Constantine's original definition of coupling is *"a measure of the strength of interconnection between modules"*. Constantine's first definition is rather informal and we will use Wand's definition to describe the basic concept of coupling [Wand and Weber, 1990, Chidamber and Kemerer, 1994].

> Two things are coupled if and only if at least one of them "acts upon" the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time.

Software systems are typically composed from several software entities — be it modules, classes, components, aspects,... These entities work together to reach their goal(s) and the collaborations that exist between these entities give rise to the notion of coupling. From this observation comes the definition of coupling from Stevens: "the measure of strength of association established by a connection from one module to another" [Stevens et al., 1974].

In the light of this definition, a higher level of coupling means that the modules concerned are more inter-related and as such these modules are more difficult to understand, change, reuse and correct. From this empirical observation stems the basic principle of the pursuit of low coupling levels within a software system [Selby and Basili, 1991]. Intuitively however, coupling will always exist within software systems, as modules or classes need to work together to reach their goals and ultimately deliver the desired end-user functionality [Lethbridge and Anquetil, 1998]. This observation, together with the definition postulated by Wand [Wand and Weber, 1990], makes that we can categorize classes that have a relatively high degree of coupling as *influential*. Influential, because they have a certain amount of control over *what* the application is doing and *how* it is doing it.

In her research about design flaws, Tahvildari uses a similar concept, called *key classes* [Tahvildari, 2003]:

> *"These key classes are described as the classes that implement the key concepts of a system. Usually, these most important concepts of a system are implemented by very few key classes, which can be characterized by a number of properties. These classes which we called key classes manage a large amount of other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system. Additionally, they tend to be rather complex, since they implement much of the legacy system's functionality."*

4.3 Dynamic coupling

4.3.1 Introduction

Coupling measures have for some time now been subject of research, e.g. in the context of quality measurements. These measures have mostly been determined *statically*, i.e. based upon structural properties of the source code (or models thereof). However, with the wide-spread use of object oriented programming languages, these static coupling measures lose precision as more intensive use of inheritance and dynamic binding occurs. Another factor that possibly perturbs the measurements is the presence of dead code, which can be difficult to detect statically in the presence of polymorphism.

This has led us to start looking at dynamic coupling measures, a branch of software engineering research that has only recently been developing [Arisholm et al., 2004]. We propose the following working definition for dynamic coupling measures:

> Dynamic coupling measures are defined based upon an analysis of interactions of runtime objects. We say that two objects are dynamically coupled when one object acts upon the other. Object x is said to act upon object y, when there is evidence in the execution trace that there is a calling relationship between objects x and y, originating from x. Furthermore, two classes are dynamically coupled if there is at least one instance of each class for which holds that they are dynamically coupled.

The basic framework we use when considering dynamic coupling measures was first introduced by Arisholm et al. [Arisholm et al., 2004].

4.3.2 Classification of dynamic coupling measures

Dynamic coupling can be measured in different ways. Each of the measures can be justified, depending on the application context where such measures are to be used [Arisholm et al., 2004]. Table 4.1 is based on the variations that Arisholm et al have defined. Each of the variations will also be discussed in this section.

Entity	Granularity (Aggregation Level)	Scope (Include/Exclude)	Direction
Object	Object Class (set of) Scenario(s) (set of) Use case(s) System	Library objects Framework objects Exceptional use cases	Import/Export ...
Class	Class Inheritance Hierarchy (set of) Subsystem(s) System	Library classes Framework classes	Import/Export ...

Table 4.1: Dynamic coupling classification.

1. **Entity of measurement**. Since dynamic coupling is calculated from dynamic data stored in the event trace, we can calculate coupling at the *object*-level or at the *class*-level.
2. **Granularity**. Orthogonal to the entity of measurement, dynamic coupling measures can be aggregated at different levels of *granularity*. With respect to dynamic *object* coupling, measurement can be performed at the object level, but can also be aggregated at the class level, i.e. the dynamic coupling of all instances of a class is aggregated. Different kinds of aggregations can be made depending on the entity of measurement. Aggregations that can be made include: at the *(sub)system*, *inheritance hierarchy*, *use case* or *scenario* level.
3. **Scope**. Another variation can be the classes we want to consider when calculating the metric(s). For example, instances of library or framework classes can sometimes be of no special interest and as such they can be excluded.
4. **Direction (import or export)**. Consider two classes c and d being coupled by the invocation of a method m_2 of d in a method m_1 in class c. This relationship can be described as a client-server relationship between the classes: the client class c uses (*imports* services), the server class d is being used (*exports* services). This situation gives rise to the concepts of import and export coupling.

4.3.3 Dynamic coupling for program comprehension

Based on the classification schema presented in Section 4.3.2 we will now discuss which properties we expect from a coupling metric in order to be useful for program comprehension purposes. Based on these properties, we will find those dynamic coupling metrics that suit our intentions best.

1. At a cognitive level, the software engineer trying to get a first impression of a piece of software, will probably try to comprehend the software at the *class-level*, as these are the concepts he/she can recognize in the source code, the documentation and the application domain.
2. As such we advocate either the use of classes as level of granularity or a further aggregation up to the the (sub)component (or in other terms package) level.
3. With regard to the scope, we discard all classes foreign to the actual project (e.g. libraries), as these have no direct influence on the program comprehension process. Furthermore, choosing an execution scenario of the software that involves the features that you are interested in from a program comprehension point of view, is essential.
4. In Section 4.2 we already stated that we are looking for classes that have a prominent role within the system's architecture. We expect these classes to *give orders* to other classes, i.e. tell them what to do and what to give in return. As such, we expect these classes to request the services of (many) other classes, which in terms of the *direction* of coupling, is translated as *import* coupling. On the other hand, we expect classes with strong export coupling to be classes that provide services to other classes.

Arisholm et al. defined twelve dynamic coupling metrics; two of these adhere to the criteria we set out, namely: working at the class-level and measuring import coupling [Arisholm et al., 2004]. We will now discuss these two metrics.

1. *Distinct method invocations.* This measure counts the number of *distinct* methods invoked by each method in each object. This information is then aggregated for all the objects of each class. Arisholm et al. call this metric IC_CM (Import Coupling, Class level, Distinct Methods). Calls to methods from the same object or class (cohesion) are excluded.
2. *Distinct classes.* This measure counts the number of distinct server classes that a method in a given object uses. That information is than aggregated for all the objects of each class. Arisholm et al. call this metric IC_CC (Import Coupling, Class level, Distinct Classes). Calls to methods from the same object or class (cohesion) are excluded.

Consider the formal definitions of IC_CC and IC_CM in Table 4.2.

C	Set of classes in the system.
M	Set of methods in the system.
R_{MC}	$R_{MC} \subseteq M \times C$ Refers to methods being defined in classes.
IV	$IV \subseteq M \times C \times M \times C$ The set of possible method invocations.
$IC\_CM(c_1) =$	$\#\ \{(m_1, c_1, m_2, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}\ ) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\ \}$
$IC\_CC(c_1) =$	$\#\ \{(m_1, c_1, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}\ ) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\ \}$
$IC\_CC'(c_1) =$	$\#\ \{(m_2, c_1, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}\ ) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\ \}$

Table 4.2: Dynamic coupling measures [Arisholm et al., 2004].

Now reconsider the IC_CC metric. When we are looking for a metric that points to classes that import a lot of services from other classes, we see that IC_CC has a limited range. IC_CC counts the number of (m_1, c_1, c_2) triples. Because the first component in this triple is m_1, the maximum metric value is the product of the number of methods in the definition of c_1 and the number of classes c_1 interacts with. Because the number of methods defined in c_1 plays a vital role in the calculation of this metric, this can become a limiting factor. Furthermore, it does not give a true reflection as to how many other classes and in particular methods in other classes are used.

Therefore, we made a variation on the IC_CC metric, called IC_CC′. This variation does not count the number of calling methods, but the number of called methods. In other words, triples of the form (m_2, c_1, c_2) are counted.

A formal definition of IC_CC′ can be found in Table 4.2.

Example A class c_1 which only has 1 method, calls 4 distinct methods m_1,... ,m_4 in a class c_2 and calls 2 methods m_5 and m_6 in a class c_3. Calculating IC_CC and IC_CC′ for c_1 would yield 2, respectively 6. This indicates that IC_CC is targeted more towards finding the number of class-collaborations, while IC_CC′ retrieves the number of method-collaborations.

In our experiment (see Chapter 6) we will make a thorough comparison of the effectiveness of the three aforementioned metrics.

4.4 Research question

The central research question of this research track is whether there is a clear link between influential classes and the classes that need to be understood during initial programming understanding. These need-to-be-understood classes will be designated *important*, as their comprehension is needed in order to understand other classes and interactions within the software system.

A more abstract description of the research question we are trying to solve is whether it is possible to identify these important classes based solely on the topological structure of the application. In this context, the topological structure is instantiated by coupling relationships between classes.

A number of subsidiary questions for this research goal are:

1. Which type of coupling measurements are best at mapping influential modules or classes on the important modules or classes?
2. Is the simple measurement of a binary coupling relation sufficient to retrieve important classes or do we need to add a measure to take into account indirect coupling?

4.5 Research plan

Over the course of this research track we have developed three heuristical techniques to identify important classes (or modules) in a system.

1. Dynamic coupling measures as indicators of classes requesting a significant amount of actions to be performed for them (see previous sections).
2. Webmining algorithms that allow to take into consideration indirect coupling (Chapter 5).
3. Static coupling measures as an alternative to their dynamic counterparts (Chapter 7).

4.6 Validation and evaluation

Each of the three techniques will be evaluated intrinsically against two case studies. The evaluation is done according to three evaluation criteria, namely:

1. The *recall* of the resultset, or in other words, the technique's retrieval power.
2. The *precision* of the resultset, or in other words, the technique's retrieval quality.
3. The *effort* it takes to perform the complete analysis, from start to finish.

Validation of each of the techniques is done the same way, with recall and precision being the deciding factors. For each type of analysis, we will also perform an effort analysis, which, although secondary to the primary criteria of recall and precision, can be a deciding factor when it comes to determining the *return on time-investment*.

4.7 Practical application

In general, we can describe the process to be followed by a software (re)engineer using our heuristics as follows:

1. Definition of the execution scenario.
2. Trace the application according to the chosen execution scenario.
3. Determine the most important classes using one of the heuristics proposed.
4. Interpret the results.

Chapter 5

Webmining

Keep on the lookout for novel ideas that others have used successfully. Your idea has to be original only in its adaptation to the problem you're working on.

—Thomas Edison

Webmining, which is a form of datamining, is a mining technique which solely uses the topological structure of a graph to determine which nodes are important within a graph. We rely on these webmining techniques to add the notion of indirect coupling to our previously built-up theory on dynamic coupling and program comprehension.

5.1 Indirect coupling

5.1.1 Context and definition

Up until now we have talked about *direct* coupling. Direct coupling is a relationship between two entities. However, when considering large-scale software systems it is far from inconceivable that more than 2 entities influence each other. Reconsider the coupling definition from Wand (see Section 4.2) and let X, Y and Z be 3 entities where, respectively (X, Y) and (Y, Z) are directly coupled, i.e. X acts upon Y and Y acts upon Z. Intuitively, it is easy to understand that it is possible that X also (indirectly) acts upon Z. Consider the example code in Figure 5.1. In this example, where (X, Y) and (Y, Z) are directly coupled, it is clear to see that it is possible that X acts upon Z through the parameter that is passed. In terms of object orientation

and polymorphism, it is furthermore possible that not only parameter values, but also a parameter's dynamic type can be of influence.

```
class X
{
    Y y = new Y();
    void doitX(int param)
    {
        ...
        y.doitY(param);
        ...
    }
}
```

```
class Y
{
    Z z = new Z();
    void doitY(int param)
    {
        ...
        z.doitZ(param);
        ...
    }
}
```

```
class Z
{
    void doitZ(int param)
    {
        ...
    }
}
```

Figure 5.1: Indirect coupling example.

Based upon this observation, we investigate the notion of *indirect* coupling [Yang et al., 2005]. Briand et al. use the following definition [Briand et al., 1999]:

> Direct coupling describes a relation on a set of elements (e.g. a relation "invokes" on the set of all methods of the system, or a relation "uses" on the set of all classes of the system). To account for indirect coupling, we need only use the transitive closure of that relation.

5.1.2 Relevance in program comprehension context

Consider Figure 5.2, where part of a system is visualized. The nodes in this graph represent classes, the edges indicate calling relationships. Furthermore, each edge is annotated with a weight, indicating the strength of the (coupling) relationship. It becomes immediately clear that the class `Task` is (strongly import-) coupled to three other classes. By that same observation, the class `Main` is weakly coupled (to `Task`).

From a program comprehension point of view, the class `Main` can still be of interest, because it can be influential to what `Task` does (e.g. the parameters

passed or the dynamic type of the parameters can have an influence). As such, by adding the concept of indirect coupling, `Main` will now benefit from the strong level of coupling exhibited by `Task`.

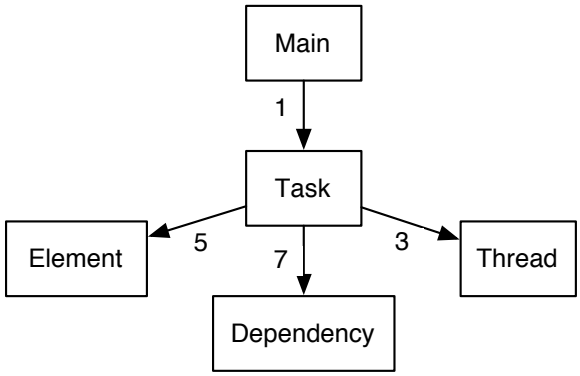


Figure 5.2: Indirect coupling example.

We will employ the iterative-recursive nature of the HITS[1] webmining algorithm to express the concept of indirect coupling towards our goal of program comprehension.

5.2 The HITS webmining algorithm

5.2.1 Introduction

Webmining, a branch of datamining research, deals with analyzing the structure of the internet (or to be more specific: the web) [Brin and Page, 1998, Gibson et al., 1998, Kleinberg, 1999]. Typically, webmining algorithms see the internet as a large graph, where each node represents a webpage and each edge represents a hyperlink between two webpages. Using this graph as an input, the algorithm allows us to identify so-called *hubs* and *authorities* [Kleinberg, 1999]. Intuitively, on the one hand, hubs are pages that refer to other pages containing information rather than being informative themselves. Standard examples include web directories, lists of personal pages, ... On the other hand, a page is called an authority if it contains useful information and is referenced by others (e.g. web pages containing definitions, personal information, ...).

Software systems can also be represented by graphs, where classes are nodes and calling relationships between classes are edges (e.g. see Figure

[1] Hypertext-Induced Topic Search

5.2). Furthermore, there is a "natural" extension to the concepts of *hubs* and *authorities* in the context of object-oriented software systems. Classes that exhibit a large level of import coupling call upon a number of other classes that do the groundwork for them. In order for them to control these assisting classes, they often contain important control structures. As such, they have a considerable amount of influence on the data and control flow within the application. Conceptually, the classes that have a high level of import coupling are similar to the hubs in web-graphs.

Export coupling on the other hand is often a sign of very specific functionality, often frequently reused throughout the system. Because of their specificity, they are conceptually bonded to authorities in web-graphs.

Because of this conceptual similarity, we found it worthwhile to try and reach our goal of identifying important classes in a system through the HITS webmining algorithm [Kleinberg, 1999].

In the context of this experiment, the calling relationships between the classes are determined dynamically.

5.2.2 HITS algorithm

The HITS algorithm works as follows. Every node i gets assigned to it two numbers; a_i denotes the authority of the node, while h_i denotes the hubiness. Let $i \rightarrow j$ denote that there is a link from node i to node j. The recursive relation between authority and hubiness is captured by the following formulas.

$$h_i = \sum_{i \rightarrow j} a_j \tag{5.1}$$

$$a_j = \sum_{i \rightarrow j} h_i \tag{5.2}$$

The HITS algorithm starts with initializing all h's and a's to 1. In a number of iterations, the values are updated for all nodes, using the previous iteration's values as input for the current iteration. Within each iteration, the h and a values for each node are updated according to the formulas (5.1) and (5.2). If after each update the values are normalized, this process converges to stable sets of authority and hub weights. Proof of the convergence criterion can be found in Appendix A or in [Kleinberg, 1999].

It is also possible to add weights to the edges in the graph. Adding weights to the graph can be interesting to capture the fact that some edges are more important than others. This extension only requires a small modification to

the update rules. Let $w[i, j]$ be the weight of the edge from node i to node j. The update rules become:

$$h_i = \sum_{i \to j} w[i, j] \cdot a_j \tag{5.3}$$

$$a_j = \sum_{i \to j} w[i, j] \cdot h_i \tag{5.4}$$

5.2.3 Example

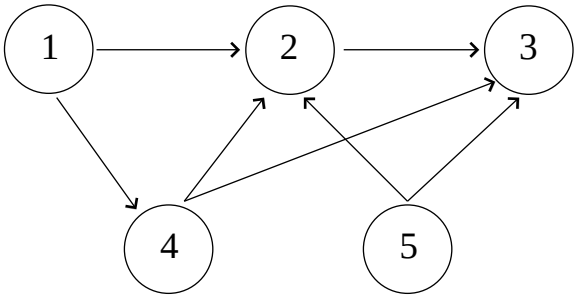


Figure 5.3: Example graph

Consider the example graph of Figure 5.3. Table 5.1 shows three iteration steps of the hub and authority scores (represented by tuples *(H, A)*) for each of the five nodes in the example graph. Even after only 3 iterations steps, it becomes clear that 2 and 3 will be good authorities, as can be seen from their high A scores in Table 5.1. Looking at the H values, 4 and 5 will be good hubs, while 1 will be a less good one.

		Nodes				
		1	2	3	4	5
Iterations	1	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)
	2	(2,0)	(1,3)	(0,3)	(2,1)	(2,0)
	3	(4,0)	(3,6)	(0,5)	(6,2)	(6,0)
	4	...	...	...	...	...

Table 5.1: Example of the iterative nature of the HITS algorithm. Tuples have the form (H,A).

5.3 Practical application

In order to apply the HITS webmining algorithm, we need the conceptual model of a graph. This graph, which we call the *compacted call graph* [Zaidman et al., 2005], is built up as follows:

- The classes in a system form the nodes, while the calling relationships between classes are indicated by edges.
- The strength of each calling relationship from class A to class B is determined by the number of elements in the set[2]:

  $$\{(m_B, A, B) | (\exists (m_A, A), (m_B, B) \in R_{MC} \wedge A \neq B \wedge (m_A, A, m_B, B) \in IV\}$$

  Each edge is annotated with the calling relationship strength (see Figure 5.3).
- The HITS webmining algorithm can now be applied on the graph.

On a side note, there is a clear equivalence relationship between building up the compacted call graph and calculating the IC_CC′ metric.

$$IC\_CC'(A) = \sum_{i \to j} w[i, j] \tag{5.5}$$

where i is the node that represents class A in the compacted call graph and j ranges over the classes to which instances from A send messages to.

[2]For information about the symbols used, please consult Table 4.2 in the previous chapter.

Chapter 6

Experiment

A thinker sees his own actions as experiments and questions — as attempts to find out something. Success and failure are for him answers above all.

—Friedrich Nietzsche

In the previous chapter we set out the theory behind our analysis that retrieves the key classes that need to be understood early on in the program comprehension process. In this chapter we use two open source software projects as case studies to compare the solutions that we have proposed and to determine how good the technique actually performs.

6.1 Experimental setup

6.1.1 Case studies

We selected two open-source software projects as case studies for the full duration of this research track. When selecting these case studies, we were specifically looking for two properties that would make the software projects particularly well-suited for our program comprehension experiments:

- Their public nature ensures the repeatability of these or similar experiments within the research community.
- The presence of extensive design documentation is very useful for validating program comprehension experiments. Furthermore, the fact that this extensive design documentation is freely available, is a further bonus with respect to the guarantee of repeatability.

Ultimately, we chose *Apache Ant 1.6.1* and *Jakarta JMeter 2.0.1* because they adhere best to the criteria we set out. Although a number of open-source projects would adhere to the above properties, the specific choice for these two projects is also given by the fact that both software systems are completely different kinds of applications, e.g. Ant is a command-line batch application, while JMeter features a highly interactive graphical user interface.

6.1.2 Execution scenarios

The choice of execution scenario is very important and can influence the resultset. On the other hand, a well-chosen execution scenario can also be an advantage when reverse engineering large software systems: a strict execution scenario that only executes use cases that the reverse engineer is interested in, can help in reducing the resultset. As such, it enables a goal oriented approach. Within the context of this experiment, the execution scenario is a two-sided sword that can help bring precision, but can also make the results less reliable.

The precise execution scenarios which we used for each of the case studies will be discussed in Sections 6.2.3 and 6.3.3.

6.1.3 Program comprehension baseline

The presence of extensive design documentation made it possible to define a baseline for our program comprehension research. This baseline is the set of classes that are marked by the original developers and/or current maintainers as *need-to-be-understood* before any (re)engineering operation can take place on the project. This baseline however remains an approximation, because it is based on the experienced developer's point of view, and not on the experience of a novice maintainer who is trying to understand the software system.

As such, this baseline enables us to do an *intrinsic evaluation* of the heuristics. Intrinsic, meaning that we use the developers' and maintainers' opinion to compare with the results we have obtained. Opposed to this intrinsic evaluation stands an *extrinsic evaluation*, where we would empirically evaluate the effectiveness of the proposed program comprehension techniques [Hamou-Lhadj, 2005a]. At this moment, we regard this extrinsic evaluation as future work.

Applying one of the heuristics we have presented in the previous chapters results in a list of classes ranked according to their relative importance

according to the heuristic. By default, we only present the 15% most highly ranked classes, the reasoning behind this is as follows:

- From the documentation of both Apache Ant and Jakarta JMeter we have learned that about 10% of the classes of the systems need to be understood before any meaningful change operation can take place. As we are working with a heuristical technique we took a 5% margin.
- For cognitive reasons, the size of the data presented to the users should be kept to a minimum, as to not overload the user with information. As such, the resultset should be kept as minimal as possible.
- Empirically, we found that lowering the threshold to the top 20% classes, did not result in an increase in recall. To be more precise, we did not notice any classes mentioned in the documentation showing up in the interval [15%, 20%] [Zaidman et al., 2005].

6.1.4 Validation

As a validation we propose to use the concepts of *recall* and *precision*. Each resultset we obtain from applying one of the proposed heuristics will be compared to the baseline that we defined.

Recall is the technique's ability to retrieve all items that are contained in the baseline, while precision is the quality of the retrieved items contained in the resultset. Recall and precision are defined as follows:

$$Recall\ (\%) : \frac{A}{A+B} \times 100 \tag{6.1}$$

$$Precision\ (\%) : \frac{A}{A+C} \times 100 \tag{6.2}$$

A: relevant, retrieved items
B: relevant, non-retrieved items
C: irrelevant, retrieved items

6.1.5 Research plan

In Sections 6.2 and 6.3 we will compare and discuss the results we have obtained from the 4 dynamic approaches to identifying important classes we have introduced in the previous chapters. To be more precise, we will compare IC_CM, IC_CC, IC_CC′ and IC_CC′ combined with the webmining approach to the program comprehension baseline we have obtained. In this comparison, recall and precision play a major role and are the deciding factors as to which of the approaches delivers the best results.

Chapter 7 then describes a control experiment in which we compare the dynamic approach that delivered the best results with a number of static

variants of our dynamic approach. For this control experiment, we will not only focus on the two primary criteria of recall and precision, but we add a third — although secondary — criterion, namely round-trip-time, i.e. the time needed to perform a complete analysis.

6.1.6 Threats to validity

We identified a number of potential threats to validity:

- In the case of Apache Ant, the design documentation we used[1] dates from 2003. Although since then, no major overhauls of the architecture were reported, the fact that the source code and the technical documentation are not perfectly synchronized can be a threat to the validation principle we propose. Other than the fact that one class that was mentioned in the documentation is no longer part of the Ant distribution, there have been no consequences with regard to our experimental setup.
- Comparing static and dynamic analysis poses some threats to the validity of our experimental setup. When considering the 15% most highly ranked classes, the size of this 15% resultset varies according to the size of the inputset, namely the number of classes. In the case of the static process, the size of the inputset equals the total number of defined classes, while in the dynamic process, this equals the number of classes that participate in the execution scenario(s). In most cases, the number of classes participating in an execution scenario will be lower than the total number of classes present in a system. As such, the primary criterion on which to compare the resultsets should be *recall*, because the precision will drop automatically when considering the often larger resultsets of static analysis.

6.2 Apache Ant

6.2.1 Introduction

Apache Ant 1.6.1[2] is a well-known build tool, mainly used in Java environments. It is a command-line tool, has no GUI and is single-threaded. It has a relatively small footprint, but it does however use a lot of external libraries (e.g. the Xerces XML library) and is user-extensible. Ant relies heavily on XML, as the propriety build files are written entirely in XML.

[1] http://codefeed.com/tutorial/ant_config.html
[2] For more information, see: http://ant.apache.org

Even though Ant is open-source, it is used both in open-source and industrial settings. Furthermore, it has been integrated in numerous (Java) Integrated Development Environments (IDE's) (e.g. Eclipse, IntelliJ IDEA, ...). A number of extensions to the basic Ant distribution have been written (e.g. GUI's) and there has even been a complete port to the .NET environment (called nANT).

The source-file distribution of Apache Ant 1.6.1 contains 1216 Java classes. Only 403 of these classes (around 83 KLOC) are Ant-specific, as most of the classes in the distribution belong to general purpose libraries or frameworks, such as Apache ORO (for regular expressions) or Apache Xerces (XML parser).

6.2.2 Architectural overview

With the help of the freely available design documentation[3], we will discuss the role the five classes that are considered important by the architects, play in the execution of a build.xml file:

1. `Project`: Ant starts in the Main class and immediately creates a `Project` instance. With the help of subsidiary objects, the `Project` instance parses the build.xml file. The xml file contains *targets* and *elements*.
2. `Target`: this class acts as a placeholder for the *targets* specified in the build.xml file. Once parsing finishes, the build model consists of a project, containing multiple targets – at least one, which is the implicit target for top-level events.
3. `UnknownElement`: all the elements that get parsed are temporarily stored in instances of `UnknownElement`. During parsing the `UnknownElement` objects are stored in a tree-like datastructure in the `Target` to which they belong. When the parsing phase is over and all dependencies have been determined, the `makeObject()` method of `UnknownElement` gets called, which instantiates the right kind of object for the data that was kept in the placeholder `UnknownElement` object.
4. `RuntimeConfigurable`: each `UnknownElement` has a corresponding `RuntimeConfigurable`, that contains the element's configuration information. The `RuntimeConfigurable` objects are also stored in trees in the `Target` object they belong to.
5. `Task` is the superclass of `UnknownElement` and is also the baseclass for all types of tasks that are created by calling the `makeObject()` method

[3]The design documentation of Ant can be found at: http://codefeed.com/tutorial/ant_config.html

of `UnknownElement`.
We tried to record the relationship between those 5 classes in Figure 6.1. Besides these 5 key classes, the design documentation also mentions five

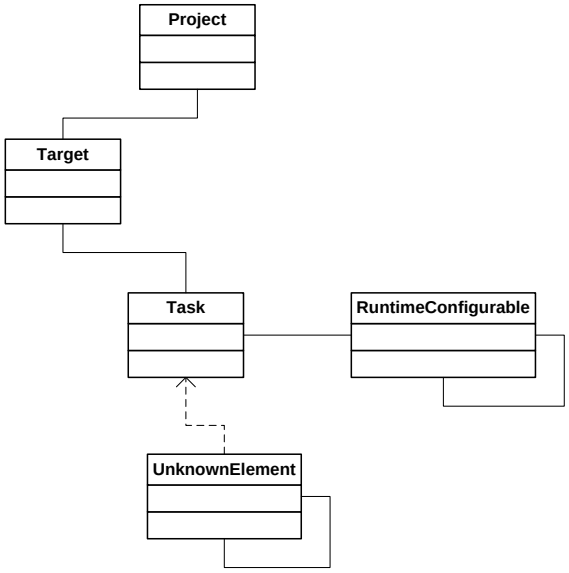


Figure 6.1: Simplified class diagram of Apache Ant.

other important (helper)classes:

- `IntrospectionHelper`
- `ProjectHelper2`
- `ProjectHelperImpl`
- `ElementHandler`
- `Main`

6.2.3 Execution scenario

We chose to let Ant build itself as the execution scenario of choice for our experiment. This scenario involved 127 classes. At first sight this may seem rather low, considering that Ant is built from 403 classes in total. This can be explained from the fact that the Ant architecture contains some very broad (and sometimes deep) inheritance hierarchies. For example the number of direct subclasses from the class *Task* is 104. Each of these 104 classes stands for a typical command line task, such as *mkdir*, *cvs*, ... As typical execution scenarios do not contain all of these commands (some are even conflicting, e.g. different versioning system or different platform), the execution scenario containing 127 classes covers all basic functionality of the Ant system.

The two main reasons why we chose this particular execution scenario are:

- It offers a good balance of features that get exercised, furthermore it contains all typical build commands, including those for copying files into different directories, generating jar (archive) files, etc.
- Every source file distribution of Ant contains this specific execution scenario, through the build.xml file that is included in the distribution.

6.2.4 Discussion of results

We will now discuss the results we have obtained from applying each of the techniques to the Apache Ant case study. Table 6.1 gives an overview of the aforementioned results.

Class	1: IC_CM	2: IC_CC	3: IC_CC′	4: IC_CC′ + webmining	5: Ant docs
Project		√		√	√
UnknownElement	√	√	√	√	√
Task	√	√	√	√	√
Main				√	√
IntrospectionHelper		√	√	√	√
ProjectHelper		√	√	√	√
RuntimeConfigurable	√	√	√	√	√
Target	√	√	√	√	√
ElementHandler			√	√	√
TaskContainer				*N/A*	√
→ **recall** (%)	40	70	70	90	-
→ **precision** (%)	27	47	47	60	-

Table 6.1: Ant metric data overview.

The IC_CM metric for a class c_1, which counts quadruples of the form (m_1, c_1, m_2, c_2), exhibits the lowest recall of all dynamic analysis solutions: 40%. The IC_CM metric counts distinct method invocations originating from the same source (m_1, c_1) combination. As such, a class c_1 using low-level functionality from c_2 in each of its methods m_i, will get a high metric value. This causes noise in the resultset, because we are actually looking for classes

that use other (high-level) classes. This explains its relatively low recall when compared to the baseline.

The IC_CC and IC_CC′ metrics, which count (m_1, c_1, c_2) and (m_2, c_1, c_2) respectively, exhibit a similar recall of 70%. Although at this point, we would have expected IC_CC′ to perform considerably better, there is no noticeable difference with regard to the recall. Our expectation for a better performance from IC_CC′ stems from the fact that, just as is the case for IC_CM, IC_CC focusses on counting the originating class/method pair, while IC_CC′ shifts focus towards the target class/method pair.

When applying the HITS webmining algorithm on the IC_CC′ metric results, we see that we get a recall of 90%. This increase in recall happens because indirect coupling is taken into account when applying the HITS webmining algorithm on the coupling data.

With regard to precision, it is clear that the webmining algorithm allows to greatly improve precision and bring it to a level of 60%, which, to our opinion, is satisfactory for a heuristic. Satisfactory, but nothing more than that, because it still means that 40% of the program comprehension "pointers" returned to the user are potentially of lesser value.

Trade-off analysis

Based on the results we have obtained from the Apache Ant case study, this is our analysis:

- Running Ant according to the execution scenario takes 23 seconds without collecting trace-information. When we collect a trace from running Ant according to the same execution scenario, this now takes slightly under 1 hour[4]. The execution generates a trace of roughly 2 GB of data.
- Processing this amount of data and calculating the IC_CM, IC_CC and IC_CC′ metrics took 45 minutes (the three metrics were calculated in parallel, only calculating one of these at a time lowers the time needed by only a fraction).
- Applying the HITS webmining algorithm on the metric data takes less than 30 seconds.

When considering the return on time-investment, we are mainly looking at the round-trip-time, i.e. the time needed to perform the full analysis, from loading the project into the environment till having the results presented. From starting the reverse engineering process till having the results at one's

[4]Experiment conducted on an AMD Athlon 800 with 512MB memory running Fedora Core 3 Linux.

disposal takes roughly 105 minutes, which is partly due to the very slow trace-collection-phase. Although we expect to be able to improve these round-trip-times, because of the prototype state of our tools, we firmly believe that the order of magnitude of the round-trip-time is set.

Rounding up, we can say that we are very much satisfied with the level of recall that the dynamic analysis approach gives us. Furthermore, precision is also good at a level of 60%, however, the round-trip-time should be seen as a major detractor.

6.3 Jakarta JMeter

6.3.1 Introduction

Jakarta JMeter 2.0.1[5] is a Java application designed to test webapplications. It allows to verify the application (functionally), but it also allows to perform load-testing (e.g. to measure performance or stability of the software system). It is frequently used to test webapplications, but it can also handle SQL queries through JBDC. Furthermore, due to its architecture, plugins can be written for other (network) protocols. Results of performance measuring can be presented in a variety of graphs, while results of the functional testing are simple text files with output similar to output from regression tests.

JMeter is a tool which relies on a feature-rich GUI, uses threads abundantly and relies mostly on the functionality provided by the Java standard API (e.g. for network-related functionality)[6].

The source-file distribution of Jakarta JMeter 2.0.1 consists of around 700 classes, while the core JMeter application is built up from 490 classes (23 KLOC).

6.3.2 Architectural overview

What follows is a brief description of the innerworkings of JMeter.

The `TestPlanGUI` is the component of the user-interface that lets the end user add and customize tests. Each added test resides in a `JMeterGUIComponent` class. When the user has finished creating his or her `TestPlan`, the information from the `JMeterGUIComponents` is extracted and put into `TestElement` classes.

[5]For more information, see: http://jakarta.apache.org/jmeter/

[6]The design documentation can be found on the Wiki pages of the Jakarta JMeter project: http://wiki.apache.org/jakarta-jmeter

These `TestElement` classes are stored in a tree-like datastructure: `JMeterTreeModel`. This datastructure is then passed onto the `JMeterEngine` which, with the help of the `TestCompiler`, creates `JMeterThread`(s) for each individual test. These `JMeterThread`s are grouped into logical `ThreadGroup`s. Furthermore, for each test a `TestListener` is created: these catch the results of the threads carrying out the actual tests.

As such, we have identified nine key classes from the JMeter documentation. The design documentation also mentions a number of important helper-classes, being:

- `AbstractAction`
- `PreCompiler`
- `Sampler`
- `SampleResult`
- `TestPlanGui`

6.3.3 Execution scenario

The execution scenario for this case study consists of testing a HTTP (HyperText Transfer Protocol) connection to *Amazon.com*, a well-known online shop. More precisely, we configured JMeter to test the aforementioned connection 100 times and visualize the results in a simple graph. Running this scenario took 82 seconds. The scenario is representative for JMeter, because many of the possible variation points in the execution scenario lie in (1) the usage of a different protocol (e.g. FTP) or (2) in the output format of the data (e.g. different type of graph or plain-text). Also of importance to note here is that these 100 connections are initiated by a number of different threads, in order to simulate concurrent access to the Amazon web application. This entails that this particular case study is an example of a multi-threaded application.

6.3.4 Discussion of results

This section presents a discussion about the results from the Jakarta JMeter case study. Table 6.2 provides an overview of these results.

The IC_CM metric clearly lags behind the other dynamic metrics proposed with a recall of 14% and a precision of 10%. The explanation for this relatively bad result is identical to the reasoning given for the Ant case study.

In contrast with our previous case study, there is a notable difference between the most tightly coupled classes as reported by IC_CC versus IC_CC′. Although not immediately visible from Table 6.2, this phenomenon is related to the feature-rich graphical user interface (GUI). Although there is

Class	1: IC_CM	2: IC_CC	3: IC_CC′	4: IC_CC′ + webmining	5: JMeter docs
AbstractAction	√		√	√	√
JMeterEngine			√	√	√
JMeterTreeModel				√	√
JMeterThread			√	√	√
JMeterGuiComponent	√	√	√		√
PreCompiler				√	√
Sampler		√	√	√	√
SampleResult		√		√	√
TestCompiler			√	√	√
TestElement			√	√	√
TestListener			√	√	√
TestPlan			√	√	√
TestPlanGui			√	√	√
ThreadGroup				√	√
→ **recall** (%)	14	21	71	93	-
→ **precision** (%)	10	14	48	62	-

Table 6.2: JMeter metric data overview.

evidence of an attempt of a model-view-controller (MVC) pattern implementation [Gamma et al., 1995] (both from source code and from design documents), there still is a high degree of coupling from the view to the model in the MVC scheme. Furthermore, a high degree of coupling exists within the GUI layer.
Because certain classes in the GUI layer of JMeter can be catalogued as god classes (many methods, large methods), the IC_CC metric falsely registers these classes as important, due to the high method count of these classes. IC_CC′ however does not suffer from this because its measure is not dependent on the number of methods defined within the class.

With regard to the heuristic where we applied webmining on top of the IC_CC′ metric, the results are fairly convincing with a recall attaining 93%, while still offering a level of precision of 62%. So again, taking indirect coupling into account makes sure that the important classes can be retrieved.

Trade-off analysis

Based on the results and the effort it took to generate the resultset, we made the following analysis:

- Running JMeter without collecting trace information takes 82 seconds. The overhead introduced when recording all necessary run-time data makes the same execution scenario last around 45 minutes. The execution generates traces of roughly 600 MB of data. Notice the difference with the Ant case study, where we collected 2 GB during a similar 45 minute execution period (when tracing). This difference can mainly be attributed to the fact that JMeter relies heavily on library functions, which are excluded from the trace. This exclusion process however, also comes at an additional cost because for each call made, an exclusion-filter needs to be consulted before deciding whether to output a call to the tracefile or not.
- Processing this amount of data and calculating the IC_CM, IC_CC and IC_CC′ metrics took slightly under 30 minutes.
- Applying the HITS webmining algorithm on the metric data takes around 30 seconds.

Here we see a very similar situation to the one we encountered during the Ant case study. Results are very much satisfactory, but the round-trip-time is worrisome when one wants to gain a quick overview of the subject application.

6.4 Discussion

6.4.1 Experimental observations

Table 6.3 gives an overview of the experimental setup we performed. The columns show the two criteria according to which we weigh the quality and the effectiveness of the 4 variations of the heuristic we proposed. The observations we have made during the two case studies are synthesized with the help of a scale ranging from $--$ (for bad conformance to a certain criterion) to $++$ (for good conformance). A dot ($\cdot$) means neither positive nor negative.

Recall. From Table 6.3 it becomes immediately clear that applying the HITS webmining technique on the dynamic IC_CC′ measure delivers the best recall results. Looking back at Tables 6.1 and 6.2, we see that this technique is able to recall 90 and 93 percent of the classes defined in the baseline. The plain IC_CC′ metric, which does not take into account indirect coupling, comes in as second best with recall percentages of around 70% in both case

	Recall	Precision
IC_CM	−	−
IC_CC	−	−
IC_CC′	+	·
IC_CC′ + webmining	++	+

Table 6.3: Strengths and weaknesses of the proposed coupling-based techniques.

studies. IC_CM has a lower level of recall (50% or lower), while IC_CC slots in somewhere between IC_CC′ and IC_CM (for the Ant case study, IC_CC performs level with IC_CC′).

Precision. When it comes to precision, IC_CC′ combined with the HITS webmining approach comes out best with a precision of 60%. No other technique is able to reach a level of precision above 50%.

Overall. Looking at the two primary criteria, recall and precision, the approach consisting out of the combination of the IC_CC′ metric and the HITS webmining algorithm delivers the best results. However, the round-trip-time needed to perform a complete analysis remains a serious detractor.

6.5 Observations with regard to the research question

To sum up, we were trying to answer the following question: *"is there a clear link between influential classes and the classes that need to be understood during initial programming understanding?"*. We can answer this question affirmatively. We have based ourselves on two open source case studies for which we had a program understanding baseline available. Singling out the combination of IC_CC′ and the HITS webmining algorithm, we have observed that this heuristic is able to retrieve around 90% (lower bound) of the important classes, while maintaining a level of precision of around 60% (lower bound).

With regard to the subsidiary questions, "which metric to use" and "whether or not to take into account indirect coupling" we can add that the dynamic IC_CC′ metric performs best when taking into account indirect coupling (through the HITS webmining algorithm).

As such, we are able to provide the end-user with a tool that can help him/her gain a overview of the application and, foremost, a number of starting points from where to start his/her further program understanding reconnaissance.

Chapter 7

Static coupling

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art.

—John A. Locke

In this dissertation we have mainly talked about dynamic or runtime coupling up until now. Now that we have also obtained the results of our case studies, we are wondering whether a similar approach, albeit performed statically, can match or even surpass the results we have obtained from performing the webmining analysis with dynamically obtained coupling data. In this chapter we first define static coupling measures that are close to the one that we used for our dynamic-analysis-based experiment and then make a comparison of the results we have obtained.

7.1 Introduction & motivation

Calculating dynamic coupling metrics and the consequent application of the webmining technique is characterized by a number of constraints:

- The need for a good execution scenario.
- The availability of a tracing mechanism.
- Scalability issues (resulting trace file, overhead from tracing mechanism, ...).

These constraints apply on the techniques that we discussed in Chapters 4 & 5. In order to verify whether we could overcome some of these constraints by working with static analysis instead of dynamic analysis, we

have undertaken a control experiment. In this experiment we apply webmining techniques on a static topological structure of the application and verify whether we can get a similar level of recall and precision as we found for the dynamic approach (see Chapter 6), all the while obtaining a significantly better round-trip-time.

The setup of this experiment is to compare the candidate of choice from our previous experiments, namely the combination of the IC_CC′ metric with the HITS webmining technique, with a similar technique that uses static information. Furthermore, because we wanted to make the comparison as objective as possible, we defined static coupling metrics that are as close as possible to the IC_CC′ metric we used in Chapters 4 & 5.

7.2 A static coupling metrics framework

The framework from Arisholm [Arisholm et al., 2004] does not have to make a distinction between static and polymorphic calls due to the dynamic nature of its measurements. We add notational constructs from the unified framework for (static) object-oriented metrics from Briand et al [Briand et al., 1999] to the definitions that we previously used from Arisholm. That way, we can still use the basic notation from Arisholm we have used in the previous chapters. For that purpose, some helpful definitions are:

Definition 1 Methods of a Class.
For each class $c \in C$ let $M(c)$ be the set of methods of class c.

Definition 2 Declared and Implemented Methods.
For each class $c \in C$, let:

- $M_D(c) \subseteq M(c)$ be the set of methods declared in c, i.e., methods that c inherits but does not override or virtual methods of c.
- $M_I \subseteq M(c)$ be the set of methods implemented in c, i.e., methods that c inherits but overrides or nonvirtual noninherited methods of c.

Definition 3 M(C). The Set of all Methods.
$M(C) = \cup_{c \in C} M(c)$

Definition 4 $SIM(m)$. The Set of Statically Invoked Methods of m.
Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in SIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

Definition 5 $NSI(m, m')$. The Number of Static Invocations of m' by m.
Let $c \in C$, $m \in M_I(c)$, and $m' \in SIM(m)$. $NSI(m, m')$ is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$.

```
1 public void foo() {
2   BaseClass base = new BaseClass();
3   base.doSomething();
4   // some other functionality
5   base.doSomething();
6 }
```

Figure 7.1: Piece of Java code to help explain metrics.

Definition 6 $PIM(m)$. The Set of Polymorphically Invoked Methods of m.
Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in PIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism and dynamic binding, be invoked for an object of dynamic type d.

Definition 7 $NPI(m, m')$. The Number of Polymorphic Invocations of m' by m.
Let $c \in C$, $m \in M_I(c)$, and $m' \in PIM(m)$. $NPI(m, m')$ is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$.

7.3 Expressing IC_CC′ statically

With these added notational constructs, we are now able to write down four static coupling measures that closely resemble the measurements that were defined in Section 4.3.3.

The fact that one dynamic metric IC_CC′ is translated into 4 static metrics can be explained by the fact that the static environment offers some degrees of choice when calculating the metrics. Consider the Java code snippet in Figure 7.1:

- The choice between *static calls* and *polymorphic calls*. In other words when considering Figure 7.1, do we only count the reference to `BaseClass` or also to all subclasses of `BaseClass`?
- Do we count duplicate calls for the same (origin, target) pairs? When considering Figure 7.1 do we count the `base.doSomething()` call once or twice (lines 3 and 5).

For the purpose of our research we have defined 4 metrics that vary over the characteristics described above.

Definition SM_SO Static Metric, Static calls, count every Occurrence of a call only once.

$$SM\_SO(c_1, c_2) = \#\{(m_2, c_2, c_1) | \exists\ (m_1, c_1), (m_2, c_2) \in R_{MC} \\ \wedge\ c1 \neq c2 \wedge\ (m_1, c_1, m_2, c_1) \in IV \\ \wedge\ m_2 \in SIM(m1)\}$$

Definition SM_SW Static Metric, Static calls, count every occurrence of a call (Weighted).

$SM\_SW(c_1, c_2)$ = identical to $SM\_SO(c_1, c_2)$, but { } should be interpreted as *bag* or *multiset*.

Definition SM_PO Static Metric, Polymorphic calls, count every Occurrence of a call only once.

$$SM\_PO(c_1, c_2) = \#\{(m_2, c_2, c_1) | \exists\ (m_1, c_1), (m_2, c_2) \in R_{MC} \wedge c1 \neq c2 \wedge\ (m_1, c_1, m_2, c_1) \in IV \wedge m_2 \in PIM(m1)\}$$

Definition SM_PW Static Metric, Polymorphic calls, count every occurrence of a call (Weighted).

$SM\_PW(c_1, c_2)$ = identical to $SM\_PO(c_1, c_2)$, but { } should be interpreted as *bag* or *multiset*.

To calculate these metrics, we used the JDT2MDR Eclipse plugin developed by Bart Du Bois, a fellow member of the LORE research group [Zaidman et al., 2006b]. JDT2MDR transforms a Java project to a graph representation closely resembling the metamodel employed by Briand et al. [Briand et al., 1999], thereby enabling the calculation of the coupling and cohesion measures formalized in their paper.

7.4 Results

This section will give an overview of the results we have obtained from applying the static approach to finding the most important classes in our two case studies, Apache Ant and Jakarta JMeter. We compare the results we have obtained with (1) the best result obtained from the dynamic approach, namely the combination of IC_CC′ and webmining and (2) the baseline obtained from the documentation from these open source projects.

Besides recall and precision, the criteria we used for determining the best dynamic approach, we will also keep a close eye on the round-trip-time of the static approach, as this is a factor where we expect the static approach to be able to significantly outperform the dynamic approach.

7.4.1 Ant

Based on the results shown in Table 7.1, two categories are formed, namely the category of metrics that takes polymorphism into account (SM_P*) and

Class	1: IC_CC′ + webmining	2: SM_PO + webmining	3: SM_PW + webmining	4: SM_SO + webmining	5: SM_SW + webmining	6: Ant docs
Project	√	√	√	√	√	√
UnknownElement	√	√	√	√	√	√
Task	√	79	81	119	120	√
Main	√	√	√	√	√	√
IntrospectionHelper	√	√	√	116	105	√
ProjectHelper	√	97	99	90	190	√
RuntimeConfigurable	√	√	√	63	63	√
Target	√	89	93	100	100	√
ElementHandler	√	192	198	125	125	√
TaskContainer	*N/A*	398	403	381	383	√
→ **recall** (%)	90	50	50	30	30	-
→ **precision** (%)	60	8	8	5	5	-

Table 7.1: Ant metric data overview.

the category that does not take polymorphism into account (SM_S*). The former category exhibits a recall level of 50%, while the latter recalls 30%. Although interesting from the point of view that polymorphism does indeed play an important role when considering program comprehension, from a practical perspective, these results are disappointing when compared to the results obtained with the dynamic approach. The observation regarding polymorphism can be explained by the fact that (1) sometimes a base class is abstract or (2) the base class is not always (or should we say mostly not) the most important class in the hierarchy. The second variation point for the static metrics, namely whether to only count an occurrence of a particular call once or to count every occurrence of a call (weighted), does not seem to make any difference with regard to our specific context (small variations exist, but these do not influence the resultset).

The fact that precision for the 4 static metrics in columns 2, ..., 5 is much lower (8% or less) than what we experienced with the dynamic approach, can be explained by the size of the inputsets, as the inputset for the static experiment was 403 classes, while for the dynamic experiment this was only 127 classes. When using our rule-of-thumb of presenting the 15% highest ranked classes in the final resultset, we end up with 60 and 15 classes respectively.

A further point to be made regarding this rule-of-thumb is that when looking at the ranking of classes that fall outside the top 15%, lowering the bar to 20% would not have resulted in a (significant) gain in recall, while precision would drop further. We can also add, that by raising the bar to 10%, recall would fall with 10%.

Considering the round-trip-time, we measured that the prototype (static) metrics engine took one hour to calculate the metrics for Ant. Applying the HITS algorithm takes less than one minute.

7.4.2 JMeter

Class	1: IC_CC′ + webmining	2: SM_PO + webmining	3: SM_PW + webmining	4: SM_SO + webmining	5: SM_SW + webmining	6: JMeter docs
AbstractAction	√	275	275	336	336	√
JMeterEngine	√	√	√	484	484	√
JMeterTreeModel	√	√	√	150	150	√
JMeterThread	√	√	√	147	147	√
JMeterGuiComponent		√	√	475	475	√
PreCompiler	√	362	362	293	293	√
Sampler	√	457	478	454	454	√
SampleResult	√	119	119	209	209	√
TestCompiler	√	√	√	145	145	√
TestElement	√	√	√	451	451	√
TestListener	√	450	443	449	449	√
TestPlan	√	113	113	234	234	√
TestPlanGui	√	93	93	√	√	√
ThreadGroup	√	140	140	157	157	√
→ **recall** (%)	93	43	43	7	7	-
→ **precision** (%)	62	8	8	1.4	1.4	-

Table 7.2: JMeter metric data overview.

Similar to what we saw in the Ant case study, two groups can be identified within the JMeter resultset presented in Table 7.2, namely one group consisting out of SM_PO and SM_PW, and one group formed by SM_SO and SM_SW. Within these two groups, recall and precision are identical, although minimal differences exist when looking at the ranking of some classes.

In contrast with the previous case study, Ant, these differences are much more pronounced. It is our opinion that this is probably due to the fact that most method calls happen only once in each unique method, as opposed to multiple occurrences of a method call in a unique method, where the *weighted* approach (of SM_PW and SM_SW) would make the difference more pronounced.

Also to be noted is the sizeable dissimilarity between the results obtained while only taking into account static calls versus also taking polymorphic calls into account. As Table 7.2 shows, the SM_P* metrics have a recall of 43%, while the SM_S metrics only recall 7%.

Of interest to note is the fact that when looking at the ranking of the classes outside the top 15%, it is clear that lowering the bar to the 20% highest ranked classes would not improve recall.

For what the round-trip-time is concerned, the metrics engine took almost $1\ \frac{1}{2}$ hours to calculate the metrics for JMeter. This is a considerable increase from what we saw with Ant. This increase can be attributed to the fact that JMeter has (1) a larger codebase and (2) uses more libraries, which also need to be parsed. Applying the HITS algorithm takes slightly over one minute.

7.5 Discussion

7.5.1 Practical implications

In Section 7.1 we talked about three drawbacks of the dynamic webmining approach. Now, after having performed a similar experiment in a static way, we will discuss each of these drawbacks and see whether these are strictly inherent to the dynamic approach we introduced:

1. *The necessity of a good execution scenario.*
   When performing static analysis, having an execution scenario is no issue. However, access to the source code remains a prerequisite. For completeness sake, we do add that reverse engineering (and the subsequent extraction of coupling metrics) from binaries is sometimes possible. However, having access to the source code is a criterion which often has a much more limited impact than having a good execution scenario. As such, static analysis is to be favored here.
2. *The availability of a tracing mechanism.*
   Although a tracing mechanism is no longer an issue, having a metrics engine remains a necessity. To implement such an engine, either open source tools need to be available or a parser needs to be constructed. Because a similar precondition exists for both processes, neither of the

two approaches has a clear advantage here.

3. *Scalability issues.*
   In terms of scalability the dynamic process is plagued by the possibly huge size of the tracefile. This has consequences on multiple levels:
   - The I/O overhead on the traced application (e.g. for Ant: execution of 23 seconds without tracing versus just under one hour with tracing).
   - The size of the trace (2 GB in the case of Ant).
   - The time it takes to calculate the IC_CC′ metric and perform the HITS webmining algorithm on this 2 GB of data. In the case of Ant this takes around 45 minutes.

   We were already aware of the below par round-trip-times from the dynamic approach. However, when comparing these times with the static approach, we observe that our prototype metrics engine took one hour to calculate the metrics for Ant and slightly over one hour for JMeter. Applying the HITS algorithm takes less than one minute, so the total round trip time is around one hour for both projects. While these times are not so different from the dynamic process, the dynamic process still needs the tracing step, which makes that the round trip time for the dynamic process is significantly larger and in the case of Ant takes around two hours.

7.5.2 Comparing static and dynamic results

In Chapter 6 we saw that the IC_CC′ metric combined with the webmining solution provides a level of recall of at least 90%, while safeguarding a level of precision of around 60%. When we look at the results of the static coupling metrics that we introduced in this chapter, we see that we are able to reach a maximum level of recall of 50%, while the level of precision drops to 8% or less. This observation makes it quite obvious that the dynamic approach is the solution of choice when only considering the recall and precision results.

7.5.3 Conclusion

Table 7.3 provides an overview of the strengths and weaknesses of both the static and the dynamic approach. Although we see that the static approach (the SM_* metrics) are better at the round-trip-time performance, they fall through when considering their recall and precision characteristics. As such, when considering early program comprehension purposes, the dynamic approach is the best choice, even though its round-trip-time performance is a severe drawback.

	Recall	Precision	Time
IC_CC′ + webmining	++	+	−−
SM_PO + webmining	·	−−	+/−
SM_PW + webmining	·	−−	+/−
SM_SO + webmining	−−	−−	+/−
SM_SW + webmining	−−	−−	+/−

Synthesis of observations from the results obtained during the experiments. We use a scale ranging from −− (for bad conformance to a certain criterion) to ++ (for good conformance). A dot (·) means neither positive nor negative and +/− signifies that the results are too much case-related to draw any significant conclusion.

Table 7.3: Comparison of the strengths and weaknesses of the static and the dynamic webmining approach.

Part III

Frequency based solutions for program comprehension

Chapter 8

Frequency Spectrum Analysis

Machines take me by surprise with great frequency.

—Alan Turing

In this chapter we look at a technique to ease the navigation of large event traces or a visualization of such an event trace, e.g. an UML sequence diagram. The technique uses the relative frequency of execution of methods or procedures within the execution of a software system to generate a visualization that we call a "heartbeat" visualization because is resembles the visualization that is typical of an electrocardiogram or ECG. With the help of the visualization it then becomes possible to navigate through the trace and identify regions in the trace where similar or identical functionality is performed.

8.1 Introduction

8.1.1 Motivation

When it comes to dynamic analysis, one of the most accessible types of information is the *execution frequency* of entities within a software system. This particular axis within the run-time information-space of software systems is commonly used in several software engineering disciplines:

- For optimization purposes the software engineer can detect frequently called (and perhaps time-intensive) entities within a software system. These particular entities can then be subjected to a closer look in order to bring about optimizations within the code of that particular

entity, because the biggest gain in performance can be obtained from optimizing these frequently called entities.

- Several virtual machine platforms employ similar schemes to detect which classes or which methods to optimize. This optimization happens mainly through the (1) inlining[1] of frequently called virtual methods or the just-in-time compilation of methods or complete classes. An example of this is the "Hot Spot" technology found in Sun's recent Java Virtual Machine implementations[2].

Even though frequency analysis has been in use within the software engineering community for some time, it was never directly used for program comprehension purposes. This changed when Thomas Ball introduced the concept of "Frequency Spectrum Analysis" (FSA) [Ball, 1999], a way to correlate procedures, functions and/or methods through their *relative* calling frequency. This correlation can e.g. happen on the basis of input data, where observations are made as to how many input-values a program receives and how many times certain procedures or methods are called internally. The same can be done for output, or one can look at relative frequencies of execution of methods or procedures that are shielded from everything that has to do with input/output.

8.1.2 Research questions

In this research track, we are looking for ways to exploit the relative execution frequency specifically for program comprehension purposes. The central research questions we have with regard to this research track are:

1. Can we use the relative execution frequency to distinguish tightly collaborating methods or procedures in a trace?
2. Can we make a visual representation of the execution trace that is at a time scalable and allows to identify these tightly collaborating entities?
3. Is it possible to use this visualization to help the end user navigate through the trace and let him/her skip parts of the trace that are similar or identical? This question can be subdivided into whether the visualization allows to discern:
   - the repetitive calling of end-user functionality (e.g. the repetition of a use case), i.e. on the macro-level.

[1] Inlining is a compiler optimization which "expands" a function call site into the actual implementation of the function which is called, rather than each call transferring control to a common piece of code. This reduces overhead associated with the function call, which is especially important for small and frequently called functions.

[2] For more information about this technology, see:
http://java.sun.com/products/hotspot/

- the repetitive calling of lower-level building blocks that are present in the application, i.e. on the micro-level.

8.1.3 Solution space

Conceptually, in object-oriented software systems, classes (or their instantiations — objects) work together to reach a certain goal, i.e. perform a certain function as specified by e.g. a use-case scenario. This collaboration is expressed through the exchange of messages between classes. This message-interaction typically occurs according to a certain interaction protocol. As such, this interaction protocol gives rise to a relationship between the two messages and the classes to which the methods belong. This relationship is also expressed through the relative execution frequency of the messages involved. It is based on this execution frequency that we will try to uncover the interaction protocol induced relationships. Furthermore, we have seen that even though the number of classes involved in an execution scenario is finite and often very limited as well, these interactions nevertheless give rise to sizeable execution traces. Intuitively, this huge size can be explained by repetitive interactions between multiple instances of classes, which furthermore strengthens the idea that execution frequency can be used to uncover interaction protocols.

Visualizations of traces, e.g. through UML Interaction Diagrams, make the trace readable, but therefore not (cognitively) scalable. A typical example of a visualization tool is IBM's Jinsight [De Pauw et al., 2001]. To ensure cognitive scalability, we ideally want to guide the end-user quickly and easily through the possibly huge execution trace (or its visualization) with the help of a heuristic [Jahnke and Walenstein, 2000]. The end-user being a software engineer trying to familiarize himself/herself with a previously unknown software system. This heuristic, based on the relative frequency of execution of methods, can help provide a program comprehension solution that helps the end-user navigate through the execution trace, by marking highly repetitive regions in the trace. These regions can be inspected and the identical or similar regions can then be quickly discarded.

We explicitly mention that we are working towards building a heuristic, because in the face of huge execution traces, thoroughness comes at a cost and the question of scalability inevitably arises [Larus, 1993,Smith and Korel, 2000]. Furthermore, when considering traditional dynamic analysis purposes such as program optimization, soundness plays a crucial role in developing a technique to guarantee behavior preservation [Mock, 2003]. Dynamic analysis for program understanding relaxes the problem considerably, because we can afford non-optimal precision.

The actual solution we propose is evolutionary with regard to the concepts presented by Ball in [Ball, 1999]. Building upon his concept of "Frequency Spectrum Analysis", we propose a scalable visualization of an execution trace. This visualization can best be described as a *heartbeat* visualization of the system, similar to the visual result of an ECG[3].

In order to try to answer the research questions within this research track, we use two open source case studies, namely Fujaba and Apache Tomcat.

8.1.4 Formal background

In a more formal way, we can say that we are actually looking for evidence of the concepts of **dominance** and **post-dominance**, borrowed from the slicing community [Tilley et al., 2005]:

> We say that an instruction x dominates an instruction y if the trace prefix which ends with y also contains an instruction x. In other words an instruction x dominates an instruction y if and only if the only way to make sure that y gets executed means that x has already been executed. x post-dominates y if every trace postfix which begins with y also contains x. Or one can say that x post-dominates y if every execution of y indicates that x will also be executed in a relatively short period of time.

8.2 Approach

The approach we follow when applying the heuristic and analyzing its results is defined as a seven-step process. This section expands on each of these steps.

Step 1: Define an execution scenario Being aware that even small software systems that are run for only a few seconds can be responsible for generating sizeable execution traces, limiting the events recorded in the execution trace is a first step towards scalability. Defining a strict execution scenario, that only exercises those use case scenarios that are of interest to the program comprehension assignment or reverse engineering context is certainly advisable. Moreover, defining a strict execution scenario helps to adhere to the goal oriented strategy we mentioned in Section 3.2.1.

[3]Electrocardiogram, the tracing made by an electrocardiograph, an instrument for recording the changes of electrical potential occurring during the heartbeat used especially in diagnosing abnormalities of heart action *(source: Merriam-Webster dictionary).*

Step 2: Define a filter A second possibility to limit the size of the execution trace is the up-front exclusion of events that lie outside our zone of interest. Good examples of such a situation are method calls to part of the system that we are not interested in, e.g. library calls. Table 8.1 shows the results of a normal tracing operation and of a tracing operation which filters out all method calls belonging to classes from the Java API[4] (Java 2 Standard Edition, release 1.4.1). This filtering operation leads to a significant reduction of the trace data, as we are able to reduce the total number of events to between 7 and 15% of the original trace.

	Jakarta Tomcat 4.1.18	Fujaba 4
Execution time (without tracing)	48s	70s
Classes (total)	13 258	15 630
Events	6 582 356	12 522 380
Unique events	4 925	858 505
Classes (filtered)	3 482	4 253
Events	1 076 173	772 872
Unique events	2 359	95 073

Table 8.1: Comparison of total tracing versus filtered tracing.

Step 3: Trace according to the scenario using the filter This step consists of running the program with an online tracing mechanism according to the previously defined execution scenario and with the tracing filter in place. The result of this step is a file which contains a chronological list of all method calls which were executed during the scenario.

Step 4: Frequency Analysis In this step, we run over the trace and we create a map which contains for every unique method found in the trace the number of times it has been called. We decided to perform this step post-mortem, i.e. after the tracing operation itself (Step 3), instead of online. The reason behind this is that by doing so, we take another measure to minimize the impact of the tracing operation on the running program, which, indeed, is already impacted by generating the trace (e.g. through the I/O cost).

Step 5: Frequency Annotation We walk over the original trace once more and annotate each event with the frequency we retrieve from the map

[4]The Java API is a standard library that contains functionality for dealing with strings, inter process communication, containers, ...

that we have created in Step 4. The result is a *chronological* list of executed methods, with an added first column which respresents the frequency of execution of the method listed in column two. Remark that the values found in the first column represent the *total* number of times a method is executed during the scenario. It is important to use the total frequency of execution because we want to distinguish methods working together based on their relative frequencies. An example can be found in Figure 8.1.

```
...
543   XMLParser.init()
978   XMLParser.parseString(String)
1243  XMLParser.closingTagFound()
1243  XMLParser.validXMLElement()
543   XMLParser.close()
...
543   XMLParser.init()
978   XMLParser.parseString(String)
1243  XMLParser.closingTagFound()
1243  XMLParser.validXMLElement()
543   XMLParser.close()
...
```

Figure 8.1: Frequency annotation example.

Please also remark from the example trace that we explicitly omit object identifiers (OID's) and parameter values, because we are looking to make an abstraction and as such, we are not interested in specific instances of interaction protocols.

Step 6: Dissimilarity Measure Using the annotated trace we sample the frequencies of a sequence of method calls, resulting in a characteristic *dissimilarity measure* for that sequence of events. Conceptually this characteristic dissimilarity measure can be compared with a fingerprint, hence its name: *frequency fingerprint*.

The sampling mechanism uses a sliding window mechanism to walk over the annotated trace. When going over the trace, we let the window fill up; once the window size is reached, we apply the dissimilarity measure on the frequencies of the events in the window and then discard the contents of the window. We repeat the process until the end of the trace is reached.

...	...	
f_{i-1}	$event_{i-1}$	
$\mathbf{f}_i$	$event_i$	} apply dissimilarity measure
$\mathbf{f}_{i+1}$	$event_{i+1}$	
$\mathbf{f}_{i+2}$	$event_{i+2}$	
$\mathbf{f}_{i+3}$	$event_{i+3}$	
$\mathbf{f}_{i+4}$	$event_{i+4}$	
f_{i+5}	$event_{i+5}$	
...	...	

We illustrate the process with a window size 5. The dissimilarity measure is applied on the frequencies of the events that lie in the interval $[f_i,\ f_{i+4}]$, after that i is incremented by 5, the window size, and the process is repeated. The implementation thus uses simple consecutive blocks for the windows.

In our experiment, we have taken the most commonly used distance metric, namely the Euclidian distance [Fraley and Raftery, 1998, Kaufman and Rousseeuw, 1990] as a dissimilarity measure to characterize how "related" the method calls within one window are.

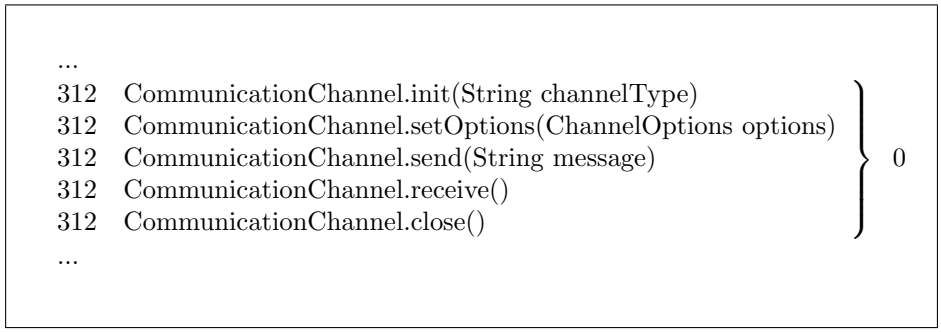
$$d = \sum_{j=1}^{w-1} \sqrt{(f_{j-1} - f_j)^2}$$

Euclidian distance: with 'w' the window size and f_j as the frequency of the j-th event in the current window on the trace.

Step 7: Analysis When the previous steps have been executed, we are in a position to analyze the dissimilarity measure and the trace looking for clues that point to interesting regions in the trace. To make this analysis step easier, we use a very simple visualization that plots the dissimilarity measure on the Y-axis for consecutive windows (X-axis). As such, the X-axis can be interpreted as being "time".

- On the one hand we are looking for regions in the trace where the frequency of execution is (almost) identical. Inspections of the traces from our case studies have learned us that these regions are often relatively small, mostly in the neighborhood of ten to thirty method calls. After which, the frequency of execution changes, before changing again to an almost identical level. Evidence of regions in the trace where the frequency of execution is identical and the resulting dissimilarity is low to near-zero, is where a frequently applied interaction protocol is used. The case studies have shown us a typical example of this, namely a wrapper construction, where an old component was wrapped. All communication from the application to that one (older) component

happened through the methods available in the wrapper. We show another example in Figure 8.2.



This annotated trace fragment shows a frequently occuring interaction protocol used for inter process communication purposes. All methods participating in the interaction protocol are executed the same number of times. When using a window size of 5, this results in a dissimilarity of 0.

Figure 8.2: Example of identical execution frequency.

- On the other hand we look for recurring patterns in the dissimilarity index. Sometimes, methods that work tightly together do not have a similar execution frequency. A typical situation of this can best be described as variation points that exist within the code. These variation points can be introduced through typical conditional constructs or through the use of polymorphism. Nevertheless, because these methods are frequently executed together, a regular pattern appears, a so-called *frequency pattern*. We have extended the example of Figure 8.1 in Figure 8.3 to show such a frequency pattern.

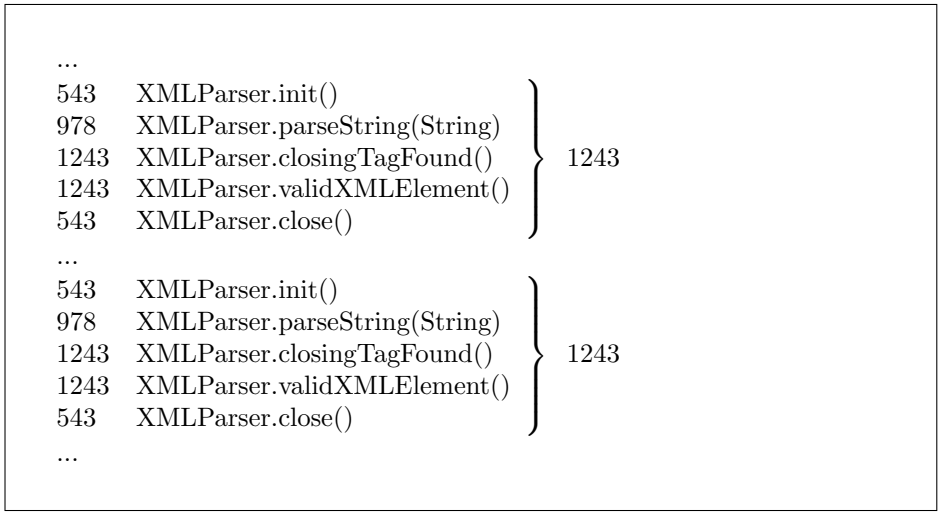
These two types of regions in the trace that carry our interest are called *clusters*.

8.3 Experimental setup

8.3.1 Hypothesis

Having explained the inner workings of the heuristic, we are now ready to formulate our four-part hypothesis.

1. The majority of the found clusters will in fact be *frequency patterns*. Frequency patterns are mostly the result of using polymorphism and because polymorphism is abundantly present in object-oriented software, we expect this type of clusters to be numerous.



Here we illustrate the concept of a *frequency pattern*, where a number of methods are frequently executed in the same order, without them being related through an identical frequency of execution. In this example we use a window size of 5 and calculate the dissimilarity value for each window.

Figure 8.3: Frequency pattern.

2. Enlarging the window size introduces noise in the frequency signatures because sequences of methods which logically form a whole are perhaps smaller than the window size. This can lead to false negatives.
3. Shrinking the window size introduces noise on the results because when frequency signatures become so small, everything becomes a frequency pattern. This can lead to false positives.
4. Regions in which a certain action is repeated become easily discernible: if at a point in time x a certain functionality is activated and at another point in time y the same functionality is activated, this will be visible in the dissimilarity values.

8.3.2 The experiment itself

We provide empirical data on and anecdotal evidence about the clusters found in the event traces in the two case-studies we used. We consider these results to be preliminary, because (1) the results have only been compared manually with the traces, (2) the validation of the results has only been done for two cases. We want to verify the results more thoroughly in another experiment which would allow us to visualize the clusters in parallel to browsing the traces in order to do a more thorough validation.

8.3.3 Case studies

We use two well-known open-source Java programs in our experiments:

1. For the representative of a non-graphical, server-like program we chose Jakarta Tomcat of the Apache Software Foundation[5]. Tomcat's origins lie with Sun Microsystems, but it was donated to the Apache open source community in 1999. Since then, the application has seen some major new releases and has been widely accepted as the reference implementation for the Java Servlet and Java Server Pages (JSP) technologies. Furthermore, it is commonly used in industrial settings in tandem with the Apache HTTP server.
2. On the other hand we have chosen Fujaba[6], an open-source UML tool with Java reverse engineering capabilities. Due to its intensive use of the Java Swing API it is an excellent representative for applications with a heavy GUI. This project originates from the University of Paderborn and has been developed by multiple students. It is frequently used as a research vehicle in the domain of UML modeling and Model Driven Engineering.

We performed three experiments. We will present them in a brief overview to give a clearer view on why we performed each of them.

1. The first experiment, performed on Jakarta Tomcat, was executed in order to validate our hypothesis about window sizes. Starting from the same event trace we used different window sizes when applying our algorithm.
2. The second experiment recreates the first one, but this time for our other case-study, namely Fujaba.
3. The third experiment on the other hand, focusses on a slightly different aspect. We wanted to know how a very specific usage scenario would be projected onto the dissimilarity graph. Therefore, we defined a usage scenario with a small number of repetitive actions in it and looked at the results of our heuristic. This experiment specifically zooms in on our third research question (see Section 8.1.2) to see whether it is possible to spot repetition at the macro-level.

As a final note we wish to add that for all three experiments we made use of the filtering technique that eliminates method calls to classes from the Java API, see also Table 8.1.

[5]More information can be found at: http://tomcat.apache.org/
In 2005 Tomcat became a project on its own and left the Jakarta umbrella. It now belongs directly to the Apache set of tools and applications.

[6]Fujaba stands for "**F**rom **U**ML to **J**ava **a**nd **B**ack **A**gain", more information on this project can be found at: http://www.uni-paderborn.de/cs/fujaba/

8.4 Results

8.4.1 Jakarta Tomcat 4.1.18

Experiment 1

As we pointed out in the previous subsection, this experiment was set up to show the results of differing the window size in our heuristic. We discuss the results of the experiment by looking at Figures 8.4 through 8.7. These figures represent the dissimilarity value of a group of methods, the current window, at a certain point in time during the execution of the program. As such, the X-axis can be interpreted as being *time*. The Y-axis then is the *dissimilarity value*.

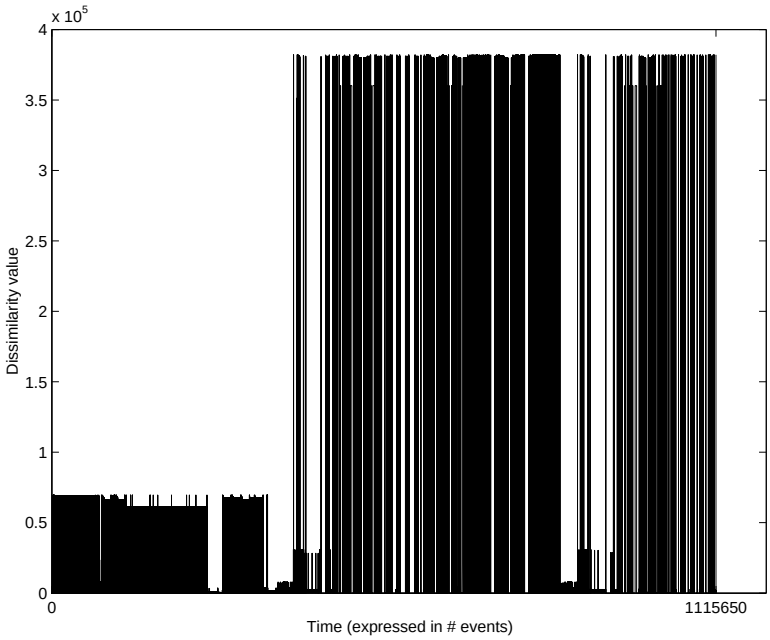
For the purpose of detecting the frequency patterns we talked about earlier, we zoomed in on an interval of the chart in Figure 8.7. The result of this is shown in Figure 8.9.

When comparing the results of our first experiment with the hypotheses we introduced in the previous section, where does this leave us?

1. From figures 8.4 through 8.7 it is clear that regions where the dissimilarity is near-zero are rather limited. In this trace we can only detect a handful of them. Frequency patterns however are much more frequent, just look at Figure 8.9: between index 86000 and 99000 on the X-axis there is a clear repetition in the dissimilarity measure.
2. Increasing the window size does not seem to have an influence on the regions with near-zero dissimilarity. This is mainly due to the fact that the execution sequences in these regions remain constant for some time, i.e., the execution pattern is longer than the (large) window size. Experimenting with window sizes in the neighborhood of 100, however, does show that noise is introduced. This is true for both the regions with near-zero dissimilarity and the frequency patterns. On the other hand, frequency patterns are more easily discernible with slightly larger window sizes: in figures 8.6 and 8.7 for example, they are much easier to spot than in figures 8.4 and 8.5.

Before going over to our second experiment, we first turn our attention to the specifics of the already mentioned frequency patterns. Some intervals show a recurring pattern in the dissimilarity measure. We took Figure 8.7 and blew up the interval [80000, 100000] for the X-axis. The result is shown in Figure 8.9.

Frequency patterns are even more interesting than the regions that have a near-zero dissimilarity value. Why? Because (1) these frequency patterns are much more common and (2) because of the polymorphic nature of object-



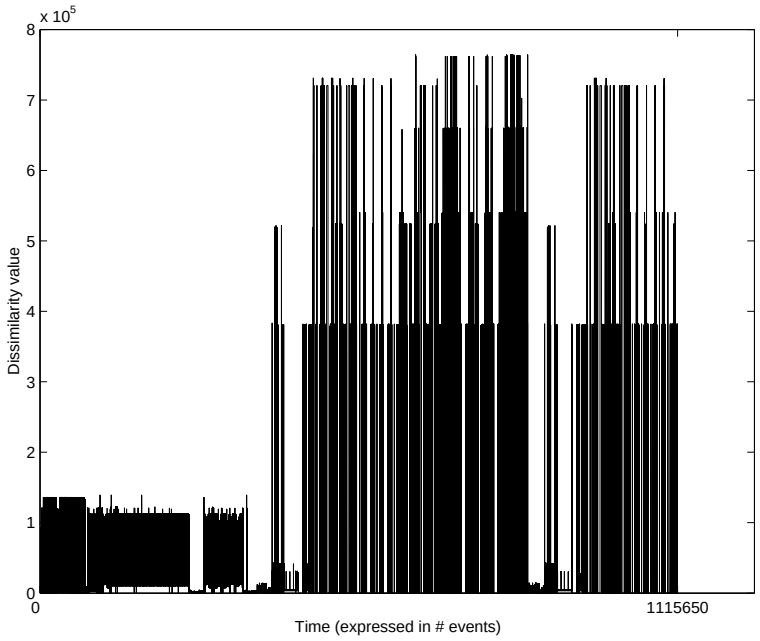
As the chart shows, until the 150.000th x-value the dissimilarity measure (Y-axis) remains low. After that there is a small period where the dissimilarity is near-zero. The interval where the dissimilarity is low, points to a high repetition of method invocations (either identical method invocations or method invocations related through their frequency of invocation). The most common instances of this kind of repetition are for example the traversal of a linked list.

Figure 8.4: Tomcat with dissimilarity measure using window size 2

oriented software, it is much more realistic to find clusters in which not every event is executed the same number of times over and over again. This can be explained by the late binding mechanism in which the exact method invocation depends on the type of data to be processed. We illustrate this with an example. Consider Figure 8.8.

In the example from Figure 8.8, after event_a and event_b have been executed, due to polymorphism there is a choice between for example events c, d or events x, y.

Suppose $f_a = f_b = f_e$ and $f_a \neq f_c$, $f_a \neq f_d$. Neither for execution sequence 1 nor execution sequence 2 would this yield a zero dissimilarity value. The chance that $f_c = f_x$ and $f_d = f_y$ is pretty slim. That is why both execution sequences give rise to a unique frequency signature. Unique, because when $f_c \neq f_x$ or $f_d \neq f_y$ they will certainly generate different values



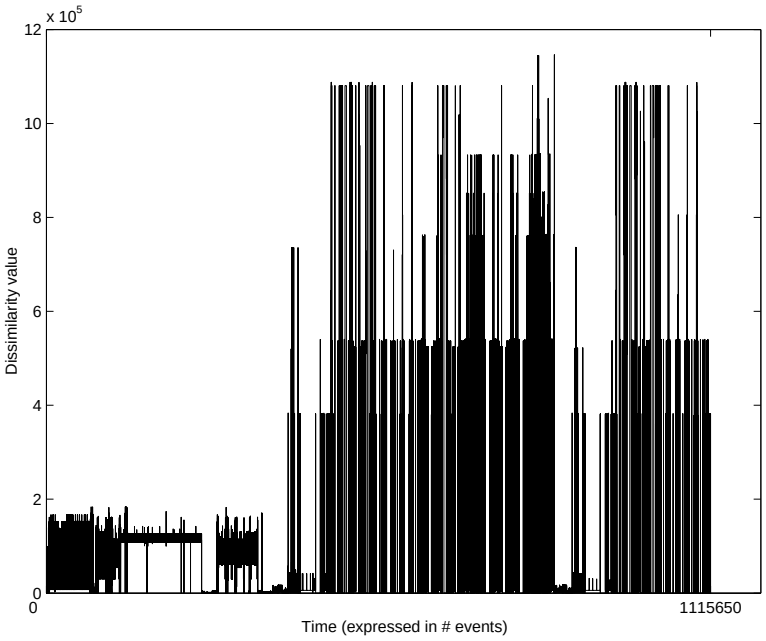
Low or almost zero values for the dissimilarity measure are still clearly visible when using a window size of 5 events. No extra places where there is a low dissimilarity value have been added, so there is no report on false positives. The false negatives did not come through either: no regions where the dissimilarity value is near-zero have disappeared with regard to Figure 8.4

Figure 8.5: Tomcat with dissimilarity measure using window size 5

for the dissimilarity measure.

Recording these frequency patterns as clusters when they tend to be present multiple times in the event trace is a good idea, because they have some interesting properties:

- They often tend to repeat themselves in the same locality.
- Manual inspection of the trace learned us that frequency patterns are much more realistic: they are not concentrated around a small number of methods and are constituted out of a variety of method invocations, often originating from many different classes. As such, these clusters are much more realistic in large-scale object-oriented systems.



When doubling the window size to 10, there is still no indication of false negatives. Intervals with low dissimilarity are still easily discernible.

Figure 8.6: Tomcat with dissimilarity measure using window size 10

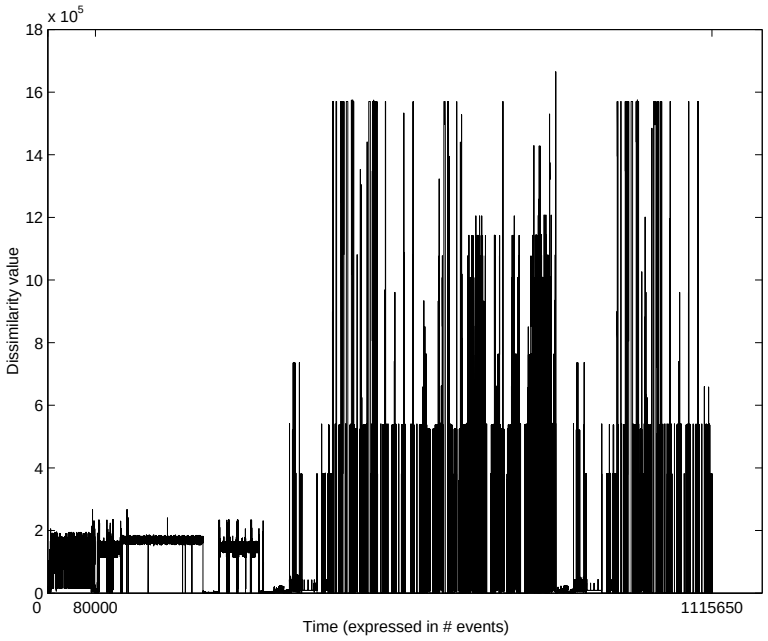
8.4.2 Fujaba 4.0

For this case-study we have opted to perform two separate experiments. One experiment is a repeat of the Tomcat experiment, but this time on Fujaba. The second is an experiment whereby a scenario with some repetitive actions is observed.

Fujaba experiment 1

For this experiment, we will not show the results for all window sizes as we did for the Tomcat case-study, but we will go straight to the largest window size, namely window size 20. In short we can say that the conclusions from the Tomcat case remain valid: medium to large window sizes remain the most interesting to distinguish the frequency patterns.

When looking at Figure 8.10, what immediately stands out is the oscillation of the dissimilarity measure in the interval [1, 35000]. From manual inspection, we learn that this behavior stems from the animated "splash



We again doubled the window size and have no indication of false negatives.

Figure 8.7: Tomcat with dissimilarity measure using window size 20

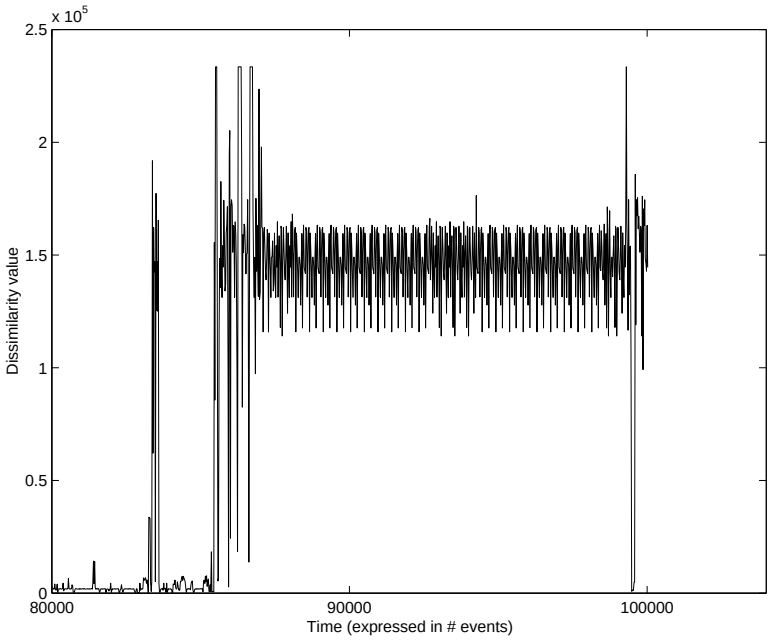
execution sequence 1	*execution sequence 2*
event_a	event_a
event_b	event_b
$\mathbf{event}_c$	$\mathbf{event}_x$
$\mathbf{event}_d$	$\mathbf{event}_y$
event_e	event_e

Figure 8.8: Example of two execution traces with possible polymorphism

screen"[7] from Fujaba. From index 35 000 onwards, we begin executing the scenario. This scenario consists of the drawing of a simple class hierarchy. Intuitively it is logical to assume that drawing a number of classes also invokes a sequence of methods the same number of times. This is exactly what Figure 8.10 shows when you look at the interval [35000, 45000].

Although this experiment is not a good example for the near-zero dis-

[7] A splash screen is an introduction screen for a program that is starting up. In the case of Fujaba it is animated and has text scrolling over it. Graphically it is quite heavy, so this can explain the heavy oscillating behavior of the dissimilarity measure.



Between time-index 87000 and 98000 there is a clear pattern of repetition, which in the middle of that interval is slightly altered. Considering the fact that this pattern ranges over around 10000 events, further investigation is warranted. Close inspection learned us that this frequency pattern is the traversal of a linked list. The slight alteration in the middle can be explained by polymorphism: not all elements in the linked list have the same dynamic type and as such, there is a slight distortation at this point.

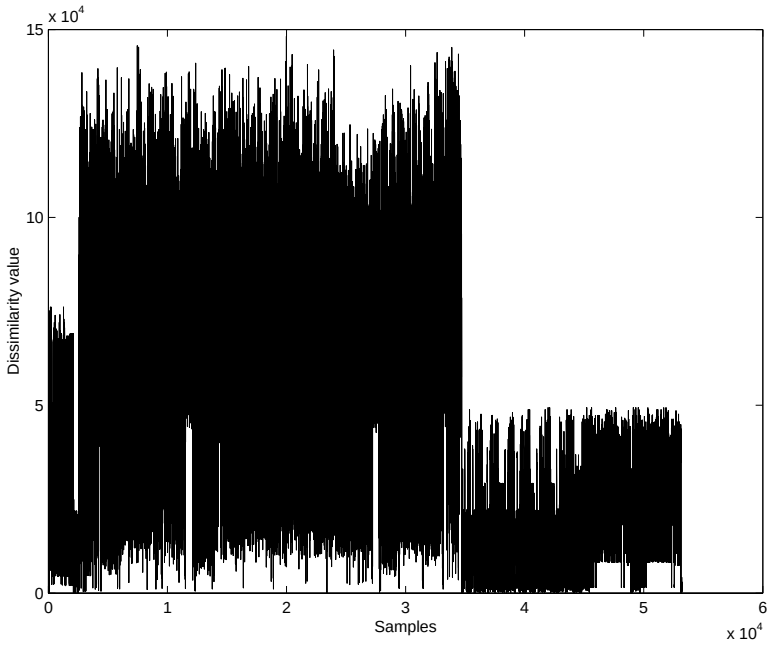
Figure 8.9: Blowup of the interval [80000, 100000] of Figure 8.7 to show frequency patterns

similarity measure, it supports the *frequency patterns* theory. The regular pattern that is visible after X-index 35000 is a good example of this.

Fujaba experiment 2

Remaining with Fujaba, we conducted a second experiment. We defined a specific usage-scenario with a highly repetitive nature. This scenario can be described as follows: after starting the program, we defined a class-hierarchy. The hierarchy consisted of one abstract base class, several child-classes, who themselves also had a number of child-classes. The total hierarchy consisted of 8 classes with a maximum nesting depth of 3.

Intuitively we expect that the visualization of the dissimilarity metric would show an 8-time repetition. Figure 8.11 shows that this is indeed the



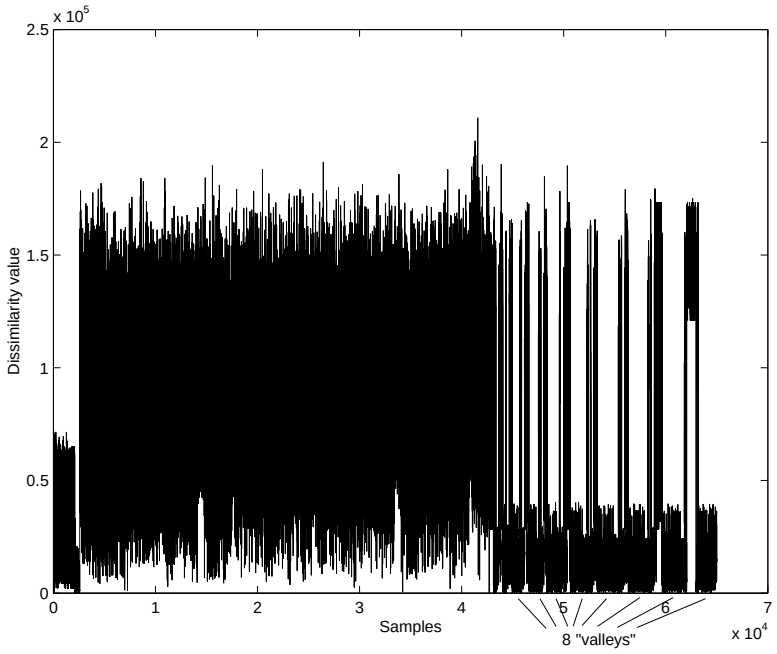
The most interesting interval is [35000, 45000]. Here we clearly see a four-time repetition pattern: first there a two repetitions, then there is a sudden drop in the dissimilarity, characterized by the thin white line in the visualization, before there is again a two-time repetition, which is identical to the first two-time repetition.

Figure 8.10: Fujaba with dissimilarity measure using window size 20

case. The graph shows 9 peaks in the dissimilarity value. Although these are interesting, we are more interested in the 8 interlying "valleys" (or depressions). The reason that these 8 regions are valleys and not peaks can be explained by the fact that the methods who are working together to draw such a class are closely related through their frequencies, which generates lower dissimilarity values.

These 8 valleys point to the functonality that is activated for drawing the class that is added to the hierarchy. Note however, how the valleys become more stretched as we add more classes to the hierarchy. Inspection of the trace showed that this is due to the *layout algorithm* which needs more actions to perform the (re)layout operation due to the higher number of objects that have to be placed.

Instead of showing listings from the actual trace to show you the repetitive nature of the actions that can be seen around the X-axis interval [44 000,



This graph shows the dissimilarity evolution of Fujaba scenario with a high degree of repetition. The executed scenario consisted of drawing a class hierarchy consisting of 8 classes. The 8 corresponding "valleys" are annotated on the graph. Note that the valleys become somewhat larger towards the end, this can be attributed to the fact that the layout algorithm has to be called more times as more objects are placed on the drawingcanvas.

Figure 8.11: Fujaba scenario with a high degree of repetition

54 000], we decided to use techniques for the detection of duplicate code. This allows us to show you that the valleys in Figure 8.11 contain a lot of repetition in the executed methods. This evidences only the repetitive nature of method invocations when performing a specific functionality. The second aspect, namely that methods working together to achieve a common goal have the same (or related) method invocation frequency became clear after manual inspection of the annotated trace (see also Section 8.2, step 4).

The duplicate code detection tool we used is called Duploc [Ducasse et al., 1999]. This tool visualizes code duplication as a dotplot. The visualization should be seen as a matrix, where both the X-axis and the Y-axis are lines in the file. Every time that an identical line is found, a black dot is placed. So, when comparing a file which contains absolutely no duplication with itself, all the dots on the main diagonal will be marked. However, when duplication is

present in the file, other dots will also be marked. For example, when the i-th line is identical to the j-th line, the dot with coordinates (i, j) will be marked black. Duploc extends this basic principle with what is called a mural view, which allows to scale the dotplot principle so that a small matrix of dots (e.g. 4) is replaced by one dot in the mural view. The color-intensity of the dot in the mural view is determined by the number of dots in the matrix that are marked. As such, the intensity can range from white (no duplication), over shades of grey, to black, when all 4 dots in the matrix indicate duplication.

The result of applying Duploc is shown in the mural view of Figure 8.12. Two interesting properties of this figure are:

1. (short) lines that run parallel to the main diagonal. This points to (quite lengthy) duplication.
2. recurring patterns in the lower right quadrant of the figure. The very similar shapes that can be spotted in the lower right quadrant also points to a lot of repetition in the execution trace.

Moreover, when we compare this with the findings from Figure 8.11 we find that the regions which are *white* in Figure 8.12 are the regions which come out as "peaks" in Figure 8.11. White regions point to no duplication. This evidences the fact that the methods which are performed during the peaks can in fact be seen as *glue code*. This is in accordance with our earlier findings from the dissimilarity value: regions with a high degree of repetition (and/or methods that work together) show a relatively low dissimilarity value.

8.5 Discussion

By analyzing the charts we have presented in this chapter, combined with the evidence we found in the execution traces and our knowledge from the internals from the case studies themselves, we have made the following observations:

1. Regions with near-zero dissimilarity value are easy to spot, even with a window size that is quite large. This means that we can easily use a big window size, thus reducing the amount of data and still find sequences of events that logically form a whole.
2. Frequency patterns are much more common than the first type of clusters. How common they are exactly is difficult to state at the moment. We presume that the size of the program, i.e. the number of classes and methods, plays a crucial role. Programs in which certain actions are performed frequently also form better candidates for detecting frequency patterns. Both Tomcat and Fujaba fall into this category. From our experiences with the two case studies presented here, our predic-

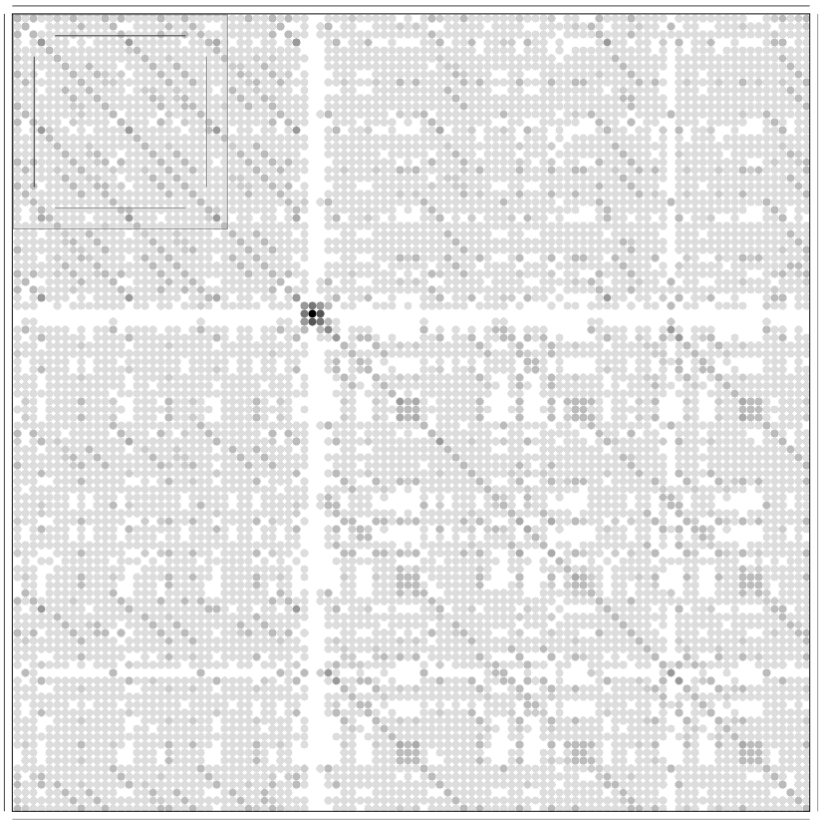


Figure 8.12 shows a mural view of the trace in the interval 44 000 till 54 000. This mural view is produced by Duploc [Ducasse et al., 1999], a tool for detecting duplicated code. In short, this technique plots a point every time a duplicate line in the event trace is found. Logically, the diagonal (from top left to bottom right) always contains such a dot. However, it becomes more interesting when you can see other lines and/or patterns in it: this points to actual duplication.

Figure 8.12: Duploc output of part of the trace (event interval 44 000 to 54 000).

tions are that of the full event trace, some 70% of the events can be catalogued as belonging to a detected cluster. This number sounds reasonable, but is nevertheless perhaps not optimal. A full 100%, however, can in our opinion never be reached because of the necessary "*glue code*" between components of a large software system.

3. The experiment in which we used a scenario with a highly repetitive nature learned us that it is quite easy to spot functionality when using

our heuristic. A groups of methods working together for reaching a common goal leave behind a very characteristic frequency pattern.

8.5.1 Connection with hypothesis

Now that we have the results from our experiment, we want to see how the results we have obtained relate to the hypothesis we set out in Section 8.3.1. We reprise our 4-part hypothesis and discuss how the hypothesis matches and diverges from the results we have obtained.

1. *The majority of the found clusters will in fact be frequency patterns.* The evidence from our two case studies does indeed point towards the fact that the frequency patterns are more numerous than then regions with near zero dissimilarity value.
2. *Enlarging the window size introduces noise in the frequency signatures.* A large window size makes the analysis-step more efficient, because there is less data for the end-user to go through. In general, frequency patterns are also easier to distinguish, but when going for a window size that is too large, frequency patterns can sometimes disappear in the visualization.
3. *Shrinking the window size introduces noise on the results.* When the windows size is set too small, patterns appear in the visualization that are not really there when browsing the actual trace. As such, very small windows sizes (e.g. window size 2) should be avoided.
4. *Regions in which a certain action is repeated become easily discernible.* As evidenced by Figures 8.9 and 8.11 this is true for the repetition of respectively use case scenarios and internal functions.

8.5.2 Connection with the research questions

In Section 8.1.2 we set out three research questions that we hoped to be able to answer within this research-track. We will recapitulate on these research questions, before going over to the actual discussion of them:

1. Can we use the relative execution frequency to distinguish tightly collaborating methods or procedures in a trace?
2. Can we make a visual representation of the execution trace that is at a time scalable and allows to identify these tightly collaborating entities?
3. Is it possible to use this visualization to help the end user navigate through the trace and let him/her skip parts of the trace that are similar or identical?

We believe that the visualization we have presented makes it possible to distinguish tightly collaborating entities. The best evidence that we have for

this claim is Figure 8.9, which visualizes an operation being performed on a self-implemented linked list. The highly repetitive nature of the heartbeat visualization typically points to tightly collaborating entities of execution. Figure 8.11 on the other hand is interesting, because the heartbeat visualization is characterized by 8 "valleys". These 8 valleys correspond to the 8-time repetition of a specific use-case. The fact that the repetition of a use-case scenario is visible as a valley in the heartbeat visualization, points in the direction that the frequency of execution of the entities participating in that use-case have a very similar frequency of execution. Similar frequencies lead to low(er) dissimilarity measures, which in turn are visualized as valleys.

Furthermore, the example of the linked list indicates that it is possible to identify repetition within a trace at the micro-level, while the 8 valley example shows that the repetition of end-user functionality can be distinguished in the visualization at the macro-level.

8.5.3 Open questions

After performing our case studies, some open questions remain.

We have not established an ideal window size, as it proved to be related to the size and structure of the program. More research however can be spent in determining a window size that is *acceptable* for a wide range of programs.

A second open question is the dissimilarity measure used. Although the Euclidian distance is the most commonly used distance metric, it is not perhaps the best one for our type of experiment [Fraley and Raftery, 1998, Kaufman and Rousseeuw, 1990]. Future experiments with different distance metrics should bring clarity here.

Part IV

Industrial experiences

Chapter 9

Industrial case studies

The outcome of any serious research can only be to make two questions grow where only one grew before.

—Thorstein Veblen

Talking about how important scalability is when performing dynamic analysis-based techniques for program comprehension does not mean too much without actually performing it on a large-scale case study. In this chapter we report on such a large scale case study. Both the coupling-based and the frequency-based approach that we have introduced earlier on will now be tried upon a large-scale industrial application. Besides presenting the results of the techniques that we have introduced earlier on, we are also reporting on some common pitfalls that occur when working in a legacy environment and more specifically on some difficulties to enable dynamic analysis in such an environment.

9.1 Motivation

As we already mentioned in Chapter 1 this research was carried out within the ARRIBA research project. This generic research project has a user-committee that is populated by industrial partners to ensure the industrial applicability of the research done by the academic partners. As such, we had the opportunity to validate our research within an industrial legacy environment.

In particular, the webmining heuristic and frequency spectrum analysis that we introduced in Chapter 5 and 8 respectively, could now be validated in

an industrial context. Both techniques were initially fine-tuned and validated using open-source case studies. Using open-source case studies for our initial experiments allowed us to (1) ensure repeatability of the experiments for the scientific community and (2) prepare this industrial experiment without burdening the industrial partners in our research project too much during the development of the heuristics. Now however, we could validate our techniques in an industrial setting.

When considering this opportunity we established 4 goals for this research track, namely:

1. We want to show the industrial relevance of the research conducted.
2. We can validate whether the techniques that were developed in the context of object-oriented software would still function correctly in a procedural context.
3. Due to the sheer size of industrial applications, we want to ensure the scalability of the proposed techniques.
4. Perform a validation with real-life developers, instead of with documentation left behind by the developers. This allows for a more interactive approach and also for feedback loops that lead back into the research and development of these techniques.

This chapter will report on our findings with regard to the 4 goals we set out.

This work has been carried out in collaboration with Bram Adams and Kris De Schutter from the University of Ghent, Belgium. Both Bram and Kris are also active in the ARRIBA research project.

9.2 Industrial partner

The industrial partner that we cooperated with in the context of this research experiment is *Koninklijke Apothekersvereniging Van Antwerpen* (KAVA)[1]. Kava is a non-profit organization that groups over a thousand Flemish pharmacists. While originally set up to safeguard the interests of the pharmaceutical profession, Kava has evolved into a service-oriented provider offering a variety of services to their pharmacist members. Amongst these services is a *tarification* service; tarification is determining the price a patient pays for his/her medication based on his/her medical insurance situation. Once the price to be paid has been established through tarification, the patient pays the pharmacist the share of the price that is not covered by the insurance, after which the pharmacist makes a claim for the other share from the insurance institution through Kava. As such they act as a financial and administrative

[1] http://www.kava.be/ (In English: The Royal Pharmacists Association of Antwerp)

go-between between the pharmacists and the national healthcare insurance institutions.

Kava was among the first in its industry to realize the need to automate this complex (tarification) process, and they have taken it on themselves to deliver this service to their members. Some 10 years ago, they developed a suite of applications written in non-ANSI C for this purpose. This suite carries the name *ICA*, an acronym for the Dutch *Informatica Centrum Apotheek*, which can be translated into "*pharmacy information processing center*".

Due to successive changes in the healthcare regulation, but also due to technology changes, the IT department at Kava is very much aware that refactoring and reengineering applications is an almost constant necessity.

Furthermore, during their recent migration from *UnixWare* for *Linux* they needed to make their application-suite ANSI-C compliant. Over the course of this migration effort, it was noted that documentation of the applications was outdated. This provided us with the perfect opportunity to undertake our experiments.

9.3 Experimental setup

Applying dynamic analysis entails the collection of run-time data. When collecting this data in a new environment, a number of technical or process related choices need to be made. This section explains some of the choices we had to make during the experiment.

9.3.1 Mechanism to collect run-time data

Introduction to aspect orientated programming

Aspect-orientation (AO) is a relatively new paradigm, grown from the limitations of Object Orientation (OO) [Kiczales et al., 1997], and a fortiori those of older paradigms. It tries to alleviate the problem of the "*tyranny of the dominant decomposition*" by proposing a solution to deal with crosscutting concerns, i.e. concerns which cannot be cleanly modularized by adhering to traditional object-oriented design principles. The proposed solution consists of the introduction of a dedicated module, called an *aspect*. More formally, aspects allow us to select by *quantification* (through *pointcuts*) which events in the flow of a program (*join points*) interest us, and what we would have happen at those points (*advice*). Hence we can 'describe' what some concern means to an application and have the *aspect-weaver* match the pointcuts to the join points and insert the advice at the appropriate place(s).

Thusfar, we only mentioned OO environments and that is also the direction AOP research was heading until recently. Nevertheless, it is important to recognize that crosscutting concerns have been in existence for many years without adequate solutions. This situation precedes the advent of object orientation and as such, deploying AOP solutions in legacy environments, seems a good idea. This was the basic premiss of the work carried out by Kris De Schutter and Bram Adams from the University of Ghent, who, in the frame of the ARRIBA project, developed *Cobble* [Lämmel and De Schutter, 2005] and *Aspicere*[2] [Zaidman et al., 2006a], AOP frameworks for Cobol and C respectively.

Why AOP?

Generating a trace in an industrial legacy environment is far from trivial. For our experiments, several constraints were in place:

- C1 The semantics of the original applications should remain intact.
- C2 We do not want to go into the original source code before applying our tools. I.e. the tools should be applicable to the source code "as is". Otherwise, we would require knowledge of what is in the sources, and this is exactly what we are trying to recover.
- C3 The tools should be deployable in other environments (operating systems, platforms, compilers, ...), so that performing other case studies or making the tools readily available to a wider audience should be no problem.
- C4 The existing build hierarchy should remain in place, with only minimal alterations. To refactor the build system, considerable knowledge of its current internals is needed, but again this is lacking.

AOP offers some interesting solutions to these constraints. Furthermore, because Aspicere, the AOP solution we used, was built to work in legacy environments, it offers additional solutions that help overcome the constraints we previously set out.

1. Constraint C1 can be overcome by carefully writing the advice body of the tracing aspect, so that one can be assured that the original semantics of the target application remain unaltered. In our particular case a tracing aspect, which outputs information when entering and exiting a procedure, was needed. This advice preserves the original semantics.
2. The base program on which the aspect-oriented solution is applied, is unaware of any changes. The AOP pointcut construct allows to *quantify*

[2]Aspicere is freely available from http://users.ugent.be/ ~badams/aspicere/

where to insert blocks of advice code. This obliviousness guarantees the satisfaction of constraint C2.

3. Aspicere is built as a preprocessor. Because the aspect-weaver acts before the actual C compiler, the result of applying Aspicere on a source file is a new source file, ready to be compiled by the platform-specific C compiler. This approach ensures constraint C3.
4. Considering the choice of a preprocessor architecture, constraint C4 can be dealt with in two ways:
   - Build an ad-hoc tool that scans the makefiles for calls to the compiler and adds a call to Aspicere, just before the call to the compiler. This can be seen as a precursor to an aspect weaver "avant-la-lettre" for makefiles.
   - Redirect all calls to the compiler, e.g. gcc, to a custom-built script that first calls Aspicere and then does the actual call to gcc. This solution is presented in [Akers, 2005].

   As we will see later on, the makefiles are characterized by a very heterogeneous structure, with calls to a variety of different compilers and tools. That is why we opted for the solution of building a simple ad-hoc tool that parses the makefiles and adds calls to Aspicere.

Tracing aspect

To collect the trace for this case study, we used two aspects: the one depicted below and a variant in which `ReturnType` is `void`.

```
ReturnType around tracing (ReturnType,FileStr) on (Jp):
    call(Jp,"^(?!.*printf$|.*scanf$).*$")
    && type(Jp,ReturnType)  && !str_matches("void",ReturnType)
    && logfile(FileName) && stringify(FileName,FileStr)
  {
      FILE* fp=fopen(FileStr,"a");
      ReturnType i;
      fprintf (fp,"before ( %s in %s ) \n",
        Jp->functionName,Jp->fileName); /* call sequence */
      fflush(fp);
      i = proceed (); /* continue normal control flow */
      fprintf (fp,"after ( %s in %s ) \n",
        Jp->functionName,Jp->fileName); /* return sequence */
      fclose(fp);
      return i;
  }
```

9.3.2 Execution scenario

Finding an appropriate execution scenario to perform a dynamic analysis solution is quite often not straightforward. Having a number of developers readily available to help with this choice is of course of great benefit. Therefore, we went along with the proposal of the developers to trace the so-called *TDFS*[3] application. The developers often use this application as a final check to see whether adaptations in the system do not have any unforeseen consequences. As such, it should be considered as a functional application, with a real-world purpose delivering the results intended, but also as a form of regression test.

The TDFS-application produces a digital and detailed invoice of all prescriptions for the healthcare insurance institutions. This is the end-stage of a monthly control- and tariffing process and acts also as a control-procedure as the results are matched against the aggregate data that is collected earlier in the process.

9.3.3 Details of the system under study

Table 9.3.3 provides some facts about the application.

Name	"ICA"
Number of C modules	407
LOC	453 000 (non-comment, non-blank)
Build process	GNU make, hierarchy consisting of 269 individual makefiles
Current build platform	Linux: vanilla Slackware 10.0
Status	in use for $>$ 10 years

Table 9.1: System passport

9.4 Results

This section will cover the results we have obtained from applying frequency spectrum analysis and webmining on the trace we have obtained from running the TDFS application according to the execution scenario that was provided to us by the developers.

[3]TDFS is an acronym for the Dutch *Tariferings Dienst Factuur (en) Statistiek(spoor).* Freely translated this would be "Tarification Service for Invoices and Statistics" in English.

9.4.1 Experimental setup of the validation phase

For the particular application we considered, TDFS, two developers were available at Kava. From now on we will call them D_1 and D_2. Both have a thorough knowledge of the structure and the inner workings of this particular application.

Before we discussed our findings of their application with the developers, we interviewed the developers separately. During this interview we used a schema were we asked three questions about the 15 modules belonging to the TDFS application:

1. Which module is the most essential?
2. Which module tends to contain most bugs?
3. Which module is the hardest to debug?

We noted their answers and also asked if there were any particular reasons why they believed a certain module to be important, hard to debug or to contain bugs. This questionnaire was particularly useful to validate the results we had obtained from the webmining approach.

We then presented the results we had obtained, technique by technique, to each of the two developers separately and wrote down their reactions, questions and/or suggestions. Afterwards, during a short session we discussed the results with both developers and highlighted similarities and differences in their answers and/or reactions.

During the final stage of our experiment there was a feedback loop back to the Kava development team in which we discussed a number of constructs that could be removed from the code in order to make future maintenance easier.

9.4.2 Webmining

Resultset

Table 9.2 lists the results of applying the webmining heuristic to the Kava case study. The classes (1st column) are ranked according to their *hubiness* value (2nd column). Due to the normalization, all hubiness values lie in the range $[0, 1]$.

Some important facts that can be derived from Table 9.2 are:

- the heuristic clearly makes module `e_tdfs_mut1.c` stand out.
- only 7 out of the 15 modules have a value greater than zero. Modules with a hubiness value of zero, do not call other modules. As such, import coupling for these modules is non-existant[4], while export coupling

[4]Import coupling measured within the full ICA project. Import coupling could exist

Module	Value
e_tdfs_mut1.c	0.814941
tdfs_mut1_form.c	0.45397
tdfs_bord.c	0.397726
tdfs_mut2.c	0.164278
tools.c	0.164278
io.c	0.12548
csrout.c	0.0321257
tarpargeg.c	0
csroutines.c	0
UW_strncpy.c	0
td.ec	0
cache.c	0
decfties.c	0
weglf.c	0
get_request.c	0

Table 9.2: Results of the webmining technique

levels are moderate to high.

- the 4 modules that are specific to the TDFS application show up in the 4 highest ranked places.

Discussion with developers

D_1 mentioned `e_tdfs_mut1.c` and `tdfs_mut2.c` as being the most essential modules for the TDFS application. `io.c` and `cache.c` are also important from a technical point of view, but are certainly not specific to the TDFS application, as they are used by many other applications of the system. D_1 was actually surprised at the fact that `cache.c` was not catalogued as being more important. `csrout.c` and `csroutines.c` are difficult to debug, but they have only once had to change some details in these file in a time period of 10 years.

D_2 clearly ranks the `e_tdfs_mut1.c` module as being the most important and most complicated module: it contains most of the business logic. `tdfs_mut2.c` makes a summary of the operations carried out by `e_tdfs_mut1.c` and checks the results generated by `e_tdfs_mut1.c`. `tdfs_mut1_form.c` is mainly responsible for building up an interface for the end-user, while `tdfs_bord.c` is concerned with formatting the output.

with regard to external libraries.

Discussion

As such, the opinions of D_1 and D_2 are indeed very similar. D_1 ranks `e_tdfs_mut1.c` and `tdfs_mut2.c` as being most important, D_2 points to `e_tdfs_mut1.c` as being the most important module.

The resultset of our own technique (see Table 9.2) clearly ranks `e_tdfs_mut1.c` as being the most important module in the system. Furthermore, all modules that are specific to this application appear at the top of the ranking.

Drawbacks – threats to validity

From the resultset of this case study, we noted two drawbacks:

- Classes or modules that are *containers*, i.e. data-structures with a number of operations defined on them, are often ranked very low by our heuristic. This can be explained by the fact that these modules are often self-contained, i.e. they do not rely on other classes or modules to do their work. As a consequence, these classes often have a high level of export coupling and a low level of import coupling. The webmining algorithm reacts to this by attributing these classes with a low hubiness value and as such, a low ranking amongst other (non-container type) classes.
  These properties explain why `cache.c` – a caching data-structure – which was expected to rank higher according to D_1, is ranked quite low.
- This particular case actually also serves as a counterexample. Our heuristic places `e_tdfs_mut1.c`, `tdfs_mut1_form.c`, `tdfs_bord.c` and `tdfs_mut2.c` at the top of the ranking. It are exactly those four modules that are specific to the TDFS application, so a simple analysis of naming conventions would have sufficed in this particular case.

9.4.3 Frequency analysis

Due to the huge size of the event trace (90GB $\approx 4.86 \times 10^8$ procedure calls), the visualization we presented in Chapter 8, did not scale up to this huge amount of data. Therefore, we opted for a slightly different solution. We still use frequency of execution as the underlying model, but summarize the results before visualizing.

A fragment of the result is shown in Figure 9.1, the full resultset can be found in Appendix B. Figure 9.1 depicts three *"frequency clusters"*. Each cluster shows the total execution frequency, and the procedures that fall into this frequency interval. Different kinds of boxes can be perceived, to indi-

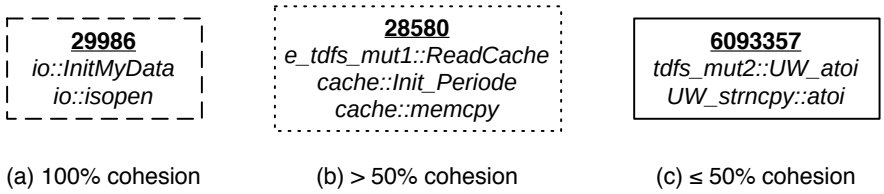


Figure 9.1: Three frequency clusters from the TDFS application

cate the level of cohesion within a frequency cluster: a box with a full line (Figure 9.1.c) indicates that $\leq 50\%$ of the methods in the cluster come from the same module, a dashed line (Figure 9.1.b) indicates total cohesion as all procedures belong to the same module. A dotted line (Figure 9.1.a) meanwhile indicates a level of cohesion within the frequency-cluster of between 50 and 100%.

In total, 237 unique procedures were executed during the scenario. Of these, 160 could be clustered into 25 frequency-clusters (these can be found in Appendix B. In other words, 67.5% of the procedures could be catalogued in clusters. When considering the cohesion of each of these frequency-clusters, we have the following distribution: two of these clusters had a full line, i.e. they did not show cohesion. 12 had a dashed line, meaning that all procedures within a frequency-cluster originated from a single module, while the 11 others had a dotted line, also indicating a strong level of cohesion.

This technique provides an easy way to find procedures that share common goals, because they are related through their frequency of execution. Furthermore, it allows to easily audit the system when it comes to cohesion.

Discussion with the developers.

D_1 immediately remarked that one of the two frequency clusters with a full line, i.e. a cluster with a limited degree of cohesion, was actually a wrapper construction they had hastily constructed when performing the migration from UnixWare to Linux.

The clusters found did not surprise the developers either.

Discussion

For our particular case study, 48% of the clusters were found to be fully cohesive. These fully cohesive clusters are accountable for 20% of the procedures. 44% were found to be strongly cohesive; these clusters contain 49%

of the total number of procedures. The largest non-cohesive cluster had a frequency of execution of *1*, consisting mainly out of procedures with initialization functionality. The other non-cohesive cluster was the one that caught D_1's attention for containing wrapper functionality.

As such, we can conclude that the system is actually well-structured, as most clusters were cohesive and these account for 70% of all procedures.

9.5 Pitfalls

This section describes some unexpected experiences we had while performing our dynamic analyses in the legacy context we described in Section 9.2. Some of these experiences seem to be closely related with the usage of AOP for collecting our traces, but we have strong indications that other trace-collection mechanisms, e.g. AST rewriting techniques as presented by Akers [Akers, 2005], suffer from similar problems when applied in similar conditions.

This section describes some of our experiences and how we coped with them.

9.5.1 Adapting the build process

The Kava application uses `make` to automate the build process. Historically, all 269 makefiles were hand-written by several developers, not always using the same coding-conventions. During a recent migration operation from *UnixWare* to *Linux*, a significant number of makefiles has been automatically generated with the help of *automake*[5]. Although a sizeable portion of the 269 makefiles are now generated by automake and thus have a standardized structure, a number of makefiles still have a very heterogeneous structure, a typical situation in (legacy) systems.

We built a primitive tool, which parses the makefiles and makes the necessary adaptations so that our AOP solution Aspicere is applied on each source-code file, before this file is compiled. A typical before and after example of the necessary makefile modifications is shown in Figures 9.2 and 9.3. However, due to the heterogeneous structure of a portion of the makefiles, we were not able to completely automate the process, so a number of makefile-constructions had to be manually adapted.

The adaptations to be made become more difficult, when e.g. Informix esql preprocessing needs to be done (see Figures 9.4 and 9.5).

[5] `Automake` is a tool that automatically generates makefiles starting from configuration files. Each generated makefile complies to the GNU Makefile standards and coding style. See http://sources.redhat.com/automake/.

```
$(CC) -c -o file.o file.c
```

Figure 9.2: Original makefile.

```
$(CC) -E -o tempfile.c file.c
cp tempfile.c file.c
aspicere -i file.c -o file.c \
    -aspects aspects.lst
$(CC) -c -o file.o file.c
```

Figure 9.3: Adapted makefile.

```
.ec.o:
   $(ESQL) -c $*.ec
   rm -f $*.c
```

Figure 9.4: Original *esql* makefile.

```
.ec.o:
   $(ESQL) -e $*.ec
   chmod 777 *
   cp `ectoc.sh $*.ec` $*.ec
   $(ESQL) -nup $*.ec $(C_INCLUDE)
   chmod 777 *
   cp `ectoicp.sh $*.ec` $*.ec
   aspicere -verbose -i $*.ec -o \
      `ectoc.sh $*.ec` \
      -aspects aspects.lst
   $(CC) -c `ectoc.sh $*.ec`
   rm -f $*.c
```

Figure 9.5: Adapted *esql* makefile.

Our tool takes only a few seconds to go over the 269 makefiles and make the necessary alterations. Detecting where exactly our tool failed through makefile code inspections took several hours and even some build cycles were lost because of remaining errors in the makefiles.

This tool, however primitive, can be seen as an AOP solution for makefiles "avant-la-lettre". We used simple string-matching to detect places where the compiler was called and inserted an extra call to Aspicere *before* the actual call to the compiler.

9.5.2 Legacy issues

... impacting quality

Even though Kava recently migrated from UnixWare to Linux, some remains of the non-ANSI implementation are still visible in the system. In non-ANSI C, method declarations with empty argument list are allowed. Actual declaration of their arguments is postponed to the corresponding method definitions. As is the case with ellipsis-carrying methods, discovery of the proper argument types must happen from their calling context. Because this type-inferencing is rather complex, it is not fully integrated yet in Aspicere.

Instead of ignoring the whole base program, we chose to"skip" (as yet) unsupported join points, introducing some errors in our measurements. To be more precise, we advised 367 files, of which 125 contained skipped join points (one third). Of the 57015 discovered join points, there were only 2362 filtered out, or a minor 4 percent. This is likely due to the fact that in a particular file lots of invocations of the same method have been skipped during weaving, because it was called multiple times with the same or similar variables. This was confirmed by several random screenings of the code. These screenings also showed that there is no immediate threat to the validity of this particular experiment (as the skipped join points were not located in files that belonged to the TDFS package). Nevertheless, similar situations in other cases could impact the validity of the resultset.

... impacting performance

Another fact to note is that we constantly opened, flushed and closed the tracefile, certainly a non-optimal solution from a performance point of view. Normally, Aspicere's weaver transforms aspects into plain compilation modules and advice into ordinary methods of those modules. So, we could get hold of a static file pointer and use this throughout the whole program. However, this would have meant that we had to revise the whole make-hierarchy to link these uniques modules in. Instead, we added a "legacy" mode to our weaver in which advice is transformed to methods of the modules part of the advised base program. This way, the make-architecture remains untouched, but we lose the power of static variables and methods.

9.5.3 Scalability issues

Compilation

A typical compile cycle of the original application consisting of 407 C modules (453 KLOC in total) takes around 15 minutes[6]. With the introduction of the AOP solution into the build process, the compile cycle now looks like:

1. Preprocess
2. Weave with Aspicere
3. Compile
4. Link

While the original compile cycle for the whole system took 15 minutes, the new cycle lasts around 17 *hours*. The reason for this substantial increase in time can be attributed to several factors, one of which may be the time

[6]Timed on a Pentium IV, 2.8GHz running Slackware 10.0

needed by the inference engine for matching up advice and join points. There is also evidence that a lot of backtracking takes place, but the currently used Prolog engine [Denti et al., 2001] does not process this in an optimal way.

Running the program

Not only the compilation was influenced by our aspect weaving process. Also the running of the application itself. The scenario we used (see Section 9.3.2), normally runs in about 1.5 hours. When adding our tracing advice, it took 7 hours due to the frequent file I/O.

Tracefile volume

The size of the logfile also proved problematic. The total size is around 90GB, however, the linux 2.4 kernel Kava is using was not compiled with large file support. We also hesitated from doing this afterwards because of the numerous libraries used throughout the various applications and fear for nasty pointer arithmetic waiting to grab us. As a consequence, only files up to 2GB could be produced. So, we had to make sure that we split up the logfiles in smaller files. Furthermore, we compressed these smaller logfiles, to conserve some diskspace.

Once compressed with `gzip`, the 90GB of data was reduced to approximately 620MB. 90GB of trace data stands for approximately 9.72×10^8 events (calls and exits), which means that there are approximately 4.86×10^8 procedure calls.

Effort analysis

Table 9.3 gives an overview of the time-effort of performing each of the analyses. As you can see, even a trouble-free run (i.e. no manual adaptation of makefiles necessary) would at least take 29 hours, when performing one analysis, and would take slightly under 40 hours when performing all analyses consecutively. Of course, some speed-ups can be obtained from running the two analyses in parallel.

9.6 Discussion

This chapter reports on a research track where we applied our recently developed techniques in an industrial legacy C context. This section presents a discussion on the goals of this research track and highlights strengths and weaknesses of the approach we have taken.

Task	Time	Previously
Makefile adaptations	10 s	–
Compilation	17h 38min	15min
Running	7h	1h 30min
Frequency analysis	5h	–
Webmining	10h	–
Total	39h 38min 10s	1h 45min

Table 9.3: Overview of the time-effort of the analyses.

Goals

In Section 9.1 we set out 4 goals for this research track. During this discussion, we will again focus on each of these goals and see whether we achieved what we set out to do.

- **Industrial relevance of the research conducted.**
  Our experiments do not give conclusive evidence on whether there is strong industrial relevance for knowing which classes or modules are essential during early stages of program comprehension. The developers at Kava did point out however, that such information is probably useful when instructing a new co-worker, who is unfamiliar with the project.
- **Validation of techniques in procedural programming context.**
  As a third case study for both the frequency analysis and the webmining technique, the general tendencies that we had found with the previous case studies is confirmed, even though this is the first case study in a procedural programming language context.
- **Scalability of proposed techniques.**
  Scalability is often seen as the major stumbling block when performing dynamic analysis [Larus, 1993]. We have taken special care during the design and development of our techniques to make them scalable. With regard to this aspect, we discuss the frequency analysis and the webmining techniques separately:
  - During the case study the webmining technique scaled more than adequately to the challenge of providing a resultset for an industrial medium-scale legacy application. In absolute terms, the 10 hour wait before the results are available is long, but we also have to consider that just reading the 90 GB of data from file takes a long time, even without performing any computation. Furthermore, we are also aware of a number of possible optimizations that we could perform on the algorithm, but at this point these remain

untested.
  - With regard to the frequency analysis technique, we were disappointed to see that even though the basic frequency analysis technique works finely, generating the visualization proved unsuccessful due to the immense size of the trace. The scalability problem we encountered has two distinct facets to it: on the one hand, the visualization could not be visualized in its entirety, due to memory problems, while on the other hand, even if we could have visualized it, the resulting visualization would have become overwhelmingly large, which would have negatively impacted the cognitive scalability.
- **Validation of resultset with real-life developers.**
  Having several developers (two in our case) cooperate, implies that several opinions exist regarding the importance of certain classes or modules in a system. However, in this case study, the general direction was clear and there were no major discrepancies in the views of both developers. Furthermore, the modules they pointed out as being most important, were the same modules that our webmining technique ranked at the top.

Results

From the resultsets we obtained from our dynamic analysis experiments, we can conclude that:

- The webmining approach results in a ranking of modules according to their importance from a program comprehension point of view. Interviews with the developers fully confirm the results that our heuristic delivered. The only false negative we could note, was a container class that the developers deemed important, but was judged as being unimportant by our technique. This is due to the low to non-existent level of import coupling for this particular module.
- The frequency analysis approach allowed to easily audit the system's internal structure. We found that most of the modules are (strongly) cohesive, which indicates that the structure is well balanced and reuse is a definite possibility. The developers agreed with our views and told us that many modules are frequently reused.

Technical limitations

As a vehicle to perform our dynamic analysis, we used Aspicere, which allowed us to use the clean and non-intrusive, yet powerful mechanism of As-

pect Orientation to trace the entire application.

As a clear downside of our approach, we should note the effort it takes to perform the entire analysis. If no problems are encountered, the entire analysis we described takes around 39 hours, for a system that should be considered as medium-scale. As such, we acknowledge that we should improve the efficiency of our tools.

Part V

Concluding parts

Chapter 10

Related Work

Programmers have become part historian, part detective, and part clairvoyant.

— Thomas A. Corbi

"Program understanding: Challenge for the 90s" is the title of a paper published in 1990 by Thomas Corbi in the IBM Systems Journal [Corbi, 1990]. In this paper he reminds us that a significant gain in efficiency can be attained when the program comprehension process can be stimulated. No wonder then, that over the last few years, program comprehension has gained much attention and has been — and still is — an active area of research. In this chapter, we will discuss some of these past and current research efforts.

10.1 Dynamic analysis

Dynamic analysis techniques come in many forms and usually they also all have slightly different goals. Some of the techniques focus on retrieving features from execution traces, others aim at performing a clustering of a static representation of a software system with the help of dynamic information. In this section we discuss a variety of dynamic analysis based techniques, which almost all share a common theme: helping the user to better understand the software system, by presenting the user with an acceptable amount of information.

Greevy Greevy is working on a solution whereby the features of a software system can be correlated to classes and vice versa. To do this, she uses

feature-traces, which are execution traces that are the result of executing a very specific feature (or a very small set of features). When a number of these features trace are available, she is able to classify classes as being responsible for only one feature, a set of features or all features available in the system. Vice versa, she also catalogs features that are demanding services from one method or class, a number of methods or classes or all methods or classes in the system [Greevy and Ducasse, 2005].

Hamou-Lhadj Hamou-Lhadj has proposed several solutions to overcome the scalability issues surrounding dynamic analysis. One of the solutions he has been working on is to automate the selection process of which classes (or other entities) to include in the execution trace and the subsequent analysis. Where in our experiments we explicitly did not trace any classes that are part of the standard library, the solution provided by Hamou-Lhadj would automate this up to a certain point. The basic idea is to detect those classes and entities in the software system that can be classified as utility components and subsequently remove them from the analysis process. The basic means by which these utility components are detected is a fan-in analysis [Hamou-Lhadj et al., 2005].

Another solution Hamou-Lhadj has presented is *trace summarization*. He describes how a number of concepts that are also used when summarizing natural text can be helpful when trying to summarize execution traces, e.g. by extracting important methods based on naming conventions [Hamou-Lhadj and Lethbridge, 2006].

Furthermore, Hamou-Lhadj also advocates the use of a meta-model to store dynamic runtime information from object-oriented systems, which is termed the *Common Trace Format* or CTF [Hamou-Lhadj, 2005b, Hamou-Lhadj and Lethbridge, 2004].

Mancoridis et al Based on the clustering tool *Bunch*, which was developed by Mancoridis et al [Mancoridis et al., 1999], Gargiulo and Mancoridis developed *Gadget*, a tool to cluster the entities of a software system based on dynamically obtained data [Gargiulo and Mancoridis, 2001]. The goal of using Gadget is to make the often complex structure of software systems more explicit and easier to understand.

Gadget builds up a dynamic dependency graph, a graph in which classes are represented as nodes and calling relationships as edges. These calling relationships are extracted from the obtained execution trace. On this graph then, they apply Bunch, which delivers a clustering of the original graph. This approach is very similar to what our webmining approach does with the

compacted call graph (see Chapter 5).

Because of the similarities between our own approach and the approach from Gadget, we did an initial experiment to see whether the clusters that were identified by Bunch had their counterparts in the resultset of our web-mining approach. To our surprise, there was no clear match between these two resultsets and as such, we see the further analysis of these two techniques as an important direction for future research.

Richner et al Richner's and Ducasse's approach is based on storing both statically and dynamically obtained information from a software system in a logic database [Richner, 2002]. First, static and dynamic facts of an object-oriented application are modeled in terms of logic facts, after which queries can be formulated to obtain information about the system. As a case study they use HotDraw implemented in Smalltalk [Richner and Ducasse, 1999].

In order to overcome scalability problems, they advocate an iterative use of the technique. This means that when having obtained a (high-level) view of the software through queries, the results of this view are used to restrict the tracing operation to the parts of the software that you are trying to focus on. This allows for a refinement of the views obtained from using the tool.

The *Collaboration Browser* tool that they describe is explicitly targeted at recovering collaborations between classes, without having to rely on visualization techniques [Richner and Ducasse, 2002]. Its focus is on understanding the system in the small, rather than understanding the system as a whole. The underlying model that is built around dynamically gathered information, is queried using pattern matching criteria in order to find classes and interactions of interest.

Systä To overcome the scalability issues of analyzing large execution traces through variations of Jacobson interaction diagrams [Jacobson, 1995], Systä uses the SCED environment to synthesize state diagrams from interaction diagrams [Systä, 2000b, Systä, 2000a]. State diagrams, which are a variation on UML statechart diagrams, allow to observe the total behavior of an object, while interaction diagrams focus more on sequential interactions between several objects.

Another research path Systä follows is the combination of static and dynamic information [Systä, 1999]. One of the observations made is that when combining static and dynamic information, one has to choose very early on which of these two sources of information will be the base layer and which approach will be used to augment this base layer. The experiment described deals with Fujaba, which is reverse engineered with the help of the Rigi static

reverse engineering environment [Wong et al., 1995] and is augmented with dynamic information [Systä, 1999].

10.2 Visualization

Using dynamic analysis for program comprehension purposes means that you have to work your way around the often sizeable sets of dynamic information that get collected during a program run. A possible solution to overcome the size of these sets of information is through a well thought-out visualization. This section describes some of the most common visualization-oriented research ideas.

De Pauw et al De Pauw et al are known for their work on IBM's *Jinsight*, a tool for exploring a program's run-time behavior visually [De Pauw et al., 2001]. Jinsight is a research prototype that first emerged from IBM's T.J. Watson Research Center in 1998. Since then a number of its features have been adopted in the *Hyades* plugin for the Eclipse Java IDE. In 2005 this plugin was absorbed into the Eclipse *Test & Performance Tools Platform* (TPTP).

One of the main program comprehension applications of Jinsight (and its derivatives) is the generation of Jacobson interaction diagrams [Jacobson, 1995], similar to UML's sequence diagrams. Even though this visualization is much more scalable then previous solutions to visualize execution traces, there is still room for improvement. A more scalable visualization is proposed by De Pauw with the concept of the *execution pattern notation* [De Pauw et al., 1998]. See Figure 10.1 for an example.

Other possible uses of Jinsight and its derivatives are: following the behavior of multi-threaded object oriented programs, detecting memory leaks, detecting hotspots, etc.

Jerding et al Jerding et al have developed a tool called ISVis (Interactive Scenario Visualizer) [Jerding et al., 1997]. One of its possible usages is to help alleviate the architecture localization problem, or the problem of finding the exact location in a system's architecture where a specific enhancement can be inserted into the system [Jerding and Rugaber, 1997].

ISVis generates views of execution traces that are similar to Jacobson interaction diagrams [Jacobson, 1995]. The tool environment however allows to make a more compact visualization by e.g. grouping together classes in package-like structures, by removing utility classes, etc. Furthermore, it allows to visually identify similar (sub)scenarios in execution traces and has

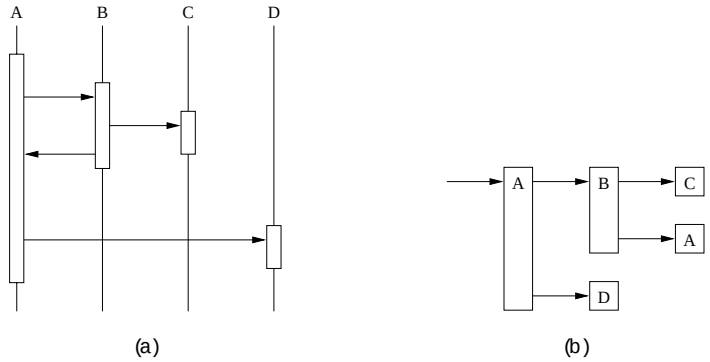


Figure 10.1: Simple interaction diagram (a) and its corresponding execution pattern (b) [De Pauw et al., 1998]

limited capabilities to recognize these similar scenarios automatically through pattern matching. Another feature that helps improve scalability is a mural view that portrays global overviews of scenarios [Jerding and Stasko, 1998]. For completeness sake, we mention that the approach of ISVis is actually a hybrid approach, wherein static and dynamic analysis are combined.

Ducasse et al Ducasse, Lanza and Bertuli describe how they use polymetric views, as used in the CodeCrawler tool [Lanza, 2003], to visualize a condensed set of run-time information [Ducasse et al., 2004]. Using a condensed set of information means that there is no need to keep and analyze the complete trace, rather their approach is based on collecting measurements during the execution, such as the number of invocations, the number of object creations, the number of used classes/methods, etc.

With these run-time measurements, they are able to provide insight into a system in a relatively lightweight manner. They present their results in three different polymetric views, namely:

- *Instance usage view*: shows which classes are instantiated and used during the system's execution.
- *Communication interaction view*: shows the (strength of) communication between classes of a system during its execution.
- *Creation interaction view*: shows the number of instances a class creates and the number of instances each class has.

Reiss and Renieris Reiss and Renieris describe several techniques to encode program executions [Reiss and Renieris, 2001]. Their main concern is

to offer a way to compact the trace. They use basis mechanism such as run-length encoding and grammar-based encoding to shorten the trace.

Another approach they discuss is *interval compactation*. For this approach, they break the execution trace into a small set of intervals (for example 1024 events) and then do a simple analysis within each of the intervals to highlight what the system is doing at that point. Although they remain quite vague about the inner-workings of their algorithm, the resulting visualization and the ideas behind it have a ressemblance to our own heartbeat visualization that we use in combination with our frequency spectrum analysis.

Walker et al Walker et Al describe a visualization that has a temporal component to it [Walker et al., 2000] [Walker et al., 1998]. The visualization consists of a temporally-ordered series of pictures, so-called *cells*, each detailing information about a corresponding point in time in the execution of the system being analyzed.

10.3 Industrial experiences

Wong et al Wong et al describe their experiences with re-documenting industrial legacy applications with the help of their *Rigi* static reverse engineering environment [Wong et al., 1995]. They have applied Rigi on COBOL, C and PL/AS[1] systems. The PL/AS experiment described in [Wong et al., 1995] exhibits a close resemblance with our own experiments, as the goals and setting were very similar: a large scale industrial legacy application with 2M LOC and 1300 compilation units (here written in a proprietary language, not ANSI-C). Because of the large scale of the application, they also focussed on delivering scalable reverse-engineering techniques. One of the most significant lessons they learned from their experiments is that in-the-large design documents describing the architecture of the software system's current state can be very beneficial for building up understanding of a software system and maintaining it. Furthermore, they have followed a path similar to ours when it comes to validating their approach, namely by involving the developers and maintainers and checking whether the mental models from the developers and maintainers concur with the information they retrieved. Another similarity with our own experiences is the effort it takes to perform their analysis, although we must remark here that the available computing power in 1995 is likely to be different from that available 10 years later.

[1] Programming Language/Advanced Systems (IBM).

Chapter 11

Conclusion

I hope you become comfortable with the use of logic without being deceived into concluding that logic will inevitably lead you to the correct conclusion.

—Neil Armstrong

This chapter presents our conclusions with regard to the heuristics we have developed and the experiments we have undertaken. Furthermore, it provides a number of possible directions for future research.

11.1 Conclusion

In our hypothesis (see Chapter 1) we state that within the run-time information space two axes, namely *dynamic coupling* and *relative frequency of execution*, are good candidates to develop heuristics for program comprehension purposes. We now discuss our experiences for each of these two axes separately.

11.1.1 Dynamic coupling

The heuristic that uses dynamic coupling measures, allows to identify the most need-to-be-understood classes in a system. Detecting these classes very early on in the program comprehension process allows the end user to direct his/her attention towards these classes and start exploring the software system from there.

We experimented with a number of different dynamic coupling metrics and also compared direct and indirect coupling solutions. To simulate this

indirect coupling, we used the HITS webmining algorithm. Our experiments have shown that taking indirect coupling into account delivers the best results.

Using publicly available extensive documentation of two open source case studies, we have performed an intrinsic evaluation of this approach. The validation has learned us that we are able to recall 90% of the classes marked as need-to-be-understood by the developers, while maintaining a precision of 60%. These results are completely satisfactory, although in an ideal situation, we would have liked to have an even higher level of precision.

We have also applied this technique on an industrial legacy C environment, where the approach again delivered good results in the sense that the modules that the developers designated as being important were ranked very high in the resultset of our approach.

With regard to scalability, our main point of focus, we have a somewhat mixed image. Our approach allows to process huge (e.g. 90 GB) event traces, but of course, this takes time to process (in our industrial case study 10 hours). We believe that our approach can still be optimized, but we have to be realistic in the fact that processing gigabytes of event traces will always take time, as will the collection of the execution trace. On the cognitive scalability front, we are very much pleased that our resultset is concise, while still being relatively precise.

As a control experiment to see whether the effort of using dynamic information is indeed beneficial, we have experimented with applying the same basic technique on statically collected coupling data. While a slight improvement in round-trip-time could be noted, we were also confronted with a drop in recall from 90% dynamically to 50% statically. Precision fell similarly from 60% to 8%. This clearly indicates that using dynamic analysis, with its goal oriented strategy, pays dividends when used for program comprehension purposes.

11.1.2 Relative frequency of execution

Through the heartbeat visualization that we have obtained with building a heuristic around the concept of *relative frequency of execution* we have been able to make an abstract visualization of the execution of a software system.

On a macro-level scale our visualization allows to identify parts in a trace where the same — or similar — functionality is executed. As an example we have drawn a simple class hierarchy in Fujaba — one of our case studies — that consists out of 8 classes. The resulting heartbeat visualization clearly contains 8 valleys at the points in time were these 8 classes are drawn. Through the knowledge that one of these valleys in the visualiza-

tion is conceptually linked with the execution of the particular functionality, the end-user can focus on studying the execution trace of only one of the applications of that functionality (instead of focussing on all 8).

On a micro-level scale on the other hand, we have been able to distinguish the traversal of a self-implemented linked list in the heartbeat visualization. The complete traversal of the linked list in our example requires around 10000 method exchanges, which, thanks to the visualization, can now be quickly skipped because of the high degree of similarity.

As such, both on a macro and on a micro scale, the visualization allows to discern the repetitive calling of specific functionality, thereby allowing the user to quickly go over these similar regions in the execution trace (or the resulting interaction diagram visualization).

With regard to scalability, the open source case studies we performed have shown that the technique is fairly scalable. In the case of the industrial case study however, where we needed to visualize 90 GB of trace data, we were unable to visualize the trace in its entirety. We did however recover the basic underlying mechanism to produce the frequency clusters visualization. This has allowed us to make a quick assessment of the industrial application's structure.

11.2 Opportunities for future research

Aspect based slicing We see a clear opportunity for future research in a concept that we call "aspect based slicing". Based on our research for identifying the important classes in a system, we want go one step further by also identifying the key collaborations among these important classes and the collaborations that these important classes have with other tightly-related classes.

To accomplish this, we are thinking of using aspect-orientation and more specifically the `cflow` pointcut, which would allow to obtain a very selective trace of all methods that belong to the important classes and their immediate collaborators.

Static analysis and hybrid approaches Another path that we want to pursue in the future is to try and improve the effectiveness of our current approaches, by also taking into account static information. This would lead to a hybrid approach, where the dynamic analysis results are augmented by static information.

Bunch As we have already indicated in Chapter 10, a thorough comparison of our approach and that of the Bunch clustering tool is also a viable research direction.

Part VI

Appendices

Appendix A

HITS webmining

A.1 Introduction

The HITS webmining algorithm we introduced in Chapter 5 is said to be convergent [Kleinberg, 1999]. This property of convergence implies that the algorithm will find a stable set of hub and authority nodes in a graph in a limited number of iterations. This appendix shows the proof of this convergence criterion, taken from Kleinberg [Kleinberg, 1999].

A.2 Setup and proof

Consider the following setting, taken directly from the domain of webmining.

Consider a collection V of hyperlinked pages as a directed graph $G = (V, E)$: the nodes correspond to the pages and a directed edge $(p, q) \in E$ indicates the presence of a link from p to q. Each page p is associated with a nonnegative *authority weight* $x^{<p>}$ and a nonnegative *hub weight* $y^{<p>}$. We view pages with larger x-values and y-values as being "better" authorities and hubs respectively.

We add an invariant that the weights of each type are normalized so their squares sum to 1:

$$\sum_{p \in S} (x^{<p>})^2 = 1 \quad ; \quad \sum_{p \in S} (y^{<p>})^2 = 1$$

The mutually reinforcing relationship between hubs and authorities is defined with the help of two operations on the weights, these operations are denoted by **J** and **O**. Given weights $\{ x^{<p>}\}$, $\{y^{<p>}\}$, the **J** operation updates the

x-weights as follows:

$$x^{<p>} \leftarrow \sum_{q:(q,p)\in E} y^{<q>} \tag{A.1}$$

The **O** operation then, which updates the y-values, is defined as follows:

$$y^{<p>} \leftarrow \sum_{q:(p,q)\in E} x^{<q>} \tag{A.2}$$

Now, to find the desired equilibrium values for the weights, one can apply the **J** and **O** operations in an alternating fashion, and see whether a fixed point is reached. Indeed, we can now state a version of our basic algorithm. We represent the set of weights $\{x^{<p>}\}$ as a vector x with a coordinate for each node in the graph G; analogously, we represent the set of weights $\{y^{<p>}\}$ as a vector y.

```
Iterate(G,k)
   G: a collection of n linked pages
   k: a natural number
   Let z denote the vector (1, 1, 1, ..., 1) ∈  R^n.
   Set x_0 := z.
   Set y_0 := z.
   For i = 1, 2, ..., k
      Apply the J operation to (x_{i-1}, y_{i-1}),
            obtaining new x-weights x'_i.
      Apply the O operation to (x'_i, y_{i-1}),
            obtaining new y-weights y'_i.
      Normalize x'_i, obtaining x_i.
      Normalize y'_i, obtaining y_i.
   End
   Return (x_k, y_k).
```

To address the issue of how best to choose k, the number of iterations, we first show that as one applies `Iterate` with arbitrarily large values of k, the sequences of vectors $\{x_k\}$ and $\{y_k\}$ converge to fixed points x^* and y^*.

Let M be a symmetric $n \times n$ matrix. An *eigenvalue* of M is a number λ with the property that, for some vector ω, we have $M\omega = \lambda\omega$. The set of all such ω is a subspace of $\mathbf{R}^n$, which we refer to as the *eigenspace* associated with λ; the dimension of this space will be referred to as the *multiplicity* of λ. It is a standard fact that M has at most n distinct eigenvalues, each of them a real number, and the sum of their multiplicities is exactly n. We will denote these eigenvalues by $\lambda_1(M)$, $\lambda_2(M)$, ..., $\lambda_n(M)$, indexed in order of decreasing absolute values, and with each eigenvalue listed a number of

times equal to its multiplicity. For each distinct eigenvalue, we choose an ortonormal basis of its eigenspace; considering the vectors in all these bases, we obtain a set of eigenvectors $\omega_1(M), \omega_2(M), \ldots, \omega_n(M)$ that we can index in such a way that $\omega_i(M)$ belongs to the eigenspace of $\lambda_i(M)$.

For the sake of simplicity, we will make the following technical assumption about all the matrices we deal with:

$$|\lambda_1(M)| > |\lambda_2(M)| \tag{A.3}$$

When this assumption holds, we refer to $\omega_1(M)$ as the *principal eigenvector*, and all other $\omega_i(M)$ as *nonprincipal eigenvectors*. When the assumption does not hold, the analysis becomes less clean, but it is not affected in any substantial way.

We now prove that the `Iterate` procedure converges as k increases arbitrarily.

Theorem A.2.1. *The sequences x_1, x_2, x_3,... and y_1, y_2, y_3,... converge (to limits x^* and y^*, respectively).*

Proof. Let $G = (V, E)$, with $V = \{p_1, p_2, \ldots, p_n\}$, and let A denote the *adjecency matrix* of the graph G; the (i, j)th entry of A is equal to 1 if (p_i, p_j) is an edge of G, and is equal to 0, otherwise. One easily verifies that the **J** and **O** operations can be written $x \leftarrow A^T y$ and $y \leftarrow Ax$, respectively. Thus, x_k is the unit vector in the direction of $A^T(AA^T)^{k-1}z$, and y_k is the unit vector in the direction of $(AA^T)^k z$.

Now, a standard result of linear algebra (see Kleinberg [Kleinberg, 1999]) states that if M is a symmetric $n \times n$ matrix, and v is a vector not orthogonal to the principal eigenvector $\omega_1(M)$, then the unit vector in the direction of $M^K v$ converges to $\omega_1(M)$ as k increases without bound. Also (as a corollary), if M has only nonnegative entries, then the principal eigenvector of M has only nonnegative entries.

Consequently, z is not orthogonal to $\omega_1(AA^T)$, and hence the sequence $\{y_k\}$ converges to a limit y^*. Similarly, one can show that if $\lambda_1(A^T A) \neq 0$ (as dictated by the assumption A.3), then $A^T z$ is not orthogonal to $\omega_1(A^T A)$. It follows that the sequence $\{x_k\}$ converges to a limit x^*. □

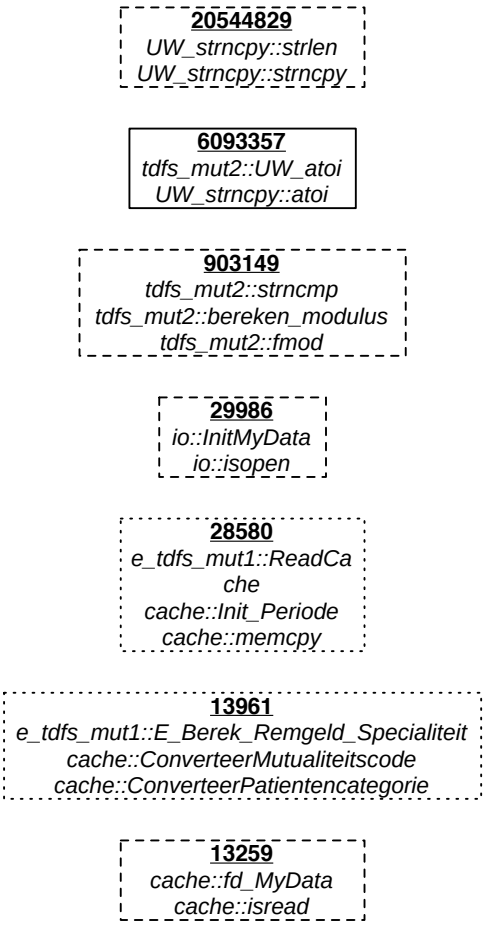
The proof of Theorem A.2.1 yields the following additional result (in the above notation).

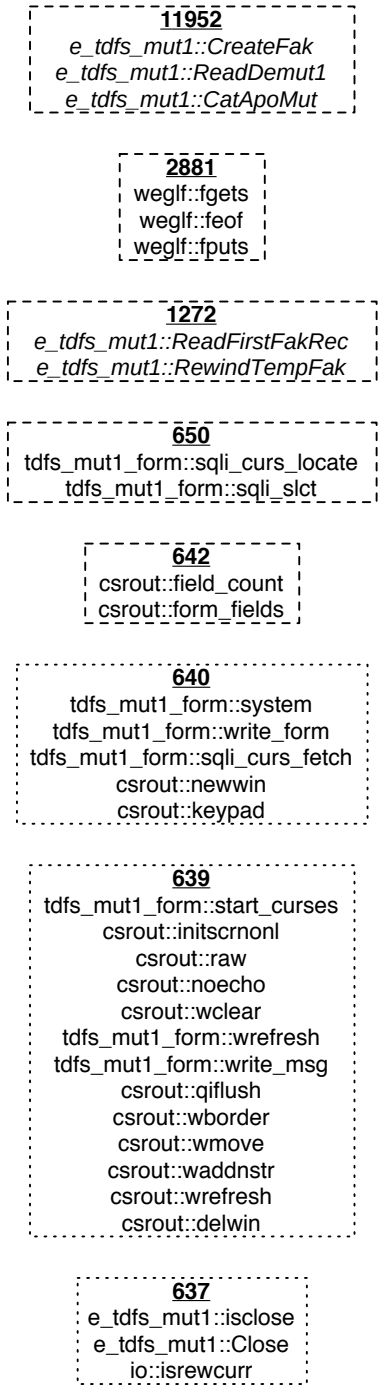
Theorem A.2.2. *(SUBJECT TO ASSUMPTION A.3). x^* is the principal eigenvector of $A^T A$, and y^* is the principal eigenvector of AA^T.*

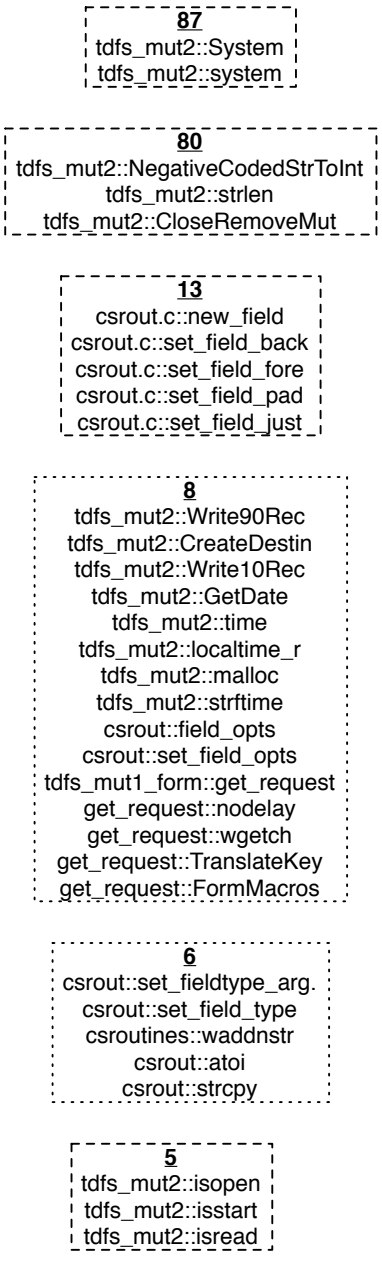
In our experiments, we find that the convergence of `Iterate` is quite rapid; one essentially always finds that $k = 20$ is sufficient for the c largest coordinates in each vector to become stable, for values of c in the range that we use. Of course, Theorem A.2.2 shows that one can use any eigenvector algorithm to compute the fixed points x^* and y^*; we have stuck to the above exposition in terms of the *Iterate* procedure for two reasons. First, it emphasizes the underlying motivation for our approach in terms of the reinforcing **J** and **O** operations. Second, one does not have to run the above process of iterated **J**/**O** operations to convergence; one can compute weights $\{x^{<p>}\}$ and $\{y^{<p>}\}$ by starting from any initial vectors x_0 and y_0, and performing a fixed bounded number of **J** and **O** operations.

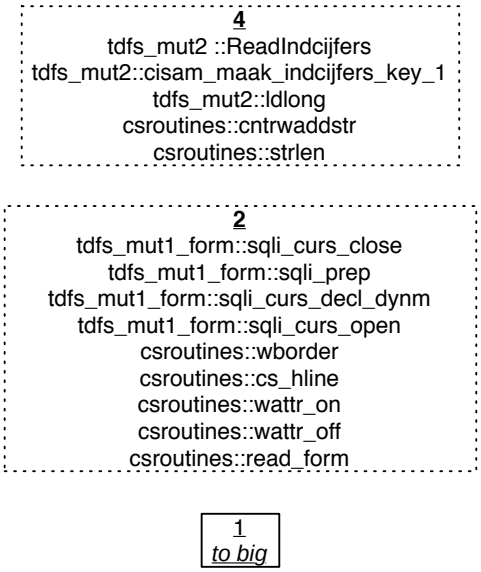
Appendix B

Frequency analysis results for TDFS




11952
e_tdfs_mut1::CreateFak
e_tdfs_mut1::ReadDemut1
e_tdfs_mut1::CatApoMut
2881
weglf::fgets
weglf::feof
weglf::fputs
1272
e_tdfs_mut1::ReadFirstFakRec
e_tdfs_mut1::RewindTempFak
650
tdfs_mut1_form::sqli_curs_locate
tdfs_mut1_form::sqli_slct
642
csrout::field_count
csrout::form_fields
640
tdfs_mut1_form::system
tdfs_mut1_form::write_form
tdfs_mut1_form::sqli_curs_fetch
csrout::newwin
csrout::keypad
639
tdfs_mut1_form::start_curses
csrout::initscrnonl
csrout::raw
csrout::noecho
csrout::wclear
tdfs_mut1_form::wrefresh
tdfs_mut1_form::write_msg
csrout::qiflush
csrout::wborder
csrout::wmove
csrout::waddnstr
csrout::wrefresh
csrout::delwin
637
e_tdfs_mut1::isclose
e_tdfs_mut1::Close
io::isrewcurr


87
tdfs_mut2::System
tdfs_mut2::system
80
tdfs_mut2::NegativeCodedStrToInt
tdfs_mut2::strlen
tdfs_mut2::CloseRemoveMut
13
csrout.c::new_field
csrout.c::set_field_back
csrout.c::set_field_fore
csrout.c::set_field_pad
csrout.c::set_field_just
8
tdfs_mut2::Write90Rec
tdfs_mut2::CreateDestin
tdfs_mut2::Write10Rec
tdfs_mut2::GetDate
tdfs_mut2::time
tdfs_mut2::localtime_r
tdfs_mut2::malloc
tdfs_mut2::strftime
csrout::field_opts
csrout::set_field_opts
tdfs_mut1_form::get_request
get_request::nodelay
get_request::wgetch
get_request::TranslateKey
get_request::FormMacros
6
csrout::set_fieldtype_arg.
csrout::set_field_type
csroutines::waddnstr
csrout::atoi
csrout::strcpy
5
tdfs_mut2::isopen
tdfs_mut2::isstart
tdfs_mut2::isread


4
tdfs_mut2 ::ReadIndcijfers
tdfs_mut2::cisam_maak_indcijfers_key_1
tdfs_mut2::ldlong
csroutines::cntrwaddstr
csroutines::strlen
2
tdfs_mut1_form::sqli_curs_close
tdfs_mut1_form::sqli_prep
tdfs_mut1_form::sqli_curs_decl_dynm
tdfs_mut1_form::sqli_curs_open
csroutines::wborder
csroutines::cs_hline
csroutines::wattr_on
csroutines::wattr_off
csroutines::read_form
1
to big

Bibliography

[Akers, 2005] Akers, R. L. (2005). Using build process intervention to accommodate dynamic instrumentation of complex systems. In *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05)*. Technical Report 2005-12, Department of Mathematics & Computer Science, University of Antwerp.

[Andrews, 1998] Andrews, J. (1998). Testing using log file analysis: tools, methods, and issues. In *Proceedings of the 13th International Conference on Automated Software Engineering (ASE'98)*, page 157. IEEE Computer Society.

[Arisholm et al., 2004] Arisholm, E., Briand, L., and Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506.

[Ball, 1999] Ball, T. (1999). The concept of dynamic analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234. Springer-Verlag.

[Bennett, 1995] Bennett, K. (1995). Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23.

[Biggerstaff et al., 1993] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering (ICSE '93)*, pages 482–498. IEEE Computer Society.

[Brant et al., 1998] Brant, J., Foote, B., Johnson, R. E., and Roberts, D. (1998). Wrappers to the rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag.

[Briand et al., 1999] Briand, L. C., Daly, J. W., and Wüst, J. K. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121.

[Brin and Page, 1998] Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117.

[Brodie and Stonebraker, 1995] Brodie, M. and Stonebraker, M. (1995). *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

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

[Corbi, 1990] Corbi, T. A. (1990). Program understanding: Challenge for the 90s. *IBM Systems Journal*, 28(2):294–306.

[de Oca and Carver, 1998] de Oca, C. M. and Carver, D. L. (1998). Identification of data cohesive subsystems using data mining techniques. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, pages 16–23. IEEE Computer Society.

[De Pauw et al., 2001] De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., and Yang, J. (2001). Visualizing the execution of java programs. In Diehl, S., editor, *Software Visualization: International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001*, volume 2269 / 2002 of *Lecture Notes in Computer Science*, page 151. Springer.

[De Pauw et al., 1998] De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M. (1998). Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*.

[Demeyer et al., 2003] Demeyer, S., Ducasse, S., and Nierstrasz, O. (2003). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

[Denti et al., 2001] Denti, E., Omicini, A., and Ricci, A. (2001). tuProlog: A light-weight Prolog for Internet applications and infrastructures. In

Practical Aspects of Declarative Languages, volume 1990 of *LNCS*, pages 184–198. Springer-Verlag.

[Ducasse et al., 2004] Ducasse, S., Lanza, M., and Bertuli, R. (2004). High-level polymetric views of condensed run-time information. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR2004)*, pages 309–318. IEEE Computer Society.

[Ducasse et al., 1999] Ducasse, S., Rieger, M., and Demeyer, S. (1999). A language independent approach for detecting duplicated code. In Yang, H. and White, L., editors, *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society.

[Eisenbarth et al., 2001] Eisenbarth, T., Koschke, R., and Simon, D. (2001). Aiding program comprehension by static and dynamic feature analysis. In *17th International Conference on Software Maintenance (ICSM'01)*, pages 602–611. IEEE Computer Society.

[El-Ramly et al., 2002] El-Ramly, M., Stroulia, E., and Sorenson, P. (2002). From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 315–324. ACM Press.

[Fraley and Raftery, 1998] Fraley, C. and Raftery, A. E. (1998). How many clusters? which clustering method? answers via model-based cluster analysis. *The Computer Journal*, 41(8):578–588.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley.

[Gargiulo and Mancoridis, 2001] Gargiulo, J. and Mancoridis, S. (2001). Gadget: A tool for extracting the dynamic structure of java programs. In *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'01)*, pages 244–251.

[Gibson et al., 1998] Gibson, D., Kleinberg, J. M., and Raghavan, P. (1998). Inferring web communities from link topology. In *UK Conference on Hypertext*, pages 225–234.

[Gold et al., 2004] Gold, N., Knight, C., Mohan, A., and Munro, M. (2004). Understanding service-oriented software. *IEEE Software*, 21(2):71–77.

[Greevy and Ducasse, 2005] Greevy, O. and Ducasse, S. (2005). Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 314–323. IEEE Computer Society.

[Gschwind et al., 2003] Gschwind, T., Oberleitner, J., and Pinzger, M. (2003). Using run-time data for program comprehension. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 245–250. IEEE Computer Society.

[Hamou-Lhadj, 2005a] Hamou-Lhadj, A. (2005a). The concept of trace summarization. In *Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis*, pages 43–47. Technical Report 2005-12, Department of Mathematics & Computer Science, University of Antwerp.

[Hamou-Lhadj, 2005b] Hamou-Lhadj, A. (2005b). *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension.* PhD thesis, University of Ottawa, Canada.

[Hamou-Lhadj et al., 2005] Hamou-Lhadj, A., Braun, E., Amyot, D., and Lethbridge, T. (2005). Recovering behavioral design models from execution traces. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 112–121. IEEE Computer Society.

[Hamou-Lhadj and Lethbridge, 2004] Hamou-Lhadj, A. and Lethbridge, T. (2004). A metamodel for dynamic information generated from object-oriented systems. *Electr. Notes Theor. Comput. Sci.*, 94:59–69.

[Hamou-Lhadj and Lethbridge, 2006] Hamou-Lhadj, A. and Lethbridge, T. (2006). Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC'06)*, pages 181–190. IEEE Computer Society.

[Hamou-Lhadj et al., 2004] Hamou-Lhadj, A., Lethbridge, T. C., and Fu, L. (2004). Challenges and requirements for an effective trace exploration tool. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC'04)*, pages 70–78. IEEE Computer Society.

[Jacobson, 1995] Jacobson, I. (1995). *Object-Oriented Software Engineering: a Use Case driven Approach.* Addison–Wesley.

[Jahnke and Walenstein, 2000] Jahnke, J. H. and Walenstein, A. (2000). Reverse engineering tools as media for imperfect knowledge. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 22–31. IEEE Computer Society.

[Jerding and Rugaber, 1997] Jerding, D. and Rugaber, S. (1997). Using visualization for architectural localization and extraction. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE'04)*, page 56. IEEE Computer Society.

[Jerding and Stasko, 1998] Jerding, D. and Stasko, J. T. (1998). The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):257–271.

[Jerding et al., 1997] Jerding, D. F., Stasko, J. T., and Ball, T. (1997). Visualizing interactions in program executions. In *Proceedings of the 19th international conference on Software Engineering (ICSE'97)*, pages 360–370, New York, NY, USA. ACM Press.

[Kaufman and Rousseeuw, 1990] Kaufman, L. and Rousseeuw, P. (1990). *Finding groups in data*. Wiley-Interscience.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag.

[Kleinberg, 1999] Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632.

[Lakhotia, 1993] Lakhotia, A. (1993). Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software*, 23(3):269–275.

[Lämmel and De Schutter, 2005] Lämmel, R. and De Schutter, K. (2005). What does Aspect-Oriented Programming mean to Cobol? In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 99–110, New York, NY, USA. ACM Press.

[Lanza, 2003] Lanza, M. (2003). *Object-Oriented Reverse Engineering — Coarse-grained, Fine-Grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne.

[Larus, 1993] Larus, J. R. (1993). Efficient program tracing. *IEEE Computer*, 26(5):52–61.

[Lehman, 1998] Lehman, M. (1998). Software's future: Managing evolution. *IEEE Software*, 15(1):40–44.

[Lehman and Belady, 1985] Lehman, M. and Belady, L. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA.

[Lethbridge and Anquetil, 1998] Lethbridge, T. C. and Anquetil, N. (1998). Experiments with coupling and cohesion metrics in a large system. Working paper, School of Information Technology and Engineering, also see http://www.site.uottawa.ca/ tcl/papers/ metrics/ExpWithCouplingCohesion.html.

[Linthicum, 1999] Linthicum, D. S. (1999). *Enterprise Application Integration*. Addison-Wesley.

[Lukoit et al., 2000] Lukoit, K., Wilde, N., Stoweel, S., and Hennessey, T. (2000). Tracegraph: Immediate visual location of software features. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM'00)*, pages 33–39. IEEE Computer Society.

[Mancoridis et al., 1999] Mancoridis, S., Mitchell, B. S., Chen, Y.-F., and Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, page 50. IEEE Computer Society.

[Mens, 2000] Mens, K. (2000). *Automating architectural conformance checking by means of logic meta programming*. PhD thesis, Vrije Universiteit Brussel.

[Mock, 2003] Mock, M. (2003). Dynamic analysis from the bottom up. In *ICSE 2003 Workshop on Dynamic Analysis (WODA'03)*.

[Moise and Wong, 2003] Moise, D. L. and Wong, K. (2003). An industrial experience in reverse engineering. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 275–284. IEEE Computer Society.

[Pennington, 1987] Pennington, N. (1987). Stimulus structures and mental prepresentations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341.

[Reiss and Renieris, 2001] Reiss, S. P. and Renieris, M. (2001). Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE01)*, pages 221–230. IEEE Computer Society.

[Renieris and Reiss, 1999] Renieris, M. and Reiss, S. P. (1999). ALMOST: Exploring program traces. In *Proc. 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77. http://citeseer.nj.nec.com/renieris99almost.html.

[Richner, 2002] Richner, T. (2002). *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Berne.

[Richner and Ducasse, 1999] Richner, T. and Ducasse, S. (1999). Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, pages 13–22. IEEE Computer Society.

[Richner and Ducasse, 2002] Richner, T. and Ducasse, S. (2002). Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 34–43. IEEE Computer Society.

[Robillard, 2005] Robillard, M. P. (2005). Automatic generation of suggestions for program investigation. *SIGSOFT Software Engineering Notes*, 30(5):11–20.

[Robillard et al., 2004] Robillard, M. P., Coelho, W., and Murphy, G. C. (2004). How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903.

[Sayyad-Shirabad et al., 1997] Sayyad-Shirabad, J., Lethbridge, T. C., and Lyon, S. (1997). A little knowledge can go a long way towards program understanding. In *Proceedings of the 5th International Workshop on Program Comprehension (IWPC'97)*, pages 111–117. IEEE Computer Society.

[Selby and Basili, 1991] Selby, R. W. and Basili, V. R. (1991). Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152.

[Smith and Korel, 2000] Smith, R. and Korel, B. (2000). Slicing event traces of large software systems. In *Automated and Algorithmic Debugging*.

[Sneed, 1996] Sneed, H. (1996). Encapsulating legacy software for use in client/server systems. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, pages 104–119. IEEE Computer Society.

[Sneed, 2004] Sneed, H. (2004). Program comprehension for the purpose of testing. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC'04)*, pages 162–171. IEEE Computer Society.

[Sneed, 2005] Sneed, H. (2005). An incremental approach to system replacement and integration. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 196–206. IEEE Computer Society.

[Spinellis, 2003] Spinellis, D. (2003). *Code Reading: The Open Source Perspective*. Addison-Wesley.

[Stevens et al., 1974] Stevens, W., Meyers, G., and Constantine, L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.

[Storey et al., 2000] Storey, M.-A. D., Wong, K., and Müller, H. A. (2000). How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207.

[Systä, 1999] Systä, T. (1999). On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99)*, pages 304–313. IEEE Computer Society.

[Systä, 2000a] Systä, T. (2000a). *Static and Dynamic Reverse Engineering Techniques for Java Software Systemds*. PhD thesis, University of Tampere.

[Systä, 2000b] Systä, T. (2000b). Understanding the behavior of java programs. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 214–223. IEEE Computer Society.

[Tahvildari, 2003] Tahvildari, L. (2003). *Quality-Drive Object-Oriented Reengineering Framework*. PhD thesis, Department of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada.

[Tilley et al., 2005] Tilley, T., Cole, R., Becker, P., and Eklund, P. W. (2005). A survey of formal concept analysis support for software engineering activities. In Stumme, G., editor, *Formal Concept Analysis*, volume 3626 of *LNCS*, pages 250–271. Springer.

[von Mayrhauser and Vans, 1994] von Mayrhauser, A. and Vans, A. M. (1994). Comprehension processes during large scale maintenance. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 39–48, Los Alamitos, CA, USA. IEEE Computer Society.

[von Mayrhauser and Vans, 1995] von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55.

[Walker et al., 1998] Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J. (1998). Visualizing dynamic software system information through high-level models. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, volume 33 of *ACM SIGPLAN Notices*, pages 271–238. ACM.

[Walker et al., 2000] Walker, R. J., Murphy, G. C., Steinbok, J., and Robillard, M. P. (2000). Efficient mapping of software system traces to architectural views. In *Proceedings of CASCON*, number TR-2000-09, pages 31–40. http://citeseer.nj.nec.com/walker00efficient.html.

[Wand and Weber, 1990] Wand, Y. and Weber, R. (1990). An ontological model of an information system. *IEEE Transactions on Software Engineering*, 16(11):1282–1292.

[Wilde, 1994] Wilde, N. (1994). Faster reuse and maintenance using software reconnaissance. Technical report, Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL.

[Wong et al., 1995] Wong, K., Tilley, S. R., Müller, H. A., and Storey, M.-A. D. (1995). Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54.

[Yang et al., 2005] Yang, H. Y., Tempero, E., and Berrigan, R. (2005). Detecting indirect coupling. In *Proceedings of the Australian Software Engineering Conference (ASWEC'05)*, pages 212–221. IEEE Computer Society.

[Yourdon and Constantine, 1979] Yourdon, E. and Constantine, L. L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice Hall.

[Zaidman et al., 2006a] Zaidman, A., Adams, B., De Schutter, K., Demeyer, S., Hoffman, G., and De Ruyck, B. (2006a). Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 89–98. IEEE Computer Society.

[Zaidman et al., 2005] Zaidman, A., Calders, T., Demeyer, S., and Paredaens, J. (2005). Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 134–142. IEEE Computer Society.

[Zaidman and Demeyer, 2004] Zaidman, A. and Demeyer, S. (2004). Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 329–338. IEEE Computer Society.

[Zaidman et al., 2006b] Zaidman, A., Du Bois, B., and Demeyer, S. (2006b). How webmining and coupling metrics can improve early program comprehension. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC'06)*, pages 74–78. IEEE Computer Society.

[Zayour and Lethbridge, 2001] Zayour, I. and Lethbridge, T. C. (2001). Adoption of reverse engineering tools: a cognitive perspective and methodology. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 245–255. IEEE Computer Society.

Publications

Conference publications (listed chronologically)

- Andy Zaidman and Serge Demeyer.
  Managing trace data volume through a heuristical clustering process based on event execution frequency
  Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR2004), pages 329-338, IEEE Computer Society, 2004
- Andy Zaidman, Toon Calders, Serge Demeyer and Jan Paredaens.
  Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process
  Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR2005), pages 134-142, IEEE Computer Society, 2005
- Orla Greevy, Abdelwahab Hamou-Lhadj and Andy Zaidman.
  Workshop on Program Comprehension through Dynamic Analysis
  Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005), pages 232-232, IEEE Computer Society, 2005
- Andy Zaidman.
  Scalability Solutions for Program Comprehension through Dynamic Analysis
  Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR2006), pages 327-330, IEEE Computer Society, 2006
- Andy Zaidman, Bram Adams, Kris De Schutter, Serge Demeyer, Ghislain Hoffman and Bernard De Ruyck. *Regaining Lost Knowledge through Dynamic Analysis and Aspect Orientation - An Industrial Experience Report*
  Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR2006), pages 91-102, IEEE Computer Society, 2006
- Andy Zaidman, Bart Du Bois and Serge Demeyer. *How Webmining and Coupling Metrics Can Improve Early Program Compehension*
  Proceedings of the 14th International Conference on Program Comprehension (ICPC2006), pages 74-78, IEEE Computer Society, 2006
- Andy Zaidman, Orla Greevy and Abdelwahab Hamou-Lhadj.

Workshop on Program Comprehension through Dynamic Analysis
Accepted for publication in the proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), IEEE Computer Society, 2006

Currently submitted work

- Bram Adams, Kris De Schutter, Andy Zaidman, Serge Demeyer and Herman Tromp.
Aspect-Enabled Dynamic Analyses for Reverse Engineering Legacy Environments – An Industrial Experience Report
Submitted to a special CSMR issue of the Journal of Systems and Software (JSS) by Elsevier, as an extension to the CSMR 2006 paper.