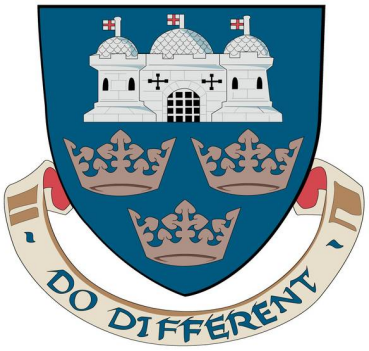
Enhancing Legacy Software System Analysis by Combining Behavioural and Semantic Information Sources



David Cutting

School of Computing Sciences

University of East Anglia

A thesis submitted for the degree of

Doctor of Philosophy

November 2016

To my parents, Peter and Valerie,
who brought me into this world,
and have tirelessly supported me.

Maybe this makes up for it all?
At least in some small part?
Even a teeny bit?

Probably not.

Thanks for the inspiration

Acknowledgements

First I must acknowledge and deeply thank my supervisor, Dr. Joost Noppen, for not only giving me a chance at this project but tolerating with good cheer my countless mistakes, poor drafts, and general laziness. Also my great gratitude goes to my second supervisor, Dr. Anthony Bagnall, for putting up with me. Any errors contained within are entirely mine.

For their "support" (usually in the form of abuse) in no particular order I heap praise upon:

1) The fellow *dishonourable members of the illustrious bigtime super funclub of the UEA*: Dani and Sam.

2) My fellow 2.02 lab members: Alex, Andrei, Awat, Bogdan, Claudia, James, Matt, and Sarah.

3) Those *funny funny people* who have also survived the Nam (*you weren't there man*) and continue to speak to me: Celia, Erwin, Joanna, Marika (*squeak with joy!*), Marjolijn, Matt, and Siew Wee.

4) The other members of the Software Engineering Group: Adam, Steve, and Sultan.

5) My partner in dubious consultancy practice Zelim.

6) My *Brothers in Stavros* of the NASRA: Amy, Justin, Lily, Matt, Mike, and Theo.

Thank you to all the school staff who have helped me, with teaching and admin as well as research, specifically Dan, Gavin, Geoff, Graeme, Pam, Rudy, and Wenjia. Special thanks also to Binoop, Heidi, Matt, and Russ in the support team.

I gratefully acknowledge and thank the Ecole Polytechnique de Montreal for providing open access to their IBDOOS tool.

Portions of the research presented in this thesis were carried out on the High Performance Computing Cluster supported by the Research and Specialist Computing Support service at the University of East Anglia.

"Ford!" he said, "there's an infinite number of monkeys outside who want to talk to us about this script for Hamlet they've worked out."

– Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Abstract

Computer software is, by its very nature highly complex and invisible yet subject to a near-continual pressure to change. Over time the development process has become more mature and less risky. This is in large part due to the concept of software traceability; the ability to relate software components back to their initial requirements and between each other. Such traceability aids tasks such as maintenance by facilitating the prediction of "ripple effects" that may result, and aiding comprehension of software structures in general. Many organisations, however, have large amounts of software for which little or no documentation exists; the original developers are no longer available and yet this software still underpins critical functions. Such "legacy software" can therefore represent a high risk when changes are required.

Consequently, large amounts of effort go into attempting to comprehend and understand legacy software. The most common way to accomplish this, given that reading the code directly is hugely time consuming and near-impossible, is to reverse engineer the code, usually to a form of representative projection such as a UML class diagram. Although a wide number of tools and approaches exist, there is no empirical way to compare them or validate new developments. Consequently there was an identified need to define and create the Reverse Engineering to Design Benchmark (RED-BM). This was then applied to a number of industrial tools. The measured performance of these tools varies from 8.8% to 100%, demonstrating both the effectiveness of the benchmark and the questionable performance of several tools.

In addition to the structural relationships detectable through static reverse engineering, other sources of information are available with the potential to reveal other types of relationships such as semantic links. One such source is the mining of source code repositories which can be analysed to find components within a software system that have, historically, commonly been changed together during

the evolution of the system and from the strength of that infer a semantic link. An approach was implemented to mine such semantic relationships from repositories and relationships were found beyond those expressed by static reverse engineering. These included groups of relationships potentially suitable for clustering.

To allow for the general use of multiple information sources to build traceability links between software components a uniform approach was defined and illustrated. This includes rules and formulas to allow combination of sources. The uniform approach was implemented in the field of predictive change impact analysis using reverse engineering and repository mining as information sources. This implementation, the Java Code Relationship Anlaysis (jcRA) package, was then evaluated against an industry standard tool, JRipples. Depending on the target, the combined approach is able to outperform JRipples in detecting potential impacts with the risk of over-matching (a high number of false-positives and overall class coverage on some targets).

Contents

List of Papers

Published

Cutting, D. and Noppen, J. (2014). Working with reverse engineering output for benchmarking and further use. In *Proceedings of the 9th International Conference on Software Engineering Advances*. IARIA. [Cutting and Noppen, 2014]

Cutting, D. and Noppen, J. (2015). An extensible benchmark and tooling for comparing reverse engineering approaches. *International Journal in Advances in Software*, 8(1&2):115-124. [Cutting and Noppen, 2015]

In Progress

Cutting, D. and Noppen, J. "Identifying semantic links beyond reverse engineering by combining repository mining". Target: *IEEE Transactions on Software Engineering*

List of Figures

List of Tables

Listings

Glossary

jcRA Java Code Relation Analysis (jcRelationAnalysis) is a tool created to analyse and explore the relationship between components within Java code

MOF Meta-Object Format, a meta language to describe objects and their relations to each other (the basis of XMI)

MDS Multi-Dimensional Scaling, the process of taking data described in one set of dimensions and projecting it as a best fit into a defined differently-dimensioned space

OMG Object Management Group, a body to set standards for object-oriented modeling

OMT Object Modelling Technique, a technique used to turn descriptions and other elements into software components

SEG The Software Engineering Group at UEA

UEA The University of East Anglia

UML Unified Modelling Language, a set of standards by which processes and software can be modeled and exchanged

XMI XMI Metadata Interchange, a format by OMG for the exchange of structural information about software

XML eXtensible Markup Language, a standard language for the representation of data

Chapter 1

Introduction

1.1 Introduction

Computer software is simultaneously highly complex, invisible, and under continual pressure to change [Brooks, 1987]. Previously the entire development process was fraught with risk. Developers wrote software with little or no reference back to the original requirements, and without this link not only was effort wasted but the majority of software projects failed in large part or completely [Naur and Randell, 1969]. In a survey of *successful* projects Naur and Randell [1969] found a common trait was that the requirements *were* referred back to during the development phase, and that code artefacts were traceable back to the requirements that necessitated them, thus forming the idea of software traceability.

Software traceability is the ability or potential to track links and relationships between artefacts within a software system. Such links may be between different levels of detail. For example they may be from requirements documentation through design to source code components and vice-versa, or detail relationships at the same level between software components such as dependencies [Cleland-Huang *et al.*, 2012; Gotel and Finkelstein, 1994]. Detailed up-to-date traceability within software has been shown to aid numerous aspects of software maintenance, including change impact analysis, bug detection and re-factoring. Clear traceability links also aid developers in quickly understanding the conceptual structure of software and their ability to make changes [Cleland-Huang *et al.*, 2012, 2014; Mäder and Egyed, 2015; Spanoudakis and Zisman, 2005].

With the growing amount of legacy software within organisations, and a general overall increase in size and complexity of such software, problems inherent to software comprehension are also growing. Early adopters of technology are often left with large legacy codebases which were not only developed prior to traceability being commonplace, but also for which original developers are no longer available, and so little detailed knowledge about the software exists within the organisation. As a result, large software archives exist without traceability links which is a challenge both in terms of maintenance and system comprehension.

The focus of this thesis therefore is to investigate and improve methods of traceability with specific regard to existing legacy software systems, be that vertical (forward and backward) traceability between design artefacts and code, or hor-

izontal dependency traceability identifying links between software components.

Ultimately, with a legacy system lacking documentation, the source code itself is the de facto standard of operation. Analysis of the code is the only way in which insight can be gained as a human simply reading the source code line by line and attempting to comprehend the overall function is next to impossible [Counsell *et al.*, 2004; Kumar *et al.*, 2015; Meyer, 2006; Philippow *et al.*, 2005]. This field is referred to as reverse engineering, starting with a working system and by analysing the components in different ways, reaching an understanding as to how it operates. The most common output of these processes is the regeneration of structural design diagrams, commonly Unified Markup Language (UML) class diagrams. These show structural components and their interrelations [Chikofsky *et al.*, 1990; Sarkar *et al.*, 2013]. Other forms of reverse engineering exist to determine specialist cases such as detection of design patterns or "bad smells" [1] within the code [Meyer, 2006; Shi and Olsson, 2006].

Although a maturing field, especially in terms of structural recovery, reverse engineering of code still has a number of challenges which often relate to the quality, standards, complexity, and size of the analysed code [Counsell *et al.*, 2004; Meyer, 2006; Pinzger and Antoniol, 2013; Sarkar *et al.*, 2013; Uchiyama *et al.*, 2011]. Furthermore, there are a wide number of different approaches and tools but no empirical way to compare their relative performance and accuracy in recovering structural design information. Such a gap also means that new techniques or approaches have no common validation set against which they can be tested and evaluated.

Consequently this led to the creation of the Reverse Engineering to Design Benchmark (RED-BM) [Cutting, 2013; Cutting and Noppen, 2015]. RED-BM contains a number of elements including a target set of software source code along with the gold standard answers, for what elements and relationships are contained within. The benchmark was applied against a number of industry-standard reverse engineering tools and their performance was measured. Additionally the output of tested tools was included in the benchmark for comparative use. Performance of reverse engineering tools, both free open-source and commercial, were

[1] "Bad smells" being common yet undesirable mistakes within a codebase that can be easily refactored away such as duplicated code [Fowler, 1997]

found to be widely varied ranging in accuracy from 8.82% to 100% [Cutting and Noppen, 2015].

A number of exchange tools were created as part of this process. This allowed, for example, the programmatic use of reverse engineering output from third-party tools, opening the potential for combinational or external use of this data [Cutting and Noppen, 2014].

Another potential source of information about a software system lacking documentation is through historic evolutionary data i.e. how the source code has changed over time. It is increasingly common for developers, especially teams of developers, to use source code repositories to coordinate development. Although not designed with such analysis is mind, repositories do record the changes made to source code, often with meta data such as who made the change and when, along with a textual "comment" about the change [Hassan, 2008]. These repositories are seen as a rich potential source of information about a software system and have become an increasing focus in many different fields from research into development practice, to bug prediction [Allamanis and Sutton, 2013; Kagdi *et al.*, 2007b; Ramadani and Wagner, 2016; Williams and Hollingsworth, 2005]. Using historical information in combination with reverse engineering in different areas e.g. to allow for "source-code aware mining" and increased accuracy is also possible [Kagdi and Maletic, 2006; Lehnert, 2011].

One approach to using repository mining to inform dependency (horizontal) traceability is the concept of co-committal; a high correlation between artefacts being changed at the same time may infer a semantic relationship between them [Dit *et al.*, 2014; Kagdi *et al.*, 2007a; Ying *et al.*, 2004]. For example if whenever A is changed, B has a change recorded at the same time, they may be linked. Previous work in this area indicates that there are links found using this technique that are not evident through static analysis [Bieman *et al.*, 2003] and that these component links may be possible to cluster into architectural structures [Beyer and Noack, 2005]. Use of co-commit information alongside static reverse engineering data was instigated as a novel technique for research in the completion of this thesis. This is also an area in which Dit *et al.* [2014] were separately working with a somewhat different focus and different implementation.

As a proof of concept, a framework was created to extract co-commit infor-

mation (what components were changed and committed at the same time) from historical repositories and present this in a format of relationship pairs suitable for use in a number of analysis techniques including clustering.

When dealing with multiple sources of information, a challenge becomes how to use these disparate forms in a compatible and comparable form, especially as part of a repeatable and consistent empirical framework. To facilitate such use, a formal definition of a generic framework was created, in which different information sources can be used together, at their highest common level of granularity, for analysis. Concepts such as "ground truth" (relationships sets known to be true), relative weighting, and significance filtering are introduced to model the real-world uncertainty of some information sources.

With continual pressure of change it is imperative that the impact any alterations may have on the wider operation of a software system are understood [Lehman and Belady, 1985]. Most commonly this is done as a form of dependency traceability. That is understanding which components are closely coupled or related to each other thus gauging the probability and impact one will have on another often through static source code reverse engineering [Lehnert, 2011]. An interesting possibility is that of increasing the accuracy of this form of predictive change impact analysis (PCIA) through the combination of multiple information sources. To this end the generic approach is implemented to determine coupling within a source code using two information sources; static reverse engineering augmented with semantic links mined from source code repository information.

1.1.1 Fundamental Challenge and Contributions

Is it possible to reconstruct richer traceability structures through the combinational use of multiple sources of information?

In order to answer this a number of contributions are made as follows:

- Reverse engineering is investigated and a benchmark created to allow for the measurement of performance between tools and approaches, highlighting weak performance and gaps in output (Chapter 3)
- Tools are created to allow the programmatic use of reverse engineering output for other purposes (Chapter 3)

- Source code repository mining as a potential source of semantic information about software is investigated with a method to analyse sets of commit data defined (Chapter 4)
- Networking and clustering techniques are applied to mined repository data to illustrate clear semantic links beyond reverse engineering (Chapter 4)
- A generalised formal approach for the combinational use of multiple information sources for a given software system is defined (Chapter 5)
- The approach is implemented in the domain of predictive change impact analysis using reverse engineering and repository data in combination as the *jcRelationAnalysis* (jcRA) tool, and evaluated and found to outperform an industry standard tool (Chapter 6)

Ultimately, it is concluded (Chapter 7) that it is indeed possible to reconstruct richer traceability from multiple sources and apply this successfully to a problem domain.

1.2 Thesis Structure

The thesis structure is shown in Figure 1.1.

A literature review is contained in Chapter 2 in which all the key topics relating to the research are investigated; software traceability, reverse engineering, source code repository mining, clustering, and change impact analysis.

In Chapter 3 a benchmark to evaluate, compare, and validate reverse engineering approaches is introduced. The benchmark is described and metrics are defined before being applied against industry standard tools.

Chapter 4 contains research into the mining of source code repositories and their potential use alongside static reverse engineering output to combine semantic relationships and make a more complete picture of component relationships.

In Chapter 5 a generic model is introduced and defined allowing for the generalised use of multiple information sources in a repeatable empirical way to find and quantify relationships between components.

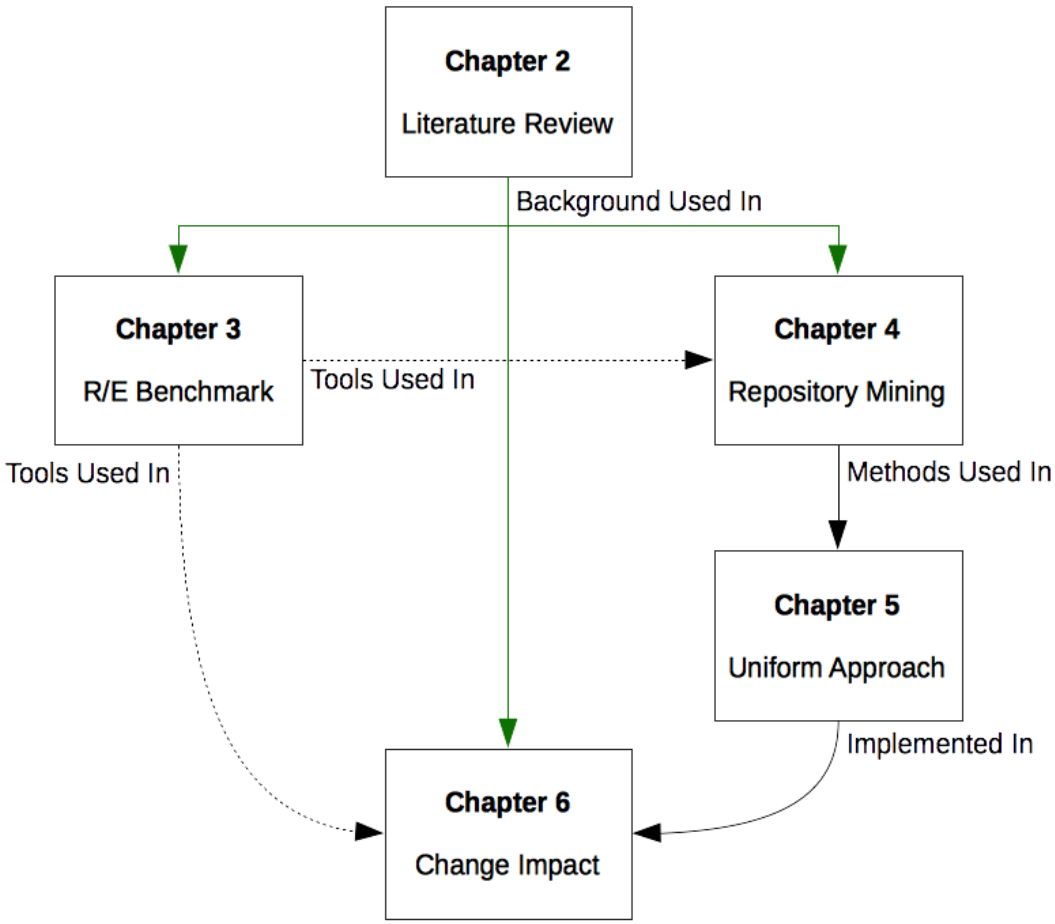


Figure 1.1: Thesis Structure

This general approach is then implemented in Chapter 6 to the problem domain of change impact analysis utilising two information sources (static analysis and repository mining) to determine impact sets for a number of real-world change scenarios. The implementation is in the form of the *jcRelationAnalysis* (jcRA) tool. This approach is then evaluated against an industry standard tool (JRipples). The evaluation finds that the new combined information approach outperforms JRipples in terms of detecting potential impacts with a risk of overmatching.

Finally, Chapter 7 reaches the overall conclusions of the research project and identifies potential future work to take the research further.

Chapter 2

Literature Review

2.1 Introduction to Literature Review

The literature review in this thesis contains an overview of the key literature related to the research work conducted on the themes identified in Chapter 1. Specifically there is a brief overview of traceability and requirements traceability (Section 2.2), reverse engineering of source code (Section 2.3) and mining of source code repository data (Section 2.4). Clustering techniques potentially offer the ability to determine varied relationships and links between software components as an aid to comprehension and are covered in Section 2.5. The domain of software change impact analysis is discussed in Section 2.6.

2.2 Software Traceability

Traceability in general terms is the ability or potential to track the semantics of links and relationships between components within a system [Asuncion *et al.*, 2007; Cleland-Huang *et al.*, 2014] or to "relate data that is stored within artefacts of some kind along with the ability to examine this relationship" [Cleland-Huang *et al.*, 2012]. With relation to software, traceability most commonly refers to *requirements traceability* (Section 2.2.1) in which high-level software requirements are linked to designs, and ultimately, the implementation of a software system e.g. linking manifestations of specific requirements through the complete project life cycle [Asuncion *et al.*, 2007; Edwards and Howell, 1991].

Traceability was initially recognised as an important factor in software engineering at the 1968 NATO conference which sought to address the "software crisis" which had become apparent, because of the majority of projects failing to deliver in part or completely. Naur and Randell [1969] analysed successful projects to find common traits and gave praise for projects in which "the system that they are designing contains explicit traces of the design process", e.g. there were clear links between levels of the system. By the mid-1970s traceability was listed by Boehm *et al.* [1976] as a topic of interest in software engineering to ensure quality. In the 1980s traceability was established as a requirement for a growing number of national and international standards often being mandated by large organisations such as the United States Department of Defence [Cleland-Huang

et al., 2014; Dorfman and Thayer, 1990]. Through the 1990s and 2000s numerous research projects and publications investigated problems with traditional [Gotel and Finkelstein, 1994; Ramesh and Edwards, 1993] and automated [Laurent *et al.*, 2007] or model-driven [Galvao and Goknil, 2007] approaches. In the current climate traceability is widely used in development projects, although in some cases the administration of traceability can become overburdening so providing little if any benefit [Asuncion *et al.*, 2007]. However the general focus and belief is that traceability can deliver significant benefits when used correctly both in terms of requirement delivery, software comprehension and maintenance impact analysis [Cleland-Huang *et al.*, 2012, 2014; Mäder and Egyed, 2015; Spanoudakis and Zisman, 2005].

Direction of Traceability

Traceability is the potential to map or link relationships between components [Asuncion *et al.*, 2007] and it has long been recognised that this mapping and linking can be focused or arranged in different directions [IEEE, 1984]. From the literature three concepts or "directions" of traceability can be defined [Asuncion *et al.*, 2007; Cleland-Huang *et al.*, 2012; Gotel and Finkelstein, 1994; IEEE, 1984; Jirapanthong, 2015]:

Forward traceability is the ability to identify sub-requirements or components from higher-level documentation.

Backwards traceability concerns linking more specific requirements or components with their higher-level source (or parent).

Dependency traceability is the linking of related and inter-dependent components, outside of a requirement tree, where one component depends on functionality within another.

One option to facilitate multiple directions of traceability (at least forward and backward between requirements and sub-requirements) is the use of a hierarchical numbering system. For example, where *1* represents a high-level requirement and *1.1*, *1.2* etc are sub-requirements of *1* [IEEE, 1984].

2.2.1 Requirements Traceability

Requirements traceability can be an important factor to support and assist various systems development processes. These can include analysis, change management, reuse and testing. It also allows for clearer acceptance by end-users showing clear links between the requirements and what has been delivered [Mäder and Egyed, 2015; Spanoudakis and Zisman, 2005]. Specifically "requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction, i.e. from its origin through to its development and specification to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases" [Gotel and Finkelstein, 1994]; tracking the semantics of links between requirements and system components at each level of the system [Harrington and Rondeau, 1993; Mäder and Egyed, 2015].

Requirements traceability aids development in a number of different ways, making clear links between requirements and lower-level steps such as designs, code and testing. At a later stage it can inform change planning and management, allowing for easier impact analysis, code verification and feature identification - tracing all these back even to initial requirements [Cleland-Huang *et al.*, 2012; Mäder and Egyed, 2015].

According to IEEE [1984] software requirements specifications, either user-generated or formed during a requirements analysis, direct the fundamental deliverables and form of the system. They are a common factor of many software development projects and when used well can be beneficial in a number of ways:

- providing a basis for agreement between developers and users on exactly what and how the system will perform;
- aid development reducing potential for omissions and re-development;
- act as a guide for costing;
- be used as the basis for testing and verification;
- enable easier transfer between developers or users; and
- serve as a base design for future enhancement.

[IEEE, 1984]

Traceability further serves development of these requirements by keeping and maintaining clear links between requirements and components within the system [Cleland-Huang *et al.*, 2012]. A well formed software requirement specification allows for traceability if "the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation" [IEEE, 1984].

2.3 Reverse Engineering

2.3.1 Introduction to Reverse Engineering

Reverse engineering is defined as "to disassemble and examine or analyse in detail (a product or device) to discover the concepts involved in manufacture" [Merriam-Webster, 2012]. According to Chikofsky *et al.* [1990] reverse engineering, specifically of software, involves the analysis of a system with the aim of:

- Identifying system components and relationships.
- Generating a view of the system at a higher level of abstraction or in an alternative form.

System development can be seen as *forward engineering*, moving from requirements and specification through design and implementation. *Reverse engineering*, by definition the opposite, is a move from an implemented system back to designs [Roscoe, 2011].

It can be sometimes claimed that software reverse engineering is a "disreputable activity", seeking to pull apart to use or copy others' solutions, potentially without permission. However, it is also commonly used by the intellectual property owner as an aid to further development or maintenance [Kumar *et al.*, 2015; Roscoe, 2011].

In an ideal world all systems would be developed in a clear and well documented manner, facilitating easy understanding of both structure and function.

However, developers are often faced with poorly documented and/or legacy systems which become extremely hard to comprehend by reading source code alone [Counsell *et al.*, 2004; Kumar *et al.*, 2015; Meyer, 2006; Philippow *et al.*, 2005].

Chikofsky *et al.* [1990], Roscoe [2011], Mamas and Kontogiannis [2000], Pinzger and Antoniol [2013], and Rasool and Streitfdert [2011] all agree that reverse engineering can have the following significant legitimate benefits with regard to poorly documented or legacy systems:

- Maintenance - in order to make even small corrective changes to the system, let alone major changes, a good understanding of its overall function, structure and inter-component reliance is required.
- Interoperability - allowing for the understanding of data flows and formats. This provides the ability to expand the system to interface with other systems for example through the transfer of data.
- Security - reverse engineering can be a very effective tool in testing the level of security of a system, discovering unforeseen security holes and other potential problems or unexpected behaviour.
- Behaviour Modification - an understanding of system operation is essential for modification or further expansion of the system.

Of these benefits aiding maintenance can be the most significant given that in cost terms maintenance is the most expensive stage of the software development lifecycle [Meyer, 2006].

In Section 2.3.2 the goals and challenges of software reverse engineering are identified. Approaches to reverse engineering are covered in Section 2.3.4 before different types of language abstracted notations are detailed in Section 2.3.3. Available tools for software reverse engineering are detailed in Section 2.3.5 before being compared with regard to features provided.

2.3.2 Goals of Software Reverse Engineering

The overarching purpose of software reverse engineering is to aid clarity and promote understanding of a system in order to facilitate maintenance and further

development [Chikofsky *et al.*, 1990; Pinzger and Antoniol, 2013]. Specifically the goal is therefore to identify components along with their relationships to facilitate the creation of understandable design-level representations of the system [Chikofsky *et al.*, 1990; Kumar *et al.*, 2015; Pinzger and Antoniol, 2013; Roscoe, 2011].

Chikofsky *et al.* [1990] defines three levels in which these representations can be seen:

- Requirements - the overall specification of the goals of the system, the problem it solves;
- Design - the specification of the overall system to meet the goal(s)/solve the problem(s) identified in requirements;
- Implementation - the actual code-level functionality.

Reverse engineering techniques may yield a wide variety of information such as the quantification of algorithms, identification of design patterns used or illustration of general component interactions [Roscoe, 2011]. However it remains the overall goal to produce design-level information in a level of detail and abstraction suitable for developers who are non-experts in a system to gain as complete an understanding as they require [Chikofsky *et al.*, 1990; Philippow *et al.*, 2005]. For example UML recovery (recreation of UML diagrams illustrating system design and operation) is a frequent aim in reverse engineering [Roscoe, 2011].

Challenges

There are a number of challenges with regard to reverse engineering from source code often owing to the inconsistent style and quality of code [Counsell *et al.*, 2004; Meyer, 2006; Uchiyama *et al.*, 2011] and success (the recreation of holistic systems designs) may vary widely depending on the methods used, the complexity, size and quality of the codebase [Counsell *et al.*, 2004; Sarkar *et al.*, 2013].

Commonly major programming languages (such as C++ and Java) have no support for in-built semantic definitions of structures such as design patterns. This leaves developers only the option of using code-level comments in natural language (possibly aided by comments in a set format compatible with javadoc

for example), something which is highly variable depending on the developers and may be entirely inconsistent between different teams [Philippow *et al.*, 2005].

Software systems often contain poor design and component relationships further complicating analysis and comprehension of the code with responsibilities being distributed throughout the system rather than compartmentalised [Meyer, 2006].

Even more formalised development approaches such as design patterns (Section 2.3.4), by definition a standardised way of implementing a solution, may have many different forms of concrete implementation or naming conventions [Uchiyama *et al.*, 2011].

2.3.3 Language Abstracted Notations

Language abstracted notations such as those detailed in this section are useful in reverse engineering to represent programmatic function in a manner which is independent of the original (concrete) code syntax. Code layout becomes irrelevant so control-flow and data-flow can be more easily determined (Fischer *et al.* [2007]; Section 2.3.1).

Abstract Syntax Tree (AST)

Abstract Syntax Trees (ASTs) have been commonly used in code compilation as an intermediate data format between a source-code language and a machine/platform specific binary output with programming language constructs represented as a tree [Fischer *et al.*, 2007; OMG *et al.*, 2011a]. In this way ASTs capture the key structure of code in tree form without unnecessary syntactic details; ASTs are a pure representation of function without detail contained in concrete trees such as language-specific flow controls [Jones, 2003]. ASTs are "a formal representation of the syntactical structure of the software that is more amenable to formal analysis techniques than is the concrete or surface syntax of software" [OMG *et al.*, 2011a].

ASTs are constructed most commonly through source-code parsing but may also be built from generation or derivation processes from another form [OMG *et al.*, 2011a].

An AST is a finite tree comprised of labelled and directed nodes with internal nodes representing operators and leaf nodes operands of the operators. Data structures that make up ASTs provide for a full collection of base compositional elements that are used by the source-code language [OMG *et al.*, 2011a].

While there are different specific implementations of AST syntax they all have the same design basis and ethos, representing a hierarchical tree of control starting from the top and executing in sequence with nodes indicating individual operations such as assignment and comparison [Fischer *et al.*, 2007]. An example of an AST can be seen in figure 2.1.

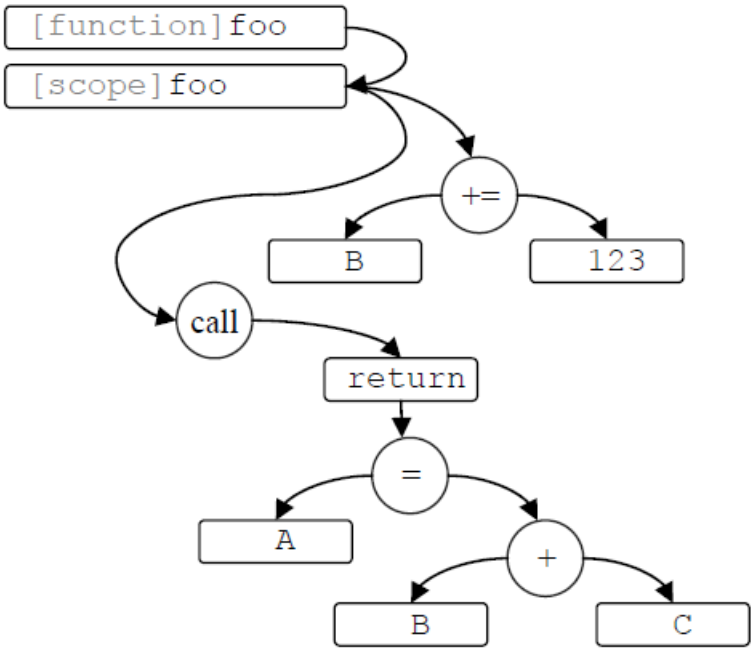


Figure 2.1: AST Example [Misek and Zavoral, 2010]

The Object Management Group (OMG) have been attempting to derive a more formalised and widely accepted metamodel for ASTs, allowing for wider standardisation and use in the form of the AST MetaModel (ASTM) [OMG *et al.*, 2011a].

OMG *et al.* [2011a] defines metamodels for three specific domains:

- Generic elements which are common in many languages represented by Generic Abstract Syntax Trees (GAST)

- Language specific where individual languages purely are modelled by Language Specific Abstract Syntax Trees (SAST)
- Proprietary when elements of languages are modelled in inconsistent formats by Proprietary Abstract Syntax Trees (PAST)

Abstract Semantic Graph (ASG)

An Abstract Semantic Graph (ASG) is an AST containing extra information providing a richer picture. In addition to the nodes contained within an AST representing source-code entities, ASGs contain *non-tree edges* showing relationships. These relationships connect references to declarations as well as declarations to types [Mamas and Kontogiannis, 2000; Raghavan *et al.*, 2004]. Figure 2.2 shows a simple ASG including edge information from Raghavan *et al.* [2004].

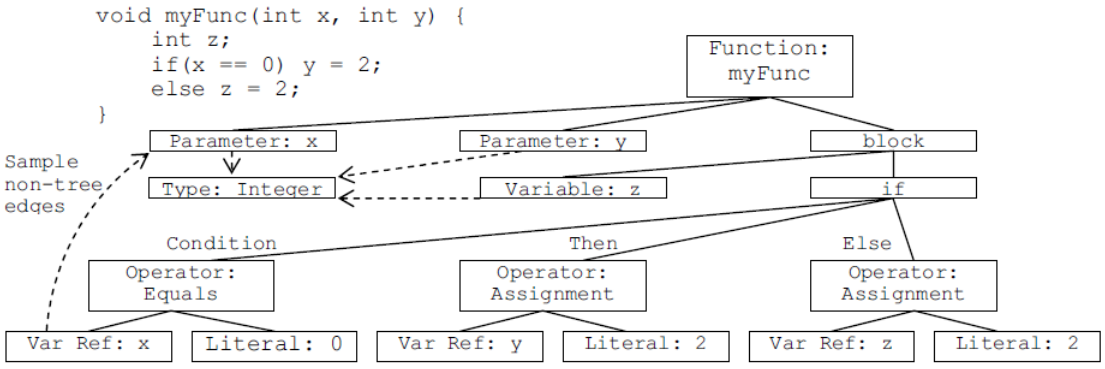


Figure 2.2: ASG Example Showing *non-tree edges* [Raghavan *et al.*, 2004]

XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is part of the Object Management Group (OMG) Knowledge Discovery Metamodel (KDM). XMI is a XML schema (specific formal definition of XML structure) and XMI documents are XML-valid documents conforming to the schema [OMG *et al.*, 2011c].

It provides for model representation of Meta Object Facility (MOF) models in a standard XML format, commonly using the MOF definitions of Unified

Modelling Language (UML - Section 2.3.3) both of which (MOF and UML) are also OMG standards [OMG *et al.*, 2011b,c].

Provision is contained within the schema to further extend information types through the in-file definition of complex types as well as the option to specify meta data including information about the XMI document itself (owner, descriptions, etc) [OMG *et al.*, 2011c]. In addition to providing for abstract analysis of code parsed into XMI (the primary use of XMI within reverse engineering) there is also provision for generation of code from XMI and the interchange of XMI data with the representative expressed functionality, between tools [OMG *et al.*, 2011c].

Although XMI can store data in different formats it is primarily used to encode UML data and supports different UML versions. Within the XMI schema every model class is described by an XML element that represents it [OMG *et al.*, 2011c]. Below is a simple XMI example representing UML information of a class:

```
<xmi:XMI
xmlns:uml="http://www.omg.org/spec/UML/20110701"
xmlns:xmi="http://www.omg.org/spec/XMI/20110701">
<uml:Class name="C1" xmi:type="uml:Class" xmi:id="_1">
<ownedAttribute
xmi:type="uml:Property" xmi:id="_2" name="a1"
visibility="private"/>
</uml:Class>
</xmi:XMI>
```

XMI XML listing of class C1 in UML format, from OMG et al. [2011c]

Unified Modelling Language (UML)

The Unified Modelling Language (UML) is an Object Management Group (OMG) specification for a visual language to document systems. It is designed to be applicable to all domains and platforms enabling all software artefacts to be specified, constructed and documented. UML has the ability therefore to support round-trip engineering with source code being generated by and/or transformed into UML. This is a feature supported by a number of UML tools or UML-enabled environments [OMG *et al.*, 2005; Sarkar *et al.*, 2013].

OMG *et al.* [2005] identify good generic design principles which are applied within UML as follows:

- Modularity - grouping is used, applying the concept of strong cohesion and loose coupling.
- Layering - package structures separate the language constructs from their use and an architectural pattern is used to separate concerns over different layers of abstraction.
- Extensibility - the language can be customised for specific platforms or domains either through the definition of new members or a new dialect.
- Reuse - through flexibility and granular construction the specification is suitable for wide reuse.

UML can be used to represent numerous types of diagrams, relationships and/or interactions within the system including class diagrams, use case diagrams and sequence diagrams. These diagrams can offer different levels of detail for example a class diagram can show just the class or include all members and properties contained within [OMG *et al.*, 2005]. Figures 2.3 and 2.4 show simple UML class diagrams with an increasing level of detail contained within the same model.

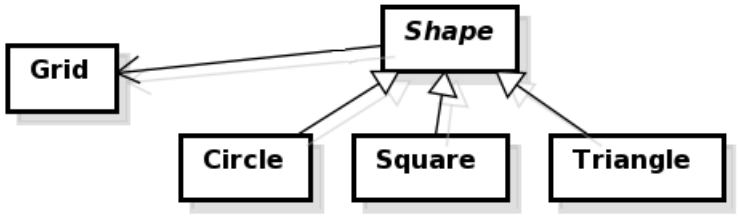


Figure 2.3: Simple UML Class Diagram showing inheritance and a relationships. Generated using reverse engineering in Astah Professional

2.3.4 Approaches to Software Reverse Engineering

To analyse system design a tool must analyse source code and extract relevant information. Commonly information is extracted from source code in a language-

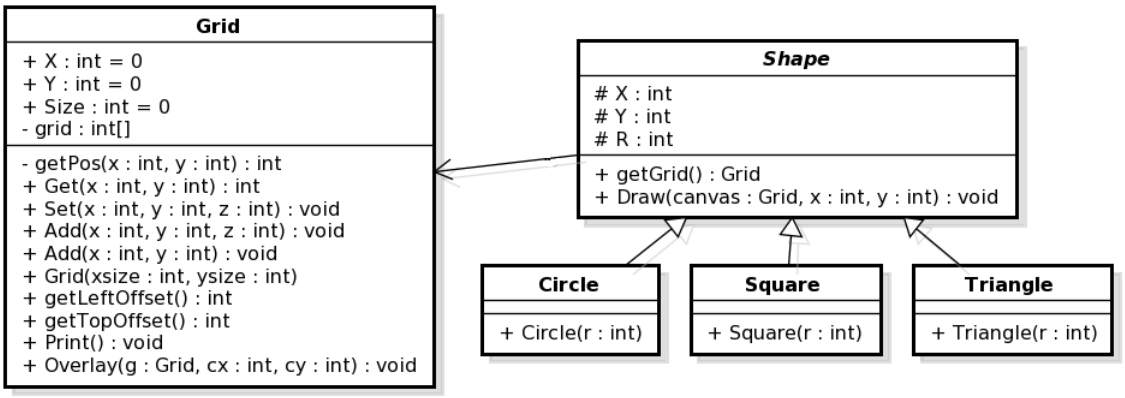


Figure 2.4: UML Diagram of the relationships shown in figure 2.3 but with added detail of properties and members

independent form (the main language abstracted notations are detailed in Section 2.3.3) building a representation of the code for analysis [Arcelli *et al.*, 2005].

Static and Dynamic Analysis

The majority of approaches use "static analysis" where the source code is directly analysed [Arcelli *et al.*, 2005; Flores and Aguiar, 2005; Labiche *et al.*, 2013; Shi and Olsson, 2006]. This can provide information into the structure and relationship of artefacts within the codebase. A complimentary "dynamic analysis" can be used where data is gathered at runtime allowing the analysis of code behaviour during execution [Cerulo, 2006; Labiche *et al.*, 2013; Rasool and Streitfdert, 2011]

Types of Analysis Process

Rasool and Streitfdert [2011] identify four main types of reverse engineering analysis:

- Structural - analysis of the structural relationships between different artefacts within the source code.
- Behavioural - uses dynamic analysis of runtime data combined with static analysis to identify behavioural characteristics within the system.

- Semantic - used in addition to structural and behavioural analysis to further refine understanding based upon information such as naming conventions. Primarily this aims to provide differentiation (for example of design patterns where the structure and/or behaviour are very similar - Section 2.3.4).

- Formal specification - used where available and a formal specification/approach was used during development, in effect a form of semantic analysis based upon more formalised criteria.

Design Pattern Recognition

Design patterns, as initially identified and defined by Gamma *et al.* [1995], are solutions to commonly occurring problems experienced in development. As they are frequently used - often core - constructs within software the identification and location of such patterns from source code is a common goal of reverse engineering [Arcelli *et al.*, 2005; Rasool and Streitfdert, 2011; Shi and Olsson, 2006; Uchiyama *et al.*, 2011]. Understanding of the design patterns used within a system can provide a significant step toward the overall goal of gaining a representation of the holistic function of the system [Counsell *et al.*, 2004].

Identification of design patterns in existing code ("design pattern recovery") is commonly attempted by transferring code into an abstract notation (as with most reverse engineering - Section 2.3.1) before using pattern matching to identify specific uses of design patterns [Rasool and Streitfdert, 2011; Shi and Olsson, 2006]. These processes can be iterative and themselves use design patterns. One such example, a tool called *SPQR*, uses a "*Pipes & Filters*" pattern to iteratively filter the source code into abstract notation until it reaches a suitable state for pattern recognition [Flores and Aguiar, 2005].

Identification of design patterns from source code is non-trivial for a number of reasons. Design patterns are designed to support forward engineering without pattern classification or identification for reverse engineering [Shi and Olsson, 2006]. Many of the patterns are structurally (and often behaviourally) very similar [Rasool and Streitfdert, 2011] and can vary in concrete implementation.

Even a relatively simple pattern such as the Singleton [Gamma *et al.*, 1995] which is also one of the most widely used [Uchiyama *et al.*, 2011] can vary signif-

icantly in implementation as the following Java listing illustrates:

Listing 2.1: Singleton Example

```
public class SingletonA
{
    private static SingletonA instance = null;
    protected SingletonA() { }
    public static SingletonA getInstance()
    {
        if (SingletonA.instance == null)
            SingletonA.instance = new SingletonA();
        return SingletonA.instance;
    }
}

public class SingletonB
{
    private static boolean hasInstance = false;
    public SingletonB() throws Exception
    {
        if (SingletonB.hasInstance)
            throw new Exception
                ("Can␣only␣have␣one␣instance");
        else
            SingletonB.hasInstance = true;
    }
}

// create A
SingletonA a = SingletonA.getInstance();
// return same instance
SingletonA a2 = SingletonA.getInstance();
// create B
```

```
SingletonB b = new SingletonB();
// create B again (throw exception as already exists)
SingletonB b2 = new SingletonB();
```

Listing showing two alternative implementations of the Singleton in Java, adapted from Gamma et al. [1995] and Uchiyama et al. [2011]

2.3.5 Tools for Software Reverse Engineering

Generally tools for reverse engineering fall into two categories: Those concerned with the identification of specific idioms such as design patterns (usually research projects) and those offering a fuller UML environment many with both forward and reverse engineering. In this section tools from both categories are identified and their features listed.

Feature Comparison

Table 2.1 shows a comparison between reverse engineering tool features. Many of these tools are commercial and have no peer-reviewed information available. In these cases the data below is taken from the manufacturer's website. When tools with multiple possible versions offering different levels of features are included the highest specification is used.

Tool	**Language (s)**	**UML**	**Idioms**	**Notes and XMI**
FUJABA [Fujaba, 2012]	Java	UML forward and backward	Design Patterns, Idioms, Anti-patterns	Meta-Object Facility (MOF) support for interoperability
DeMIMA [Guéhéneuc and Antoniol, 2008]	Java	None	Design pattern recognition	None
PINOT [PINOT, 2012]	Java	None	Design pattern recognition	None
SPOOL [SPOOL, 2012]	Java (possible others such as C++ and Smalltalk)	None	Design pattern recognition	None
Osprey [Shi and Olsson, 2006]	Java, C++ and some support for C	None	Design pattern recognition	None
Columbus [Ferenc *et al.*, 2002]	C++	None	Design pattern recognition	None
SPQR [Arcelli *et al.*, 2005]	Java	None	Design pattern recognition	None

DP++ [Philippow *et al.*, 2005]	C++	None	Design pattern recognition	Aimed at HPC applications
KT [Philippow *et al.*, 2005]	Smalltalk	None	Design pattern recognition	None
Agile J Structure Views [AgileJ, 2015]	Java	Class Diagrams	-	Eclipse plugin
Altova UModel [Altova, 2012]	Java, C#, VB.NET	UML 2.0	-	Eclipse/Visual Studio plugin, XMI Support
ArgoUML [ArgoUML, 2016]	Java (with plugin options available for other languages)	Reverse engineering to class diagrams supported (plugin possibilities and creation tools for other UML diagrams)	-	-
Astah Professional [ChangeVision, 2016]	Java, C++, C#	UML 2.x	-	XMI Support
BOUML [Pages, 2016]	C++, Java, PHP, MySQL	UML Projection	-	XMI Support

Enterprise Architect [SparxSystems, 2012]	C++, Java, C#, VB.Net, Visual Basic, Delphi, PHP, Python, ActionScript	All UML 2.3.1 Diagrams supported (unclear how many are supported in reverse engineering)	-	XMI Support
Rational Rhapsody (IBM) [IBM, 2015]	C++, C, Java, C#, Ada	UML 2	-	XMI Support
MagicDraw UML [NoMagic, 2012]	Java, C++, C#, CL (MSIL), CORBA IDL	UML 2	-	XMI Support
Modelio [Modeliosoft, 2012]	Java, C++, C#, SQL	UML 2	-	-
objectiF [Microtool, 2012]	C#, C++, Java, VB.NET, BPEL, XSD, WSDL	UML 2	-	Eclipse and Visual Studio plugin
Software Ideas Modeller [Rodina, 2012]	C#, VB.NET, Java, .NET Assemblies	UML 2.4	-	XMI Support

StarUML [StarUML, 2008]	Java	UML Class Diagrams	-	No longer maintained
Umbrello UML Modeller [Umbrello, 2012]	C++	No automated creation but classes and relationships available for manual use in UML	-	-
Visual Paradigm for UML [VisualParadigm, 2012]	Java, C++	UML 2	-	XMI Support
Rational Rose (IBM)	Java, C++, Ada, VB	UML 2	-	XMI Support
Borland Together	Java, C++, CORBA IDL	UML 1.4+2.0	Design Pattern support in creation only	XMI Support

Table 2.1: Comparison of Reverse Engineering Tools

2.4 Source Code Repository Mining

Source code repository mining refers to the collection and analysis of data from within software source code repositories such as git, CVS, or other historic collections. These repositories are used as a central store of source code and other related artefacts shared between development teams. Changes to these artefacts are made by developers and then "committed" to the repository. Information

available within the repository commonly includes incremental versions of (or changes to) source code files or other artefacts, such as documentation, along with meta data surrounding changes committed to the repository such as the developer, files changed and comments. This data is seen as a rich potential source of information about software offering the ability to gain comprehension into areas such as the evolution of a software package, bug fixing, semantic relationships between artefacts and developer activity either at a given point in time or mapped over the life of a project [Allamanis and Sutton, 2013; Kagdi *et al.*, 2007a; Ramadani and Wagner, 2016; Williams and Hollingsworth, 2005].

Although mining is performed most often against source code repositories themselves, other historic sources of information relating to software development are included within the field. These sources, identified in Table 2.2, may be mined on their own or in conjunction with each other. For example matching source code changes with bug reports [Hassan, 2008; Williams and Hollingsworth, 2005].

Extraction of data from source code repositories can be complex, as although an increase in open-source projects makes an ever-growing volume of such repositories available to researchers, the repository management software itself is not designed with this type of data collection in mind. Successful extraction and use of data does, however, continue to grow as does interest in the possibilities inherent in such large collections. In many ways this is demonstrated by the manner in which raw data extraction from complex repository formats has now become commonplace and the primary challenge remains how to analyse and use this data to form useful information [Hassan, 2008]. Development processes may also lead to unrelated changes being made as a single transaction which can further complicate meaningful analysis [Herzig and Zeller, 2013].

Source	Details
Source Code Repositories	All development history of a project including all changes to files and meta data. May also include large repositories holding information about a number of projects e.g. Sourceforge.net
Bug Repositories	Track the history of bugs including reporting, potentially narrative of investigation, and closure/resolution
Archived Communications	Track interpersonal communication about a project such as mailing lists, chat logs; general archived communications
Deployment Logs	Records of specific deployments for example error or transaction logs

Table 2.2: Examples of software repository types [Hassan, 2008; Kagdi *et al.*, 2007a]

2.4.1 Types of Repository Mining

Repository mining has often been used to try and comprehend or analyse software evolution over time [Kagdi *et al.*, 2007a], as well as in a more applied fashion to improve bug/error detection [Williams and Hollingsworth, 2005]. Also, of specific interest to this project, to rebuild or refine traceability links between code and documentation [Ali *et al.*, 2013; Antoniol *et al.*, 2002; Kagdi *et al.*, 2007b; Marcus and Maletic, 2003] or between code artefacts [Kagdi *et al.*, 2007a]. Before covering traceability recovery in terms of requirements (section 2.4.2) and dependencies (section 2.4.3), other main types of repository mining will be identified.

Understanding and/or Visualising Software Evolution

Comprehension of large software systems can be a major challenge for organisations and new developers. Many solutions exist to aid in the comprehension of the static system as it currently is, such as reverse engineering (section 2.3) and technical documentation when available. One use of source code mining is there-

fore to provide more detailed and historic information about system components including dependencies enabling their life to be traced and details such as the original change commit messages and notes to be recovered [Hassan, 2008]. In a wider context, information from multiple projects can be used to gain insight into the general form of software evolution; how large open-source projects grow and change, and levels of change over time [Allamanis and Sutton, 2013; Kagdi *et al.*, 2007a].

Developer or Team Analysis

Repositories contain meta data concerning who committed changes to what artefacts and when. Not only can this data be used for simple analysis such as the number of active developers or the change in activity patterns over time but, especially in combination with other types of repository such as mailing lists or bug reports, enable the profiling of activity on one or more projects [Hassan, 2008; Williams and Hollingsworth, 2005].

Bug Prediction

The ability to predict and so prevent or mitigate bugs is in many ways one of the holy grails of software engineering. If such knowledge was available project resources could be more finely targeted and testing focused [Hassan, 2008]. In terms of such analysis Graves *et al.* [2000] found a direct relationship between the number of changes or previous bugs in software with the number of bugs occurring in the future with more recent changes or bugs being more significant than historic ones. Use of such metrics as a more effective predictor of bugs compared with traditional complexity measures has also been demonstrated using the codebase from the Eclipse project [Moser *et al.*, 2008].

Enabling and Encouraging Reuse

Large projects naturally contain large amounts of code. In many cases this is poorly documented or there is a lack of awareness as to the precise content or purpose. This confusing picture does little to encourage reuse with poor visibility

of existing APIs or frameworks. Mining of repository data could allow for location of such "lost" artefacts for example by identifying previous components in which they have been used and providing the developers with this code and any contemporaneous comments or documentation [Hassan, 2008].

2.4.2 Source Code Mining for Traceability Recovery

Recovery of requirements traceability has become an increasing focus of research efforts. These are often centred on rebuilding links between higher and lower level documents within a software system e.g. from documentation to source code. These approaches typically use Information Retrieval (IR) techniques, such as textual comparison, which attempts to find similarities between artefacts at different levels. Performance has been mixed and often generally quite low in terms of balancing both recall (recovered over expected links) and precision (correctly identified links) [Ali *et al.*, 2013; Antoniol *et al.*, 2002]. A typical process for text-based IR processing of source and documentation is shown in figure 2.5.

One possibility for increasing the performance of traceability recovery is by incorporating information contained within repositories. Benefits from this can be realised in a number of ways: a historic analysis of a previous time when documentation and source code were closely linked with good traceability can be performed with artefacts then traced forward through their evolution. Documentation which was regularly changed at the same time as specific source artefacts may also indicate relationships [Ali *et al.*, 2013; Kagdi *et al.*, 2007b].

Ali *et al.* [2013] present such an approach building on their previous work [Ali *et al.*, 2011] that combines repository information with traditional IR traceability recovery techniques giving an average increase of 22.7% precision and 7.66% recall.

2.4.3 Change Coupling for Dependency Analysis

In addition to using repository data to augment IR processes for requirement traceability recovery, it is also a potential source of information relating to dependencies and relationships between source code artefacts such as classes. As with documentation and source, a high correlation in change coupling (co-committal; artefacts being changed at the same time as part of the same commits) may indi-

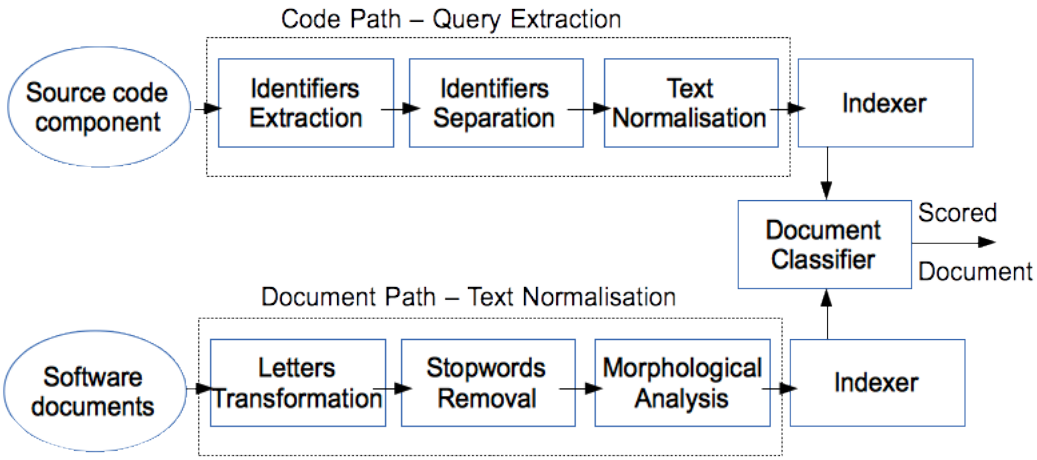


Figure 2.5: Possible traceability link recovery method [Antoniol *et al.*, 2002]

cate a relationship [Kagdi *et al.*, 2007a; Ramadani and Wagner, 2016; Ying *et al.*, 2004].

Work by Bieman *et al.* [2003] on software visualisation showed that there were apparent links between source code artefacts shown by change coupling that did not appear through static analysis (reverse engineering). Beyer and Noack [2005] introduced a clustering model for co-changes demonstrating that areas of commonality can be found and clustered but in most cases some artefacts cannot be sufficiently grouped, however the possibility of incorporating additional information from the changes (such as size of change) as well as other information sources holds the prospect of increased accuracy.

2.5 Clustering of Related Objects

A rich potential source of information with relation to structural components or other artefacts within software is their similarity (or dissimilarity) in terms of a number of different data sources. Data sources can include reverse engineering (class structural relationships) or source code repositories (co-commit semantics) as used by [Beyer and Noack, 2005]. Once initial data has been generated from a given source not only is it possible to determine specific similarities but to cluster components into groups which can be used for analysis, further grouping (sub-clusters) or comparison between data sources. Clustering has been used in the domain of software analysis including clustering of reverse-engineered data [Anquetil and Lethbridge, 1999], and although not using the direct clustering techniques here, methods used for design pattern detection (Section 2.3.4) may use forms of specialised grouping akin to clustering.

2.5.1 Generation of a Normalised Dissimilarity Matrix

To facilitate comparison between different data sources showing relationships between components (multi-variate analysis) it is necessary to convert the data to a standard form, a normalised dissimilarity matrix [Rogers and Girolami, 2011]. The steps followed from Rogers and Girolami [2011] and Du [2010] to achieve this are as follows.

The first step is to identify commonalities between components and generate a matrix similar to that shown in Table 2.3 for commonalities between components A, B, and C for data source (variable) X. This is a similarity matrix, it indicates how many points of similarity exist between A, B, and C.

	A	B	C
A	-	-	-
B	2	-	-
C	3	4	-

Table 2.3: Similarity or Commonalities of Components A, B, C for a given data source X

A similarity matrix can be converted to a distance matrix by subtracting each similarity value from the maximum similarity.

$$\forall X_i \in X : X'_i = X_{max} - X_i \tag{2.1}$$

In the case of the example data shown in Table 2.3 the resultant distance matrix would be as shown in table 2.4.

	A	B	C
A	-	-	-
B	2	-	-
C	1	0	-

Table 2.4: Distance Matrix of Components A, B, C for a given data source X

More commonly used is a dissimilarity matrix where the dissimilarity is the square root of the distance as calculated by equation 2.2, with example results in table 2.5.

$$\forall X_i \in X : X'_i = \sqrt{X_{max} - X_i} \tag{2.2}$$

	A	B	C
A	-	-	-
B	1.414	-	-
C	1	0	-

Table 2.5: Dissimilarity Matrix for Components A, B, C for a given data source X

To normalise ranges, ensuring consistent impact when using multi-variate input (multiple data sources for comparison where sources have different absolute numbers of similarities), it is also required to further subtract the mean and divide by the standard deviation. As negative dissimilarities are non-nonsensical and should be zero, a function Pos is defined in equation 2.3 to turn negative values into zero. This process shown in equation 2.4 with example output in table 2.6.

$$\begin{aligned} Pos(d) &= d, \qquad if \;\; d > 0 \\ Pos(d) &= 0, \qquad otherwise \end{aligned} \tag{2.3}$$

$$\forall X'_i \in X' : X''_i = \frac{Pos(X'_i - \bar{X'})}{X'_{sd}} \tag{2.4}$$

	A	B	C
A	-	-	-
B	0.838	-	-
C	0.269	0	-

Table 2.6: Normalised Dissimilarity Matrix for components A, B, C for a given data source X

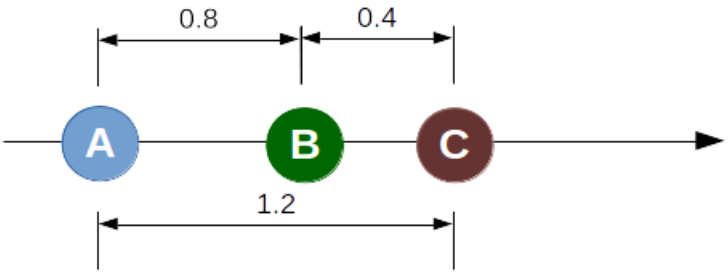


Figure 2.6: Data from Table 2.7 Projected in One Dimension

2.5.2 Multi-Dimensional Scaling (MDS)

Multi-Dimensional scaling is a technique used to add additional dimensions to multivariate data to a facilitate more accurate representation within a spatial model for analysis or visualisation [Pilch, 2009].

For example consider the distance data represented in Table 2.4. The distances between components are as follows: $A \Rightarrow B = 1$, $A \Rightarrow C = 1.5$ and $B \Rightarrow C = 1$. In a singular dimension these distances cannot be completely resolved as B and C must be one unit apart ($B \Rightarrow C = 1$) but at the same time B and C are different distances from A. To resolve this some alteration of the data would be required perhaps averaging distances between points to find a "best fit".

	A	B	C
A	-	-	-
B	1	-	-
C	1.5	1	-

Table 2.7: Distance Matrix of Components A, B, C

The attempted projection of this data on a single dimension can be seen in Figure 2.6.

To alleviate this problem additional dimensions could be added. If the above example was scaled to two dimensions it would be possible to satisfy the distance requirements completely. The position of the points in two-dimensional space would be calculated where the Euclidean distance (or similar distance measure) was closest to the absolute distance between the data points. Such a projection

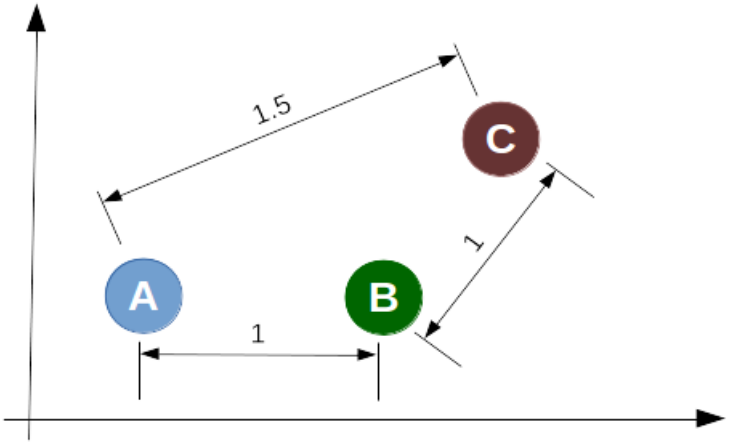


Figure 2.7: Data from Table 2.7 Projected in Two Dimensions

is shown in Figure 2.7. The higher the number of dimensions the easier it is to satisfy distance requirements and keep the integrity of the data however higher dimensions become harder to project and display [Pilch, 2009].

$$A \Rightarrow B \approx \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2} \tag{2.5}$$

2.5.3 Clustering Techniques

There are a significant number of highly varied techniques to approach clustering of related data items including randomised initialisation of a set number of clusters to highly complex models [Du, 2010; Rogers and Girolami, 2011]. As clustering was not explored in great detail within the project only two methods are formally reviewed with other candidates identified.

K-Means Clustering

K-Means clustering is a process by which data is partitioned into K partitions where each partition is regarded as a cluster. Each data point is within one partition and hence one cluster. The basic process is iterative by nature and begins by randomly selecting K points as centres (centroids) of prototype or prospective partitions. Data points are then assigned to the nearest centroid

usually based on simple Euclidean distance. Centroids are then relocated to the mean centre of their respective data points and the process is repeated with data points reassigned to their nearest centroid. Once the assignments stabilise (no further changes to data point assignment occur), the partitions are finalised [Du, 2010].

Because of the randomised nature of the initial points the partitioning can be very imprecise and although iterations will naturally lead to partitions becoming more refined the output can be highly varied and more suited to certain types and distributions of data [Du, 2010; Rogers and Girolami, 2011].

Variations on K-Means exist including K-Mode (the mode rather than the mean of partitioned data is used for positioning of the centroid) and K-medoid (where actual data points are used as centroids e.g. the data point nearest to the mean centre of a partition is used) [Du, 2010]. Methods are also in use to improve both the refinement and consistency of output by using informed as opposed to random starting positions for centroids [Rogers and Girolami, 2011], something which is implemented in Weka models [Du, 2010].

Expectation-Maximisation Clustering

Expectation-Maximisation (EM) clustering is an approach that uses statistical probability calculations to try and find then apply the best model to fit the presented data. Unlike K-Means there is no requirement to pre-specify the number of clusters, though this is possible. Rather, sensitivity values are set which inform the process. Estimates of model parameters are initially randomly selected and then repeatedly tested and refined until the required quality is met [Du, 2010].

Alternative Methods

From Anquetil and Lethbridge [1999] a number of other potentially applicable hierarchical clustering techniques are identified:

- Single linkage aka closest neighbour.
- Complete linkage (furthest neighbour).

- Weighted average linkage - using the mean of relations between clusters to form a hierarchy.
- Unweighted average linkage - the size of the cluster is also considered and the entities are not weighed.

2.6 Change Impact Analysis

Software is inherently under continual pressure to change, and this is especially true for the most successful or widely used software [Brooks, 1987; Lehman and Belady, 1985]. With such high rates of change it is important to be able to understand the impact individual changes may have on the software system as a whole. Bohner and Arnold [1996] define software change impact analysis as the ability to identify possible effects resulting from a change to a software system, e.g. the unanticipated and undesired impact on other components relating from the change which can lead to an error state.

2.6.1 Methods

The majority of software change impact approaches, and those of particular interest to this thesis, involve the analysis of the source code in one form of another and fall into two distinct categories; static and dynamic analysis [Lehnert, 2011; Li *et al.*, 2013]. This is the same as with the more generalised reverse engineering detailed in Section 2.3.4, with some work on the combination of both types or incorporation of other data. Generally, change impact analysis is a very specialised form of reverse engineering and analysis, with the goal of predicting impact rather than purely aiding general comprehension [Queille *et al.*, 1994].

In his review of software change impact analysis Lehnert [2011] groups the methods used by techniques included in the review based on their similarity. The following definition of methods is based upon this grouping:

Call Graphs

Generating a call hierarchy throughout the software to build graphs representing which components call others, to allow the identification of the potential propagation of change. This most often operates at the method or function level of granularity and may have to include a number of test cases for specific operation to facilitate tracing.

Dependency Analysis

Components can depend on each other in a number of ways such as control, data storage or object relationships (generalisation etc). Such dependencies are especially prevalent in object-orientated systems where more traditional analysis will generate a huge number and range of dependencies, many of which are of minor importance. A number of approaches therefore exist to gauge the significance of these relationships based upon factors such as their type and frequency.

Program Slicing

Slicing is based around static dependency analysis and identifying then removing instructions which are not relevant to the analysis criteria such as those that do not affect the state of a variable or component being analysed. The ultimate aim being to decompose the software system into just those specific statements or elements that do have a potential effect on the target of change to the point where the remaining system can then be analysed or visualised easily.

Execution Traces

A dynamic method uses that runtime analysis to determine the path of execution from component to component (i.e. from method to method) and recording these. Specific approaches vary from modification of source code to enable specific entrance and exit logging of components, to analysing the stack through general debugging or memory inspection. As with all dynamic-only methods only the actual traces can be used, so rare or untested states are not detected in the analysis, and a large set of test cases may be required.

Explicit Rules

In a strictly controlled and enforced system domain built following explicit rules, expert knowledge of this domain can be used to determine the effect a component will have on other components. Such a method relies on consistency throughout the system and that all development has taken place only in precise accordance with the governing rules without even minor variations.

Information Retrieval (IR)

This method revolves around using natural language processing of some form, either on code or other artefacts, to build a set of relationships upon which change impact analysis can be performed. In general terms these methods operate as with most IR approaches by identifying "meaningful words" from the source material and then building models to relate their use within the analysed system using techniques such as ranking.

Probabilistic Models

The application of mathematical models of probability such as Markow chains or Bayesian Belief Networks (BBN). Such a method tries to resolve the "likelihood" (probability) of a component being impacted by a given change. Using source data generated by another method such as call graphs or history mining, numerous techniques are used to weight and rank the input data to calculate probabilities.

History Mining

This method uses analysis of historical changes within a software system (most commonly through repository mining, Section 2.4) to determine impact coupling between components. The specific approaches used to the data vary widely and include analysing individual changes (snapshots of the software before and after), or simply recording "real world" changes if the information is available e.g. "component A was changed before and impacted B and C that time". Because of the large volume of raw data usually contained within history, ranking and weighting is often used to determine which detected relationships have a greater strength.

A large range of granularity from the class to statement level is found in different methods.

Combined Technologies

Significant advantages are potentially found through the combination of different methods and information sources. For example, combining static analysis of call graphs with history mining methods could provide a more accurate and validated impact analysis, to leverage both the detailed analysis of actual source code with the less detailed but factually accurate mined change history. Note; this is explored in more detail in Section 2.6.3.

Other Technologies

Lehnert [2011] does mention a further "unclassifiable" approach which is intended to provide predictive impact analysis and involves a simile to slicing in order to focus on key code by repeatedly analysing the code and "brushing away" code not related to the data fields in question.

2.6.2 Predictive Change Impact Analysis and Measurement

Predictive change impact analysis (PCIA) is concerned with identifying the potential or likely impacts that a change has on other elements within software. This is distinct from identifying which components *should be* changed to accomplish a specific outcome or indeed directly analysing *previous* changes purely to analyse what their impact was [Kagdi and Maletic, 2006; Lehnert, 2011].

Li *et al.* [2013] define a general process for predictive change impact analysis, shown in Figure 2.8. First the intended change and software are analysed to identify which components will actually be changed, which forms the *change set*. Using a method of analysis (Section 2.6.1), an estimate is made of what other components are likely to be impacted by the change, the *estimated impact set* (EIS). The EIS can contain more than a simple set of components and may include confidence (probability) values or other meta information.

The change is now performed on the system, and any components found to have been actually affected by the change are put into an *actual impact set* (AIS).

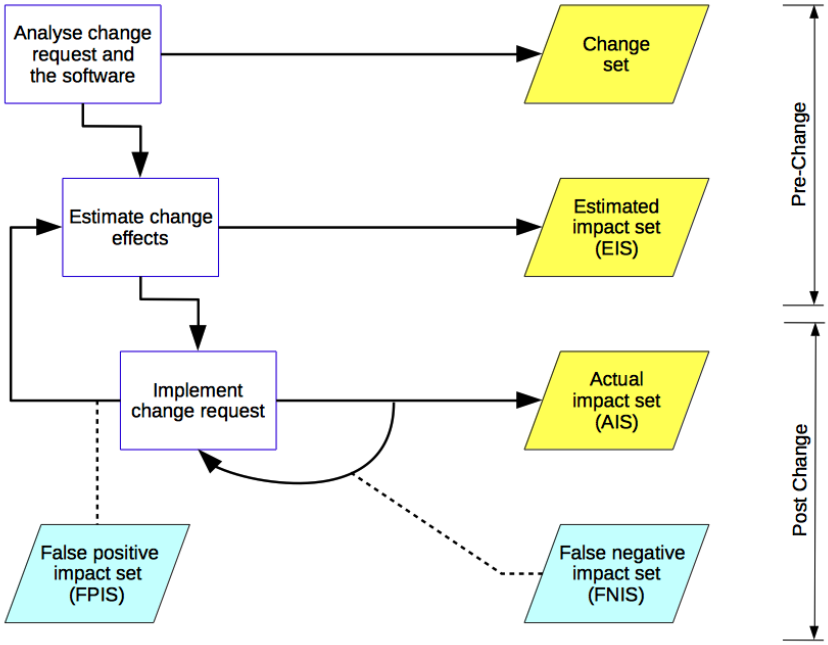


Figure 2.8: Predictive Change Impact Analysis Process, adapted from Li *et al.* [2013]

With the expected (EIS) and actual (AIS) extent of the changes known it is possible to generate a *false positive impact set* (FPIS) with impacts that were expected but not found, and a *false negative impact set* (FNIS) of those actually impacted but not expected. The accuracy performance of the approach can also be measured using standard information retrieval metrics.

Li *et al.* [2013] presents this standard method as being iterable in nature, where the output can be fed back into the system to further improve performance but many approaches are one-pass and standalone e.g. analyse the status *now* and use that as the basis for the EIS.

2.6.3 Combined Information Sources

One area of more recent attention is that of using sources of information in combination to provide a better, more accurate, change impact model. There are a number of approaches which combine some form of static with dynamic analysis such as Huang and Song [2007, 2008]. Another method is to include other

artefacts of the software development process, usually architecture diagrams or other design documentation, to aid in tuning the code analysis elements. This type of combination is used in approaches by Bohner [2002], Briand *et al.* [2002], and Khan and Lock [2009].

Use of historical information about the software, in combination with more traditional source code analysis, is another area that offers possibilities for significant improvement on traditional impact analysis [Lehnert, 2011]. Work by Kagdi and Maletic [2006] and Kagdi and Maletic [2007]; Kagdi *et al.* [2007b] ties the recovery of traceability information from mining of source code repositories into the sphere of impact analysis, when used in combination with code analysis, creating a change impact prediction model of "source-code aware mining" to increase accuracy. The historic analysis element contained in Kagdi and Maletic [2006] identifies "change coupling" by looking at previous change requests in the context of these being "successful" or not and using these patterns further.

More recent work by Dit *et al.* [2014] uses the concept of historical coupling through co-changes; the concept that if historically a component is frequently changed at the same time as another component, outside of any other information about the change, it is inferred that the components are related. Applying this to the field of change impact, Dit *et al.* [2014] mine SVN repositories and reconstruct each interim version, performing an analysis at the method level of granularity and combine this with source code analysis. The results show "significantly better" results than obtained by a single approach such as that employed by the industry standard JRipples (JRipples results from Buckner *et al.* [2005]).

Chapter 3

Benchmarking Reverse Engineering

3.1 Introduction

Reverse engineering of systems, through static analysis of source code, is the most common way to comprehend existing software without other documentation. In this context reverse engineering is the process of taking source code and running an analysis to detect certain information often including structural elements (such as classes in an object-oriented language) and relationships between them. It is therefore a very suitable approach to first take when considering forensic traceability. The general motivations and aims of reverse engineering are detailed in Sections 2.3.1 and 2.3.2 but mainly revolve around aiding comprehensibility or detecting specific properties of software code. The contributions described in this Chapter are published in [Cutting and Noppen, 2015] and [Cutting and Noppen, 2014].

With ever growing numbers of valuable but poorly documented legacy codebases within organisations, reverse engineering has become increasingly important and its use widespread in situations where the only reference material available is the source code itself. In response, there are a wide number of reverse engineering techniques, which offer a variety in both their focus and the precise techniques they use to perform the analysis.

The most common techniques are based around extracting general structural information, often generating diagrams such as Unified Modelling Language (UML) projections, while more specialised approaches aim to detect specific elements such as design patterns or particular code arrangements (Section 2.3.4). Variations exist in the precise methods used to extract information with the majority of techniques using static analysis of source code while some techniques use dynamic analysis with examination of states performed at runtime [Arcelli *et al.*, 2005; Kumar *et al.*, 2015; Labiche *et al.*, 2013; Rasool and Streitfdert, 2011; Roscoe, 2011]. A list and comparison of major reverse engineering tools is contained in Section 2.3.5, specifically Table 2.1.

Although a wide number of approaches and tools exist with various different focuses their efficacy or fidelity has not been evaluated in a reproducible manner. Further, it is difficult to compare their effectiveness against each other as no standard set of targets exist to support this goal over multiple approaches; a

problem also found in the verification and validation of new tools and techniques [Sim *et al.*, 2003]. Any performance evaluations which do exist are specific to an approach or technique such as those defined by Arcelli *et al.* [2005]; Bellay and Gall [1997]; Bellon *et al.* [2007]; Meyer [2006]; Rasool and Streitfdert [2011]. It is therefore currently not possible to gain a comparative understanding of performance for a range of tasks or to validate new techniques or approaches. To address this gap and determine the suitability of reverse engineering for recovery and reconstruction of design information this chapter introduces a benchmark of such targets, the Reverse Engineering to Design Benchmark (RED-BM) [Cutting, 2013], that can be used to compare and validate existing and new tools for reverse engineering.

To determine performance a benchmarking approach was defined (Section 3.2) before being applied against a number of real-world targets (Section 3.3), and the results evaluated (Section 3.5).

3.2 The Reverse Engineering to Design Benchmark (RED-BM)

RED-BM [Cutting, 2013] facilitates the analysis of reverse engineering approaches based on their ability to reconstruct accurate class diagrams of legacy software systems. This is accomplished by offering the source code of projects of differing size and complexity as well as a number of reference UML class diagrams that serve as the gold standard to compare the output of reverse engineering against. In addition the benchmark provides a set of performance metrics that facilitate the comparison of reverse engineering results, for example class detection, to reference models including a gold standard and a number of meta-tools to aid in the analysis of tool outputs.

The benchmark allows ranking of reverse engineering approaches by means of an overall performance measure that combines the performance of an approach with respect to a number of criteria, such as successful class or relationship detection. This overall measure is designed to be extensible through the addition of further individual measures to facilitate specific domains and problems. In ad-

dition the benchmark provides analysis results and a ranking for a set of popular reverse engineering tools which can be used as a yardstick for new approaches. Full details, models, targets, results as well as a full description of the measurement processes used can be found in Cutting [2013]. Although based on Java source code, the concepts and measurements are applicable to any object-oriented language and the benchmark could be extended to include other languages.

3.2.1 Target Artefacts

The benchmark consists of a number of target software artefacts that originate from software packages of varying size and complexity. The benchmark artefact targets represent a range of complexity and architectural styles from standard Java source with simple through high complexity targets using different paradigms such as design patterns and presentation techniques. Candidate projects were chosen for consideration on the basis that they could represent this range and also that the source code was freely available and re-distributable under open source licencing. This enables a graduated validation of tools as well as a progressive complexity for any new tools to test and assess their capabilities. Also, included within RED-BM are a set of gold standards for class and relationship detection against which tool output is measured. These standards were created by manual analysis supported by tools as described in Section 3.4.

The range of artefacts is shown in Table 3.1 where large projects (e.g. JHotDraw) are broken down into constituent components. The projects were broken into smaller parts allowing a single source to form multiple targets so that a common style and format could be used in a variety of target sizes. Further the real-world application of reverse engineering is seldom to gain understanding of an entire large project but rather a specific insight into a component or sub-package within. Breaking the projects down in such a manner allows for the benchmarking then to be conducted in a process analogous to a real world analysis. In addition the table contains statistics on the number of classes, sub-classes, interfaces and lines of code for each of the artefacts.

The source code used was downloaded as source code or cloned from the repository and so does not represent a specific release. The code used represents

the current available stable source on the 11th of December 2012, and that source is bundled within [Cutting, 2013].

Artefacts were chosen for inclusion on the basis that they provided a range of complexity in terms of lines of code and class counts, used a number of different frameworks with diverse code structure, offered some pre-existing class diagrams and were freely available for distribution (under an open source licence). Two artefacts (ASCII Art Examples A and B) were created by the author specifically for inclusion as a baseline offering a very simple starting point with full UML design and use of design patterns.

Cactus, although depreciated by the Apache Foundation, has a number of existing UML diagrams and makes use of a wide number of Java frameworks. Eclipse was included primarily owing to a very large codebase which contains a varied use of techniques. The large codebase of Eclipse also provides for the creation of additional targets without incorporating new projects. JHotDraw has good UML documentation available both from the project itself and some third-party academic projects which sought to deconstruct it manually to UML. As with Eclipse, Libre Office provides a large set of code covering different frameworks and providing for more targets if required.

Table 3.1. Software Artefact Targets of the RED-BM [Cutting, 2013]

Software				
Target Artefact	**Main Classes**	**Sub Classes**	**Inter-faces**	**Lines of Code (SLOC)**
ASCII Art Example A				
Example A	7	0	0	119
ASCII Art Example B				
Example B	10	0	0	124
Eclipse				
org.eclipse.core. commands	48	1	29	3403
org.eclipse.ui.ide	33	2	6	3949
Jakarta Cactus				
org.apache.cactus	85	6	18	4563
JHotDraw				
org.jhotdraw.app	60	6	6	5119
org.jhotdraw.color	30	7	4	3267
org.jhotdraw.draw	174	51	27	19830
org.jhotdraw.geom	12	8	0	2802
org.jhotdraw.gui	81	29	8	8758
org.jhotdraw.io	3	2	0	1250
org.jhotdraw.xml	10	0	4	1155
Libre Office				
complex.writer	11	33	0	4251
org.openoffice.java. accessibility.logging	3	0	0	287
org.openoffice.java. accessibility	44	63	1	5749
All bundled code (sw + accessibility)	241	173	33	39896

3.2.2 Reverse Engineering Performance Measures

RED-BM enables the systematic comparison and ranking of reverse engineering approaches based on a set of performance measures or metrics. These measures quantify the performance of reverse engineering approaches and are based on

accepted quality measures for reverse engineering such as successful detection of classes and packages [Koschke, 2003][Roman and Cox, 1993]. These measures provide a basic foundation for measurement to be built on and represent the most common requirement in reverse engineering for detection of structural elements. Further, as seen in Section 3.5, these measures by themselves are already capable of differentiating between different ranges of tool performance. The performance of tools with respect to a particular measure is expressed as the fraction of the relevant data that has been successfully captured. Individual measures are then used in conjunction with each other to form a weighted compound measure of overall performance. In RED-BM three base measures are defined to assess the performance of reverse engineering tools and approaches, which are recall metrics with no consideration of precision at this stage:

- **Cl:** The fraction of classes successfully detected
- **Sub:** The fraction of sub-packages successfully detected
- **Rel:** The fraction of relationships successfully detected

Each of these measures are functions that take the actual content of a system to be reverse engineered s (the perfect reverse engineering result) and a result r that is produced by a reverse engineering approach when applied to s. The formal definition of theses three base measures are as follows:

$$Cl(s,r) = \frac{C(r)}{C(s)} \quad , \quad Sub(s,r) = \frac{S(r)}{S(s)} \quad , \quad Rel(s,r) = \frac{R(r)}{R(s)} \tag{3.1}$$

where

$C(x)$ is the number of correct classes in x

$S(x)$ is the number of correct (sub-)packages in x

$R(x)$ is the number of correct relations in x

The overall performance P of a reverse engineering approach for the benchmark is a combination of these performance measures. The results of the measures are combined by means of a weighted sum which allows users of the benchmark to adjust the relative importance of, e.g., class or relation identification. The

overall performance of a reverse engineering approach that produces a reverse engineering result r for a system s is defined as follows:

$$P(s,r) = \frac{w_{CL}CL(s,r) + w_{Sub}Sub(s,r) + w_{Rel}Rel}{w_{CL} + w_{Sub} + w_{Rel}} \tag{3.2}$$

where

$$w_{CL} + w_{Sub} + w_{Rel} = n \in IR_{+}$$

In this function, w_{CL}, w_{Sub} and w_{Rel} are weightings that can be used to express the importance of the performance in detecting classes, (sub-)packages and relations respectively.

3.2.3 Complexity Categories

To further refine the evaluation of the reverse engineering capabilities of approaches the artefacts of the benchmark are divided into three categories of increasing complexity; C1, C2 and C3. These categories allow for a more granular analysis of tool performance at different levels of complexity. For example, a tool can be initially validated against the lowest complexity in an efficient manner only being validated against higher complexity artefacts at a later stage. The complexity classes have the following boundaries:

- **C1:** $0 \leq$ number of classes ≤ 25
- **C2:** $26 \leq$ number of classes ≤ 200
- **C3:** $201 \leq$ number of classes

The complexity categories are based on the number of classes contained in the target artefact. As source code grows in size both in the single lines of code and the number of classes, it becomes inherently more complex and so more difficult to analyse [Bellay and Gall, 1997; Fenton and Pfleeger, 1998]. While a higher number of classes does not necessarily equate to a system that is harder to reverse engineer, this metric was chosen as it provides a quantitative measure without subjective judgement.

The bounds for these categories were initially chosen as seemingly reasonable placeholders to start with. When they were applied within the experiment however results demonstrated a noticeable drop-off in detection rates observed in the tools with these bounds as can be seen in Section 3.5. For this reason they have been kept at their initial levels for this analysis. However, any user of the benchmark can introduce additional categories and relate additional performance measures to these categories to accommodate for large scale industrial software or more specific attributes such as design patterns. The extensibility aspect of this work is explained in more detail in Section 3.2.4.

3.2.4 Extensibility of the Benchmark

Extensibility of Measurements

RED-BM's included performance measures provide a foundation to evaluate and compare current standards of reverse engineering. To accommodate the continual advancements in this field the performance measure aspect of the benchmark is extensible. Any user of the benchmark can introduce new performance measures, such as the fraction of successfully detected design patterns in a given code base. Once a gold standard has been determined for a specific detection within the artefacts it can be tested against tool output (as explained in Section 3.2.3 for the initial criteria). With these new measures the performance of approaches can be defined for specific reverse engineering areas. To facilitate uniform comparison of performance a performance measure is defined as a function M that for a given system contents s and a reverse engineering output r returns a result that maps to the domain [0..1], where 0 means the worst and 1 the best possible performance.

In addition to providing means for creating new performance measures, the possibility to create new compound performance measures (i.e., measures that are compiled from a set of individual performance measures) is provided. Formally, a compound measure is defined as a function C that maps a system s and its reverse engineering result r through a number [1..n] of performance measure functions M to the domain [0..1], where 0 means the worst and 1 the best possible performance:

$$C(s,r) = \frac{\sum_{i=1}^{n} w_i M_i(s,r)}{\sum_{i=1}^{n} w_i} \tag{3.3}$$

For example an input function M could be the detection measure defined in equation 3.2.

In this expression w_i is the weighting that determines the importance of the individual performance measure i. Note that the performance measures introduced in Section 3.2.2 conform to this definition and, therefore, can be seen as an example of the extensibility of the benchmark.

To further illustrate how researchers and practitioners can use this mechanism to specialise the application of RED-BM a performance measure is created that acknowledges the capability of an approach to detect design patterns during reverse engineering. This is an active research field for which according to a literature survey and expert discussion a specialised benchmark is not available.

According to literature the detection of creational and structural design patterns is easier than behavioural design patterns [Philippow *et al.*, 2005] and so creational and structural design patterns are grouped together. Therefore, two new performance measures are introduced D_b for the successful identification of creational and structural design patterns(D_{cs}) and behavioural design patterns (D_b) for a system s and reverse engineering result r which are recall metrics:

$$D_{cs}(p,s) = \frac{P_c(r) + P_s(r)}{P_c(s) + P_s(s)} \quad , \quad D_b(p,s) = \frac{P_b(r)}{P_b(s)} \tag{3.4}$$

where

$P_c(x)$ is the number of creational design patterns in x
$P_s(x)$ is the number of structural design patterns in x
$P_b(x)$ is the number of behavioural design patterns in x

In addition to these performance measures additional measures are introduced

that demonstrate how to consider negative influences on performance. In this case, precision is also considered through false positives (i.e. falsely identified) creational and structural design patterns (E_{cs}) and behavioural design patterns (E_b) a reverse engineering approach produces as part of the overall result:

$$E_{cs}(p,r) = 1 - \frac{F_c(r) + F_s(r)}{P_c(r) + P_s(r) + F_c(r) + F_s(r)} \tag{3.5}$$

$$E_b(p,r) = 1 - \frac{F_b(r)}{P_b(r) + F_b(r)} \tag{3.6}$$

where

$F_c(x)$ is the number of false positives creational design patterns in x
$F_s(x)$ is the number of false positives structural design patterns in x
$F_b(x)$ is the number of false positives behavioural design patterns in x

Using these measures it is also possible to introduce precision measurement first by deriving the false discovery rate (FDR for system s and result r):

$$FDR(s,r) = \frac{F_c(r) + F_s(r) + F_b(r)}{P_c(s) + P_s(r) + P_b(r)} \tag{3.7}$$

And from this calculate the precision value PRE:

$$PRE(s,r) = 1 - FDR(s,r) \tag{3.8}$$

These individual performance measures for design patterns can now be combined into a single compound performance measure DPR for design pattern recognition in system p with reverse engineering result r that includes weightings for

each individual component:

$$DPR(p,r) = \frac{w_{D_{cs}} D_{cs} + w_{D_b} D_b + w_{F_{cs}} F_{cs} + w_{F_b} F_b}{w_{D_{cs}} + w_{D_b} + w_{F_{cs}} + w_{F_b}} \tag{3.9}$$

where

$$w_{D_{cs}} + w_{D_b} + w_{F_{cs}} + w_{F_b} = n \in IR_+$$

Extensibility in Data Exchange

Another prominent aspect that needs to be addressed for a reusable and extensible benchmark is the gap that exists between input and output formats of various reverse engineering tools. Indeed, to make further use of reverse engineering output, for example, between tools or for re-projection of UML there is an Object Management Group (OMG) standard, the XML Metadata Interchange (XMI) format [OMG *et al.*, 2011c]. XMI is a highly customisable and extensible format with many different interpretations. Therefore, in practice, tools have a wide variation in their XMI output and exchange between reverse engineering tools, useful for interactive projection between tools without repetition of the reverse engineering process, is usually impossible [Jiang and Systä, 2003]. This variance in XMI format also hinders use of XMI data for further analysis outside of a reverse engineering tool as individual tools are required for each XMI variation.

During the creation of the reverse engineering benchmark, two tools were developed, which could analyse Java source code identifying contained classes, and then check for the presence of these classes within XMI output. Detail on this tooling and the capability of the tools beyond the benchmark is contained in Section 3.4.

3.3 Application of the Benchmark

To analyse the effectiveness of this benchmark, a range of commercial and open source reverse engineering tools were applied (shown in Table 3.2) to each target artefact. Each of the tools is used to analyse target source code, generate

UML class diagram projections (if the tool supports such projections) and export standardised XMI data files. Although the source code target artefacts used for testing are broken down into the package level for analysis, the reverse engineering process is run on the full project source code to facilitate package identification. The output produced by each of the tools is subsequently analysed and compared to the generated gold standard using a benchmark toolchain specifically created for comparison of class detection rates (see Section 3.4). Finally, a manual consistency check is performed between the standard tool output and XMI produced to identify and correct any inconsistencies where a tool had detected an element but not represented it within the generated XMI. For this analysis weightings were used as stated, where all types of elements are of equal weight ($w_{Cl} = w_{Sub} = w_{Rel} = 1$), and categories of increased complexity have higher weight in the compound measure ($w_{C1} = 1, w_{C2} = 1.5, w_{C3} = 2$). These weightings were used to model the belief that larger more complex code is harder to reverse engineer and hence worth a higher score. They are fully configurable by any user of the benchmark however so only used with these values for demonstrating the benchmark.

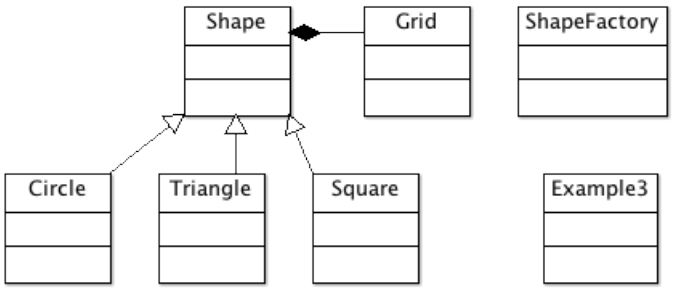


Figure 3.1. Reference Class Diagram Design for ASCII Art Example A

When analysing the results a wide range of variety can be observed even for simple targets such as Example A, one of the simplest targets with just 7 classes, as depicted in Figure 3.1. Please note that although Example A only contains generalisation and composition relationships other target artefacts contained associations and these were included in the measurement. It can be seen in Figure 3.2 that Software Ideas Modeller failed to identify and display any relationship

between classes. Other tools such as ArgoUML [ArgoUML, 2016] (Figure 3.3) were very successful in reconstructing an accurate class diagram when compared to the original reference documentation.

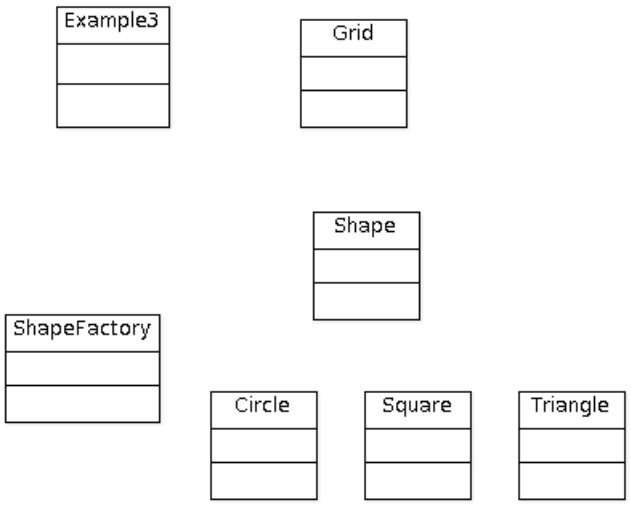


Figure 3.2. ASCII Art Example A Output for Software Ideas Modeller

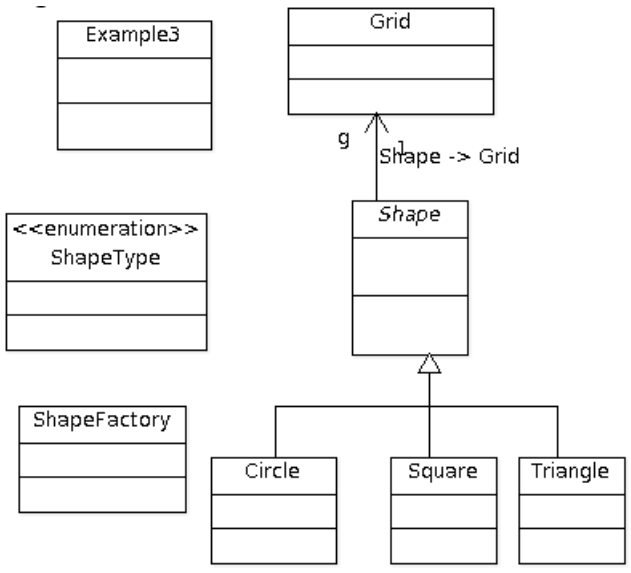


Figure 3.3. ASCII Art Example A Output for ArgoUML

Another aspect in which differences are obvious relates to tool presentation, particularly when the target artefact is a Java package comprising of sub-packages

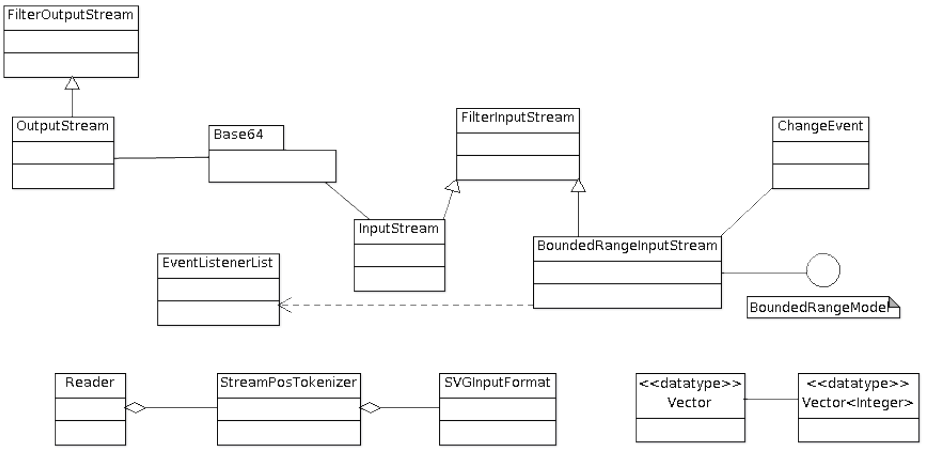


Figure 3.4. org.jhotdraw.io Output from Astah Professional (reconstructed)

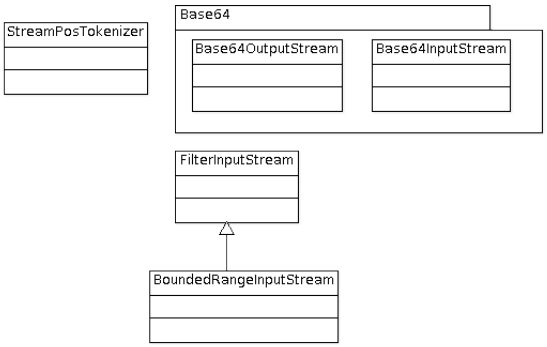


Figure 3.5. org.jhotdraw.io Output from Rational Rhapsody (reconstructed)

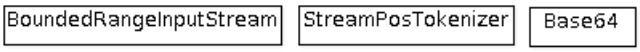


Figure 3.6. org.jhotdraw.io Output from ArgoUML

nested to multiple levels. Some of the different ways tools visualise this, even for a single nesting level, is shown by the *org.jhotdraw.io* target. Tool output varies from a simple display of classes and packages at the top level (ArgoUML, Figure 3.6), a partial decomposition of top-level sub-packages showing contained constituent items (Rational Rhapsody, Figure 3.5), to a full deconstruction showing all constituent parts and relationships but without indication of sub-package containment (Astah Professional, Figure 3.4).

In stark contrast to tools which performed well (e.g., Rational Rhapsody and

ArgoUML) a number of tools failed to complete reverse engineering runs of benchmark artefacts and even crashed repeatedly during this procedure. The result of which is that they are classified as detecting 0 classes for those target artefacts. While some tools failed to output valid or complete XMI data, a hindrance to their usability and ease of analysis, this has not affected their performance evaluation as their performance could be based on direct manual analysis of their UML projection.

Table 3.2. List of Tools and Versions for Use in Evaluation

Tool Name (Name Used)	**Version Used (OS) Licence**
ArgoUML	0.34 (Linux) Freeware
Change Vision Astah Professional (Astah Professional)	6.6.4 (Linux) Commercial
BOUML	6.3 (Linux) Commercial
Sparx Systems Enterprise Architect (Enterprise Architect)	10.0 (Windows) Commercial
IBM Rational Rhapsody Developer for Java (Rational Rhapsody)	8.0 (Windows) Commercial
NoMagic Magicdraw UML (MagicDraw UML)	14.0.4 Beta (Windows) Commercial
Modeliosoft Modelio (Modelio)	2.2.1 (Windows) Commercial
Software Ideas Modeller	6.01.4845.43166 (Windows) Commercial
StarUML	5.0.2.1570 (Windows) Freeware
Umbrello UML Modeller (Umbrello)	2.3.4 (Linux) Freeware
Visual Paradigm for UML Professional (Visual Paradigm)	10.1 (Windows) Commercial
IBM Rational Rose Professional J Edition (Rational Rose)	7.0.0.0 (Windows) Commercial

3.4 Benchmark Toolchain

As identified previously, one of the main challenges for reverse engineering approaches is to correctly identify classes in source code. This makes the fraction of correctly recognised classes a frequently used metric to assess reverse engineering performance. However, establishing performance for reverse engineering approaches based on this metric requires the analysis of a significant amount of data as potentially large code bases need to be cross-validated against tool-produced results.

It is possible to generate the required data through manual processing with a manual analysis of the source code and reverse engineering tool output. This solution rapidly becomes cumbersome and error prone as code becomes inherently more complex and difficult to analyse as size increases [Bellay and Gall, 1997; Fenton and Pfleeger, 1998]. To alleviate this problem many tools exist that provide metrics for, in particular, Java code including a count of the number of classes. It would be a trivial exercise to simply count the number of classes produced in reverse engineering output either within the tool itself or through simple analysis of XMI (a standard for the interchange of reverse engineering information produced by the majority of tools) [OMG *et al.*, 2011c] or UML/XML output. However some tools include utility and other third-party classes in their output and therefore it is not sufficient to simply count classes in code and compare against a separate count from tool output. To measure tool performance accurately it is necessary to verify that the classes detected by the tool are actually present within the source code.

These constraints make systematic analysis and comparison of reverse engineering approaches particularly hard to perform and repeat in an effective manner. During the creation of RED-BM measures were incorporated to differentiate the performance of tools, with class detection a key measure (Section 3.2.2).

Specifically RED-BM includes a measure which defines class detection performance as the proportion of classes found by a reverse engineering tool. More precisely this performance measure determines the fraction of performance by dividing the number of correctly identified classes by the total number of classes that should have been identified i.e. recall. It is clear that for this measure the

number of classes within the source code and the number of these classes identified in the tool output need to be determined. This is a problem identified earlier as typical for the assessment of reverse engineering approaches. It became clear that this data could not be produced manually as RED-BM consists of 16 target artefacts each containing between 5 and 447 classes. These are then processed by 12 reverse engineering tools resulting in a large amount of potential manual analysis. Not only would this have limited the accuracy of the benchmark results, it would also have prevented others from repeating benchmark analysis and produce additional results.

As a result a support tool suite was developed as part of the RED-BM tools to facilitate analysis of both source code and tool output in a largely automated fashion. One tool (*jcAnalysis*) analyses Java source code and detects classes, outputting this class list in a simplified XML meta format (DMI). A reverse engineering tool is used to process the same source code, the output from which is produced as XMI or UML/XML. A second tool (*xmiClassFinder*) is then used to analyse both the XMI and DMI output. This highlights any classes found to be missing through comparison and producing a ratio for classes found by the reverse engineering tool. The workflow is illustrated in Figure 3.7 where square boxes illustrate processes (RED-BM tool processes are highlighted) and the waved boxes represent documents or artefacts. The process followed and the tools are introduced in more detail within the following sections.

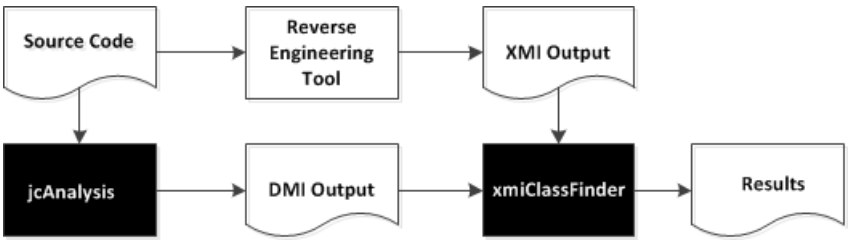


Figure 3.7. RED-BM Process with Toolchain Elements Highlighted

3.4.1 Java Code Analysis

In order to make a meaningful comparison with reverse engineering tool output it is necessary to first analyse Java code to generate a list of classes contained and

output that list in a suitable format for automated analysis. This is accomplished through the *jcAnalysis* tool.

Although numerous tools are available to generate Java source metrics including class counts and lists of components, none of these are able to output class lists in a format suitable for comparison. Consequently, development of a new tool was required which was then cross-validated with existing tools both in terms of class count and also direct manual comparison of output.

jcAnalysis is a regular expression based recursive analyser to find classes and interfaces meeting a specific set of criteria. From the top level of a given directory it will find all .java files, optionally recursing to include sub-directories. Options can be given to specify the package to be analysed and whether sub-packages of the target package should be included. Example output from *jcAnalysis* is shown in Figure 3.8.

Java files found within the search will be analysed and their package along with constituent classes, sub-classes, and interfaces retrieved through regular expressions. Classes found to match the package criteria will then be included in a list and displayed on the screen.

Once analysis is completed *jcAnalysis* can then optionally output the class list in an XML format (described in section 3.4.2) for later use.

3.4.2 Exchange of Information in Meta Format

To allow for exchange between the tools at different stages of the workflow a standard format is required to contain class information. This information is generated by analysis of source code and then used in a comparison with reverse engineering tool output. For this purpose the DMI (Dave's Metadata Interchange) format is defined.

The DMI specification in the version used by *jcAnalysis* and *xmiClassFinder* (version 0.01) is a highly simplified XML schema to represent classes in a format conducive for further analysis. Classes or interfaces are listed in individual elements including their type (class or interface in version 0.01), simple name and fully-qualified (*package.subPackage.class*) form.

An example listing of a DMI 0.01 file is as follows:

```
Filename: /home/sys/rfm11aqu/Dropbox/PhD/tooldemo/cactus/src/ServletTestCaller.j
ava
Package: org.apache.cactus.internal.server
ClassCount = 1 (main: 1, sub: 0)
EnumCount = 0, InterfaceCount = 0
Lines of Code: 40

FullClass: org.apache.cactus.internal.server.FilterTestCaller
ClassName: FilterTestCaller
Line: public class FilterTestCaller extends AbstractWebTestCaller
Filename: /home/sys/rfm11aqu/Dropbox/PhD/tooldemo/cactus/src/FilterTestCaller.ja
va
Package: org.apache.cactus.internal.server
ClassCount = 1 (main: 1, sub: 0)
EnumCount = 0, InterfaceCount = 0
Lines of Code: 36

Totals:
Package:
ClassCount = 91 (main: 85, sub: 6)
EnumCount = 0, InterfaceCount = 18
Lines of Code: 4563

[cactus]$ []
```

Figure 3.8. jcAnalysis running for Cactus target

```
<dmi version="0.01">
<item type="class" name="someClass"
 package="somePackage"
 fullname="somePackage.someClass" />
<item type="class" name="otherClass"
 package="somePackage"
 fullname="somePackage.otherClass" />
<item type="class" name="aClass"
 package="somePackage.subPackage"
 fullname="somePackage.subPackage.aClass"
 />
</dmi>
```

3.4.3 XMI Analysis and Comparison

Once Java code has been analysed to identify constituent classes (section 3.4.1), to produce a performance measure these constituent classes must be compared

Table 3.3. Simplified Comparative XMI Output from Tools

ArgoUML	**Enterprise Architect**	**Astah Professional**
<**UML:Class** xmi.id = "..." name = "Circle" visibility = "package" ...> <**UML:GeneralizableElement.generalization**> <**UML:Generalization** xmi.idref = "..." /> </**UML:GeneralizableElement.generalization**> ... <**UML:Classifier.feature** <**UML:Operation** xmi.id = "..." name = "Circle" visibility = "public" ...> </**UML:Class**>	<**packagedElement** xmi:id = "..." name = "Circle" visibility = "package" ...> <**ownedOperation** xmi.id = "..." name = "Circle" visibility = "public" ...> ... <**generalization** xmi:type = "uml:Generalization" xmi:id = "..." general = "..." > </**packagedElement**>	<**UML:Class** xmi.id = "..." name = "Circle" version = "0" ...> <**UML:ModelElement.namespace**> <**UML:Namespace** xmi.idred = "..." </**UML:Namespace** ... <**UML:ModelElement.visibility** xmi.value = "package" /> ... <**UML:GeneralizableElement.generalization**> xmi.idref = "..." /> xmi.idref = "..." /> </**UML:GeneralizableElement.generalization**>

against output from reverse engineering tools. For this purpose the *xmiClassFinder* tool was created to perform such a comparison.

xmiClassFinder is an XML analyser capable of identifying classes within XMI and UML/XML output from reverse engineering tools. It can also compare this output with a DMI file highlighting any missing classes (those present in the source code represented in the DMI but not present in the XMI), calculating a percentage for classes found divided by those present. An example output of this process is shown in Figure 3.9.

Widely differing XMI formats from tools means *xmiClassFinder* operates a multi-pass approach, starting with the most common format and on failure trying other formats in turn. Classes highlighted as unfound within the XMI can then be manually checked either in the XMI or directly in the reverse engineering tool. The diversity in range of XMI output can be seen in Table 3.3.

Although *xmiClassFinder* concentrates purely on the class information it forms a starting point from which a fully-featured XMI analyser has been developed, capable of the identification of all relevant structural elements of XMI

```
NOTFOUND: org.apache.cactus.internal.HttpServiceDefinition (HttpServiceDefinitio
n)
NOTFOUND: org.apache.cactus.internal.configuration.Configuration (Configuration)
NOTFOUND: org.apache.cactus.internal.configuration.WebConfiguration (WebConfigur
ation)
NOTFOUND: org.apache.cactus.internal.configuration.JspConfiguration (JspConfigur
ation)
NOTFOUND: org.apache.cactus.internal.configuration.ServletConfiguration (Servlet
Configuration)
NOTFOUND: org.apache.cactus.internal.configuration.FilterConfiguration (FilterCo
nfiguration)
NOTFOUND: org.apache.cactus.internal.CactusTestCase (CactusTestCase)
NOTFOUND: org.apache.cactus.internal.server.runner.XMLConstants (XMLConstants)
NOTFOUND: org.apache.cactus.internal.server.WebImplicitObjects (WebImplicitObjec
ts)
NOTFOUND: org.apache.cactus.spi.client.connector.ProtocolState (ProtocolState)
NOTFOUND: org.apache.cactus.spi.client.connector.ProtocolHandler (ProtocolHandle
r)
NOTFOUND: org.apache.cactus.spi.client.ResponseObjectFactory (ResponseObjectFact
ory)
NOTFOUND: org.apache.cactus.spi.server.TestController (TestController)
NOTFOUND: org.apache.cactus.spi.server.ImplicitObjects (ImplicitObjects)
Classes = 109, Found = 90, Accuracy = 83.0%
[cactus]$
```

Figure 3.9. xmiClassFinder running for Cactus target using Software Ideas Modeller XMI output

output including detected relationships as well as classes. This not only provides for more detailed automated measurements of reverse engineering performance but also the direct comparison of output. Direct comparison of output could also theoretically allow the output from multiple tools of lower performance to be used in combination, providing better coverage.

3.5 Evaluation of Analysis Results

For the analysis of the results produced by the reverse engineering tools, a standard class detection performance measure is used for all targets (*CD*, based on equation 3.1). This measure produces an average performance value for each of the individual performance measures, both class and relationship detection rates, with equal weighting. Such a measure provides an overall indication of performance, with individual measurements and complexity levels providing further insight.

To further refine the evaluation of the reverse engineering capabilities of ap-

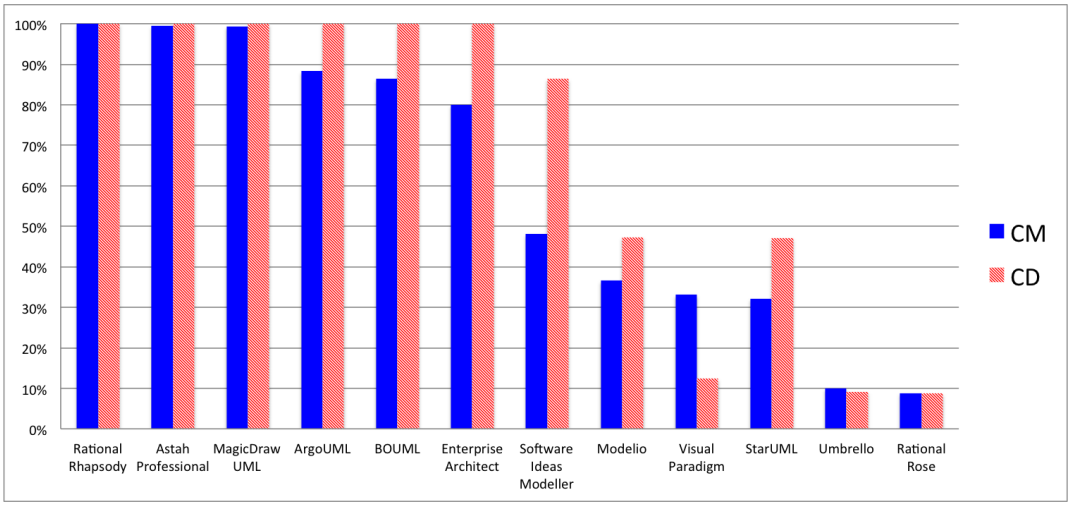


Figure 3.10. Overall Class Detection (CD) and Compound Measure (CM) Performance by Tool

proaches, the artefacts of the benchmark are divided into three categories of increasing complexity; C1, C2 and C3, as described in Section 3.2.3, with the boundaries as follows:

- **C1:** $0 \leq$ number of classes ≤ 25
- **C2:** $26 \leq$ number of classes ≤ 200
- **C3:** $201 \leq$ number of classes

Finally, compound measure *CM* is used based on equation 3.3, which contains the three complexity measures with weighting as follows: $w_{C1} = 1, w_{C2} = 1.5, w_{C3} = 2$; giving a higher weighting to target artefacts that contain more lines of code.

Using these performance measures a wide range of results between the tools used for analysis can be seen (Table 3.4). Some tools offer extremely poor performance, such as Rational Rose and Umbrello, either as they crashed or reported errors during reverse engineering or UML projection failing to detect or display classes and relationships entirely for some targets or just failed in detection. To avoid accidental bias or failure through incorrect operation of the applications great care was taken to follow instructions and where doubts about settings existed repeat the experiment with all possible configurations before recording the

Table 3.4. Criteria Results by Tool

Criterion > ∨ **Tool**	**CD** %	**C1** %	**C2** %	**C3** %	**CM** %	**Rank**
ArgoUML	100	98.15	75	100	88.27	1
Astah Professional	100	97.62	100	100	99.47	2
BOUML	100	92.59	75	100	86.42	3
Enterprise Architect	100	66.67	62.22	100	80.00	4
Rational Rhapsody	100	100	100	100	100.00	5
MagicDraw UML	100	98.15	100	100	99.38	6
Modelio	47.33	95.92	29.66	12.02	36.54	7
Software Ideas Modeller	86.41	62.15	41.48	46.04	48.10	8
StarUML	47.11	47.22	23.47	31.16	32.17	9
Umbrello	9.2	35.79	5.95	0	9.94	10
Visual Paradigm	12.42	38.18	51.68	16.67	33.12	11
Rational Rose	8.69	38.05	1.09	0	8.82	12

best output. A specific immediately obvious result was the poor performance of Rational Rose especially when considering the very high performance of Rational Rhapsody (both now IBM products from the same original software house). A concern was that the software was not being correctly operated and so the experiments were repeated a number of times with Rational Rose validated against documentation and the result was the same or worse.

As a general trend, the percentage of classes detected on average declined as the size of the project source code increased. As the number of classes detected varied significantly in different tools (Figure 3.10) so did the amount of detected relationships. To a degree this can be expected as if a tool fails to find classes it would also fail to find relationships between these missing classes. In this figure the difference between the standard class detection measure CD and the compound measure CM becomes clear as, for example, ArgoUML was very strong in class detection but performed at a slightly lower level on relation detection, which is explicitly considered in the compound measure. It is also interesting to note that Visual Paradigm offered better performance for the compound measure as opposed to class detection highlighting its superior ability to deal with relations and packages as compared to class detection.

Overall the benchmark identified IBM Rational Rhapsody as the best performer as it achieved the maximum score for the compound measure (100%) with two other tools, Astah Professional and MagicDraw UML coming in a close second scoring in excess of 99%. As the poorest performers this work highlighted Umbrello, Visual Paradigm and notably IBM Rational Rose which scored the lowest with a compound measure of just 8.82% having only detected 8.69% of classes. A detailed breakdown of the performance of the tools for individual targets is provided with the benchmark [Cutting, 2013].

This range of performance scores clearly shows a very marked differentiation between tools. At the top end some six tools score 80% or above in the compound measure, with three over 90%. In most a clear drop-off in detection rates are seen in the complexity measures as the size and complexity of the targets increase with an average measure score of 73.47%, 58.70% and 54.66% through the complexity categories C1, C2 and C3, respectively (Table 3.4 and Figure 3.11).

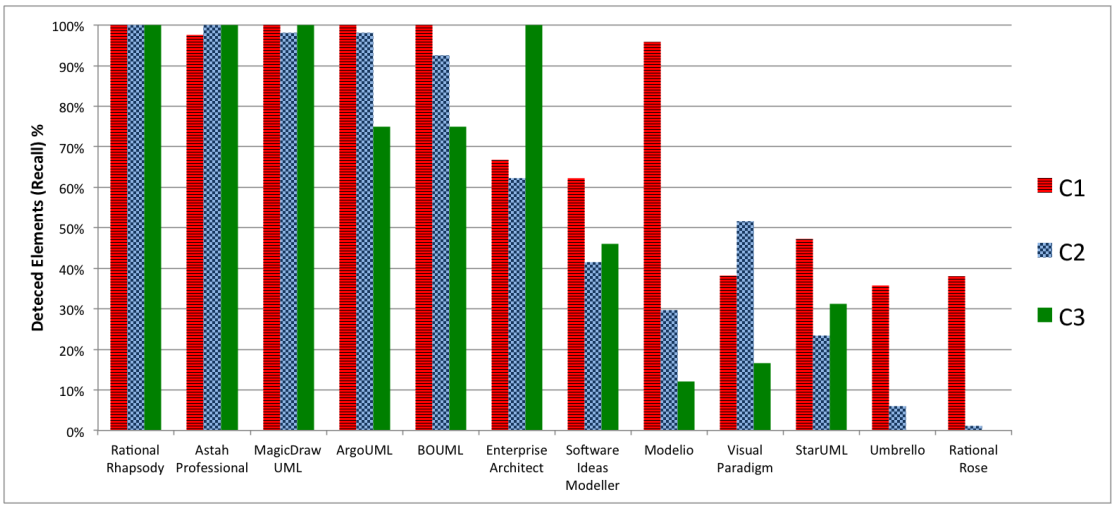


Figure 3.11. Tool Performance by Complexity Criteria

There is a noticeable distribution of tool performance for the compound measure; five score under 40%, six score in excess of 80% and only one lies in the middle (48.1%).

It is interesting to note that of the top four performing tools three are commercial with ArgoUML, a freeware tool, scoring 88.27%. This makes ArgoUML a significantly better performer than well-known commercial solutions such Soft-

ware Ideas Modeller and Rational Rose. For complete results, targets and reference documentation for this analysis please visit the benchmark website [Cutting, 2013].

Although outside the scope of this analysis, in general, it was also found that the workflow processes of some tools were much more straightforward than others. For example, Change Vision Astah Professional and IBM Rational Rhapsody provided for straightforward generation of diagrams with configurable detail (such as optional inclusion of members and properties within class diagrams) either during or immediately after reverse engineering. On the other hand, tools such as BOUML and IBM Rational Rose required considerable manual effort in the generation of class diagrams with the need for individual classes to be placed in the diagram although relationships between classes were automatically generated. For a number of tools the lack of usability was further aggravated as their reverse engineering process repeatedly crashed or returned numerous errors on perfectly valid and compilable source code.

3.6 Related Work

The use of benchmarks as a means to provide a standardised base for empirical comparison is not new and the technique is used widely in general science and in computer science specifically. Recent examples where benchmarks have been successfully used to provide meaningful and repeatable standards include comparison of function call overheads between programming languages [Gaul, 2012], mathematical 3D performance between Java and C++ [Gherardi *et al.*, 2012], and embedded file systems [Olivier *et al.*, 2012]. The Reverse Engineering to Design Benchmark (RED-BM) provides the ability for such meaningful and repeatable standard comparisons in the area of reverse engineering.

Previous work reviewing reverse engineering tools has primarily focused on research tools many with the specific goal of identification of design patterns [Arcelli *et al.*, 2005; Pettersson *et al.*, 2010; Philippow *et al.*, 2005; Roscoe, 2011; Uchiyama *et al.*, 2011], clone detection [Bellon *et al.*, 2007] or a particular scientific aspect of reverse engineering, such as pattern-based recognition of software constructs [Meyer, 2006]. A previous benchmarking approach for software reverse

engineering focused on pattern detection with arbitrary subjective judgements of performance provided by users [Fulop *et al.*, 2008]. The need for benchmarks within the domain of reverse engineering to help mature the discipline is also accepted [Sim *et al.*, 2003].

3.7 Conclusion

Having identified reverse engineering as the most common approach to automated recovery of design information, it was clear there was no empirical evaluation which could gauge nor benchmark performance of the myriad of available approaches. This led to the creation of the Reverse Engineering to Design Benchmark (RED-BM). This is a set of measures which can be applied to reverse engineering, along with a number of targets and gold standard output for comparison.

To analyse the effectiveness of RED-BM it was applied to a range of reverse engineering tools, ranging from open source to comprehensive industrial tool suites. This demonstrated that RED-BM offers complexity and depth as it identified clear differences between tool performance. In particular, using the compound measure (CM), RED-BM was capable of distinguishing and ranking tools from very low (8.82%) to perfect (100%) performance.

The results, therefore, show that even for information *expected* to be detected and returned by static analysis, there is a large variation in tool performance and presentation. Such output can not be completely trusted from the majority of tools and, even when all the expected structural elements and relationships are found, their presentation to the user for decision making is inconsistent. Ultimately even the best reverse engineering tools simply show static relationships, and say nothing about wider possible relationships such as semantic or runtime interactions.

It is therefore clear that though reverse engineering shows some of the relationships, a broader and more inclusive approach is required. One which can make use of single (or multiple) reverse engineering output, but in a more generic and usable fashion.

Chapter 4

Augmenting Reverse Engineering Using Repository Mining

4.1 Introduction

As demonstrated through benchmarking in Chapter 3 reverse engineering techniques can be imprecise, vary widely in output format, and in some cases miss even basic information such as generalisation relationships.

Generally however reverse engineering tools are reasonably good at detecting structural elements and relationships but not all relations in software systems are purely structural, as components can for example support the same function without having explicit links to each other. These relations cannot be detected by source code analysis alone. Such semantic links represent more subtle relationships within the software. There is some work within specialist reverse engineering tools to examine some semantic relationships (Section 2.3.4), but these are mainly focussed on textual analysis of element names only. As a result in general reverse engineering tools no account is taken of semantic relationships even when structural recall is perfect meaning important links may be missed.

This means that decisions dependent on a clear understanding of software structure are often made with imperfect and/or un-validated information. Consequently change decisions based purely on the information identified by these tools can therefore introduce significant risk.

One approach to address this issue is through the incorporation of additional sources of information beyond the static analysis performed by reverse engineering tools alone. Examples of other information sources include dynamic run-time analysis methods such as call tracing or stack analysis. Incorporating information from other sources could have a number of potential benefits such as enriching the picture (providing more detailed insight), validating one source with another, or identifying other forms or types of relationships unavailable from one source or another. A generic approach which allows for the uniform combination of different information sources is outlined in Chapter 5.

In this chapter a method is proposed that considers behavioural information stored in software repositories to address the issue of imperfect information. Mining of source code repositories, the meta data of which holds evolutionary information of a software project, is a wide field and is being exploited in a number of different domains (as seen in Section 2.4). This data is seen as a rich potential

source of information about software offering the ability to gain understanding of those areas such as the evolution of a software package, bug fixing, semantic relationships between artefacts and developer activity either at a given point in time or mapped over the life of a project [Allamanis and Sutton, 2013; Kagdi *et al.*, 2007a; Williams and Hollingsworth, 2005]. The specific focus of source code mining for this research is to find, and ideally quantify, semantic relationships between components within software in a presentation form similar enough to reverse engineering to allow combinational use.

Source code repositories are used by development teams as a central store of development artefacts such as source code or documentation files. The central store allows teams to collaborate together on a centrally held version of the software, each "committing" their individual changes to this central store. Every committal of changes can be stored incrementally with all previous versions available and certain meta data such as files changed, author and date is captured for each commit.

Extraction of useful data from source code repositories can be complex. Although an increase in open source projects makes an ever-growing volume of such repositories available to researchers, the repository management software itself is not normally designed with this type of data collection in mind. Successful extraction and use of data however continues to grow as does interest in the possibilities inherent in such large collections. In many ways this is demonstrated by the way in which raw data extraction from complex repository formats has now become commonplace and the primary challenge remains how to analyse and use this data to form useful information [Hassan, 2008].

Work by Bieman *et al.* [2003] on software visualisation showed that there were apparent links between source code artefacts shown by change coupling that did not appear through static analysis (reverse engineering) alone. Beyer and Noack [2005] introduced a clustering model for co-changes demonstrating that areas of commonality can be found and clustered but in most cases some artefacts cannot be sufficiently grouped. However the possibility of incorporating additional information from the changes (such as size of change) as well as other information sources holds the prospect of increased accuracy.

For the purposes of this research the aim is to use repository data analy-

sis to detect relationships between source code artefacts such as classes. Such relationships may be representative of a direct link in code, a dependency for example, or a looser form. The method used to detect such relationships is built around the concept of change coupling, e.g. what artefacts are more commonly changed together. For example a high correlation in change coupling (also called co-committal; artefacts changed at the same time as part of the same commit) may indicate a relationship between these artefacts [Kagdi *et al.*, 2007a]: *if A and B are commonly changed at the same time some form of semantic relationship between them can be inferred.*

Either of these two sources, reverse engineering or repository mining, on their own can yield useful information about relationship structures within software, but this information can be imprecise or overwhelming in volume [Grossman *et al.*, 2005]. Furthermore there is expected to be significant overlap, e.g., in the majority of cases, relationships between elements found by one method would also be detected by the other offering potential for cross-validation between information sources.

4.1.1 Problem Statement

Static analysis is imperfect as even when there is full recall of structural elements other important semantic relationships will not be detected. Such relationships may be found within the information contained by a source code repository. The problem is therefore to determine if it is possible to determine such semantic links and then use this in combination with reverse engineering output to build a more complete insight into a software system. Consider an example system where Figure 4.1 shows the output from static reverse engineering but no link has been found between the *PrintMethod* and *PrintException* classes through this technique, even though such a link does exist. Note the same system is used extensively in the worked example presented in Section 4.2.3.

4.1.2 Working Hypotheses

Meaningful semantic relationships between software components can be identified based on the frequency with which they have been changed together at the

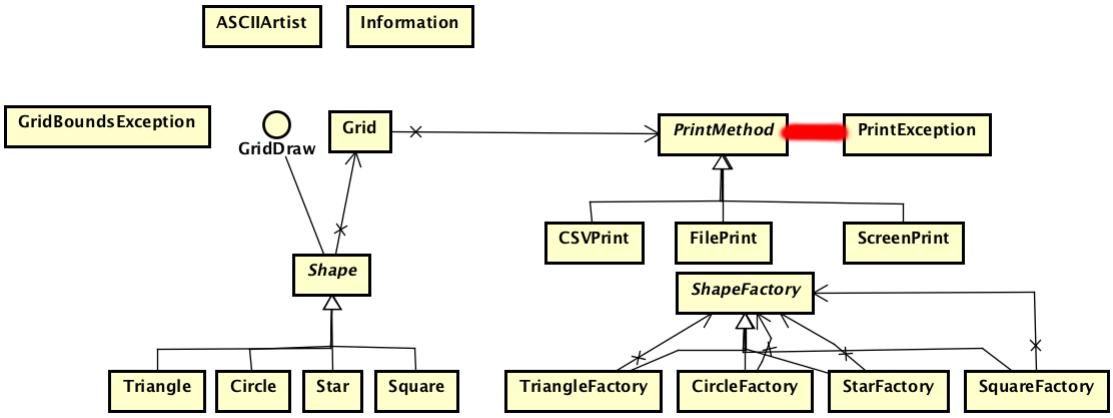


Figure 4.1. ASCII Artist Worked Example: Static Analysis with Missing Link Highlighted

same time by developers, e.g. the correlation of co-commits. The stronger the correlation the stronger the inferred relationship.

4.2 Approach

The approach to gain potentially useful information from repository data is built around the concept of co-commits, as detailed in the Introduction (Section 4.1). Although this is not a new concept in itself (see Section 2.4), defining a formal and repeatable approach to the consistent use of mined repository data is novel, and forms the basis of one aspect of the uniform multi-source approach defined in Chapter 5.

The process overview with the major steps is shown in Figure 4.2. First the raw repository data (step 1) is mined to gain a time series of commits within a given window of time (the period under analysis) (step 2). Each is a set of alterations made to the system, within the window, along with their constituent changes (step 3). An analysis phase is then used in which, for each commit in turn, every constituent class and pair of classes is identified (step 4) before a counter for that pair is incremented (step 5). For example a commit containing three components (A, B, C) would increment the relationship counters between each of these pairs $(A - B, A - C, B - C)$.

The result of this stage is a similarity table (step 6) showing, for each related

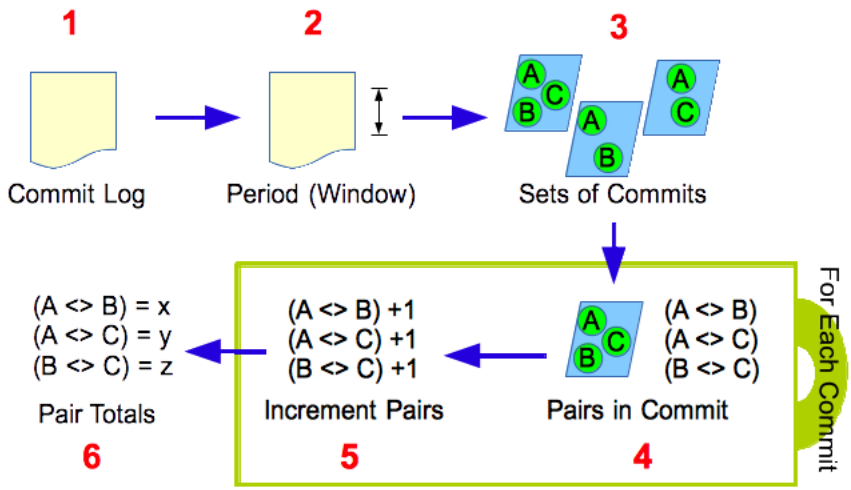


Figure 4.2. Overview of the Process Used to Extract Semantic Relationships Between Classes

component, the frequency of co-commits (the frequency at which they have been changed as part of the same update). In this form reverse engineering data can also be presented, as a similarity table containing relationships, and so the two sources can be used directly together.

4.2.1 Concrete Example

Consider a more concrete example of a software system following the process detailed in Section 4.2 and illustrated in Figure 4.2. The example has the following commit log (step 1) in time series order T shown in Table 4.1.

Table 4.1. Example Commit Log

T	**Components in Commit**
1	A, D
2	A, B, C
3	A, B
4	A, C
5	C, D

The period (window) under analysis (for which data should be gathered - step 2) is from time index 2 to 4 inclusive ($2 \leq T \geq 4$) which results in three commit

sets [A,B,C], [A,B], and [A,C] (step 3). For each of these each individual pair is identified (i.e. [A,B,C] = A<>B, A<>C, B<>C - step 4) and the value held for the similarity between these elements incremented by one (step 5). These steps and the corresponding running totals are shown in Table 4.2. The end state is the final pair totals (step 6).

Table 4.2. Example Commits Incremented to Pairs

T	**Components**	**Pairs/Increment**	**Totals**
Begin			A<>B=0, A<>C=0, B<>C=0
2	A, B, C	A<>B, A<>C, B<>C	A<>B=1, A<>C=1, B<>C=1
3	A, B	A<>B	A<>B=2, A<>C=1, B<>C=1
4	A, C	A<>C	A<>B=2, A<>C=2, B<>C=1
End			A<>B=2, A<>C=2, B<>C=1

4.2.2 Formal Definition

A *commit* C_x is defined as the *xth* addition to a version management system of an unordered set whose elements correspond to classes c that have been modified since the previous commit:

$$C_x : \{c_1, c_2, ...\} \tag{4.1}$$

Consequently a *version management system* V is defined as a completely ordered set of commits.

$$V : \{C_1, ..., C_n\} \tag{4.2}$$

A *commit window* $V_W(x, y)$ of a version management system V is defined as

the continuous subset of commits from C_x to C_y contained in V:

$$V_W(x,y) : \{C_x, C_{x+1}, ...C_{y-1}, C_y \mid C_i \in V, x \leq i \leq y\} \tag{4.3}$$

A *class set* S_V of a version management system V (or commit window W) is a set of all the unique classes contained within all the constituent commits:

$$S_V : \{c_1, ...c_n\} \in \{C_1, ...C_n\} \in V \tag{4.4}$$

Co-Committed Class Elements

A class c_x is defined as to be included within (part of) a commit C_z if it is in the set of classes described by C_z using function Po (part of):

$$\begin{aligned} Po(c_x, C_z) &= 1, \qquad if\ c_x \in C_z \\ Po(c_x, C_z) &= 0, \qquad otherwise \end{aligned} \tag{4.5}$$

Two classes c_x and c_y are said to be committed simultaneously (co-committed) with respect to a commit C_z if both c_x and c_y are part of C_z using function Co (co-committed):

$$\begin{aligned} Co(C_z, c_x, c_y) &= 1, \qquad if\ c_x \in C_z \wedge c_y \in C_z \\ Co(C_z, c_x, c_y) &= 0, \qquad otherwise \end{aligned} \tag{4.6}$$

The overall value of the relationship between two classes c_x and c_y is defined as the sum of their shared co-commits as function Re (relationship):

$$Re(c_x, c_y) = \sum_{C_z \in V} Co(C_z, c_x, c_y) \tag{4.7}$$

Confidence Values

When two classes c_x and c_y are co-committed they are defined as having a co-commit or *semantic relationship*. This semantic relationship models that the

classes c_x and c_y are part of the same semantic concept such as a component or an architectural concept (UI).

As Version Management Systems provide an imperfect source of information on semantic relations, not every commit will contain a clear co-commit of a given class pair. The confidence in a semantic relation existing between two classes is quantified based on the frequency with which they are co-committed in the repository. The confidence in a semantic relationship is expressed as a number on the domain [0..1] where 0 means "*no confidence they are part of the same semantic concept*" and 1 means "*certain they are part of the same semantic concept*". To calculate the confidence of the *semantic relationship* between two classes c_a and c_b in window W of repository V the function S_{V_W} is defined that calculates the number of co-commits for a pair of classes c_a and c_b (this is the same fundamental calculation as equation 4.7 but for a given window W).

$$S_{V_W}(c_a, c_b) = \sum_{C_x \in V_W} Co(C_x, c_a, c_b) \tag{4.8}$$

and define A_{V_W} to be the set of all classes in window V_W.

$$A_{V_W} = \bigcup V_W \tag{4.9}$$

Let the function M that determines the maximum co-commmit value for any pair of classes in a window V_W be

$$M(V_W) = \max\{S_{V_W}(c_a, c_b) \mid (c_a, c_b) \in A_{V_W} \times A_{V_W}\} \tag{4.10}$$

Finally, the function R_{V_W} that determines the confidence of the semantic relation between classes c_a and c_b in window V_W is defined as

$$R_{V_W}(c_a, c_b) = \frac{S_{V_W}(c_a, c_b)}{M(V_W)} \tag{4.11}$$

4.2.3 Illustrative Example

To illustrate this approach an example will be used, making references to the steps shown in the process outline Figure 4.2, and the equations provided in Section

4.2.2. Taking a Java project called ASCII Artist as an example, the constituent classes are detailed in Table 4.3 and shown, without relationships, in Figure 4.3.

Table 4.3. Classes contained within ASCII Artist Example

Class	**Description**
ASCIIArtist	Main class
Information	Project information (version, copyright, etc)
Grid	Two-dimensional Grid for drawing and projection
GridDraw	Interface for classes able to draw onto a Grid
Shape	Abstract base class for shapes
Triangle, Circle, Star, Square	Shape classes for the respective shape
TriangleFactory, CircleFactory, StarFactory, SquareFactory	Factory classes to create instances of the respective shape classes
PrintMethod	Abstract base class for printing (output) methods
CSVPrint, FilePrint, ScreenPrint	Printing methods
PrintException	Exception class for printing/output error
ShapeFactory	Abstract base class for shape factories

Using standard static reverse engineering techniques (Astah Professional ChangeVision [2016]) relationships can be recovered from source code including inheritance/generalisation and composition, as shown in Figure 4.4. For example the derived shapes can be seen from the Shape class and that Shape has a constituent Grid and applies the GridDraw interface. In total 18 relationships are found, in four cases there are two types of relationships (inheritance and composition) between the same classes.

The following step is to detect semantic repository relationships. Starting with the raw commit log (Figure 4.2 step 1; equation 4.2) for a given window generating a set of commits (steps 2 and 3; equations 4.3 and 4.4). These are then used to identify contained pair classes (step 4; equation 4.6), which are

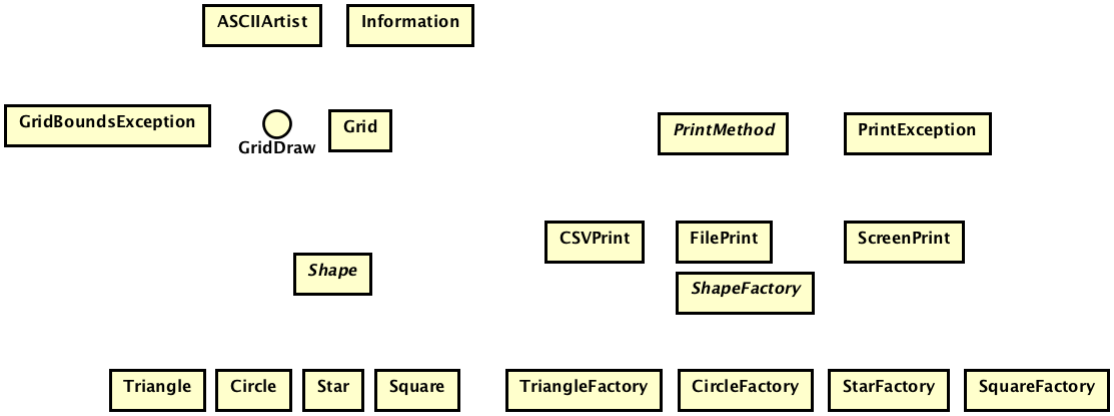


Figure 4.3. ASCII Artist Illustrative Example: Classes Only

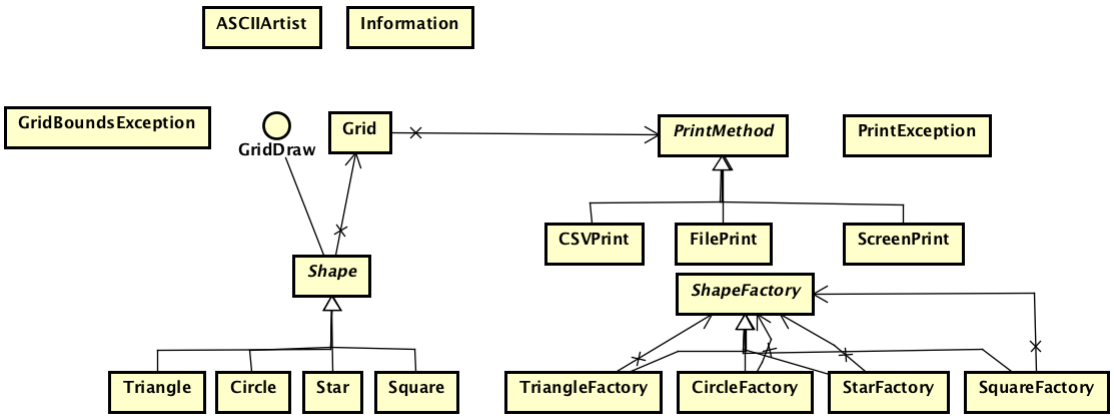


Figure 4.4. ASCII Artist Illustrative Example: Classes with Static (Reverse Engineering) Relationships

added together (step 5) to build a set of numerical relationships (step 6; equation 4.7). An example of these steps with figures can be seen in Section 4.2.1 and the process is outline in pseudo-code in Listing 4.1.

After this process one may, for example, find the results shown in Figure 4.5, a set of five relationships of which two are novel (not contained in the reverse engineering; highlighted in red). Because the source of data (number of co-committals) provides a relative value for each relationship, a significance value between 0 and 1 can be assigned to each relationship in turn (equations 4.8, 4.9, and 4.11).

Listing 4.1. Process Pseudo-code

```
Initialise pairs to zero
Initialise maximum to zero

Open Commit_Log CL

For each commit C in CL
  If C.Time is within Window.Start.Time and Window.End.Time
    For each component X in C
      For each component Y in C
        If X is not Y
          If pair(X,Y) exists
            Increment pair(X,Y) by 1
          else
            Create pair(X,Y) equal to 1
          endif
          If pair(X,Y) value is greater than maximum
            Set maximum to pair(X,Y) value
          endif
        endif
      endfor
    endfor
  endif
endfor

Close Commit_Log CL

For each pair P
  Divide P by maximum value
endfor
```

Having detected classes and relationships from each methods independently, they can be brought together and combined. As the relationships found from static reverse engineering are definite, existing solidly in the code, it can be assumed they are definitely correct e.g. have a significance value of 1. That is they are the *ground truth*. Relationships contained in the mining data, after filtering for significance above a threshold, are then combined with the static reverse engineering set. The resultant class and relationships example is shown in Figure 4.6, with the co-detected (found in both methods) relationships in green, reverse engineering only in black, and those found only in source code mining in red.

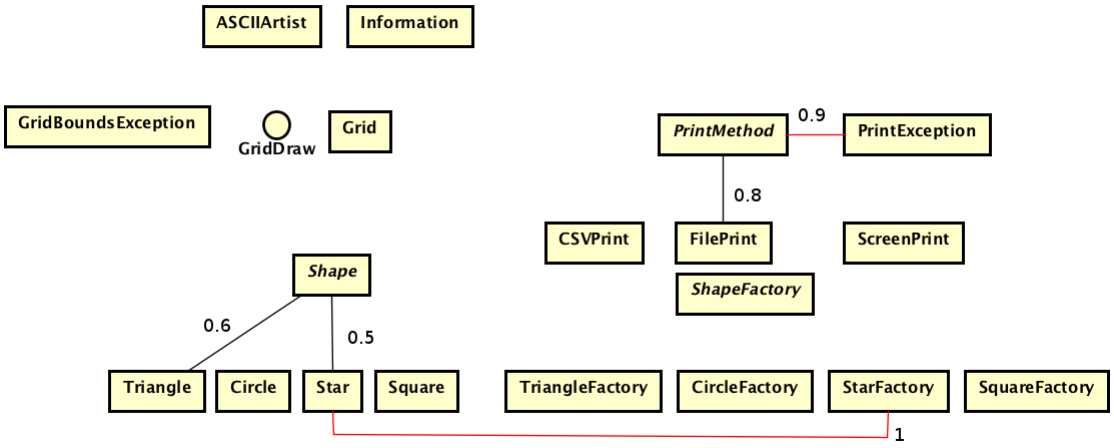


Figure 4.5. ASCII Artist Illustrative Example: Classes with Repository Mining Semantic Relationships (Novel Relationships in Red) and Confidence Values

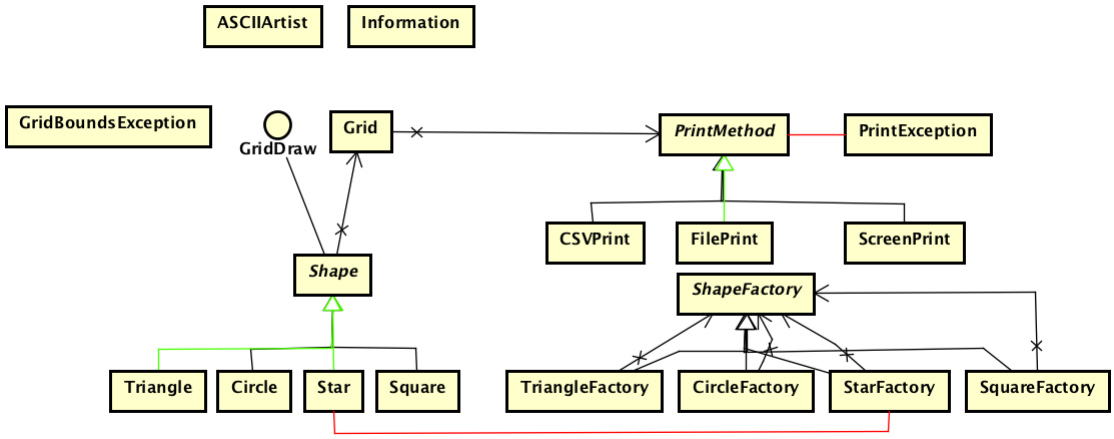


Figure 4.6. ASCII Artist Illustrative Example: Classes with Both Relationship Sets (Duplication in Green, Repository only in Red)

4.3 Implementation

To implement the approach three main components were required; programmatic access to reverse engineering output, co-commital data from a source code repository, and an analysis engine to work with both these inputs.

Access to reverse engineering output is provided through the use of a generic XMI parsing component. This component facilitates the loading of different forms

of XMI into a consistent abstracted memory model based on the benchmarking work (Chapter 3 and Cutting and Noppen [2014]). This can then be converted into a set containing relationship links for all the contained classes.

The committal information from repository logs is gathered through a mining component. The initial raw data gathering function of which was inspired by and based on re-engineering of the IBDOOS tool, generously provided by the Ecole Polytechnique de Montreal [Ali *et al.*, 2013].

Repository data is very noisy and so the mining component includes a number of configurable pre-filters to identify which commits should, or should not, be included in the data. For example commits which include a very high proportion of the overall class count are likely to be trivial changes, such as notices, or affect such a large number of files as to be meaningless. Filters based on structure and packages are supported to remove non-code artefacts from the data. Logic is also used to rename files into classes, creating a format comparable in content to that from the reverse engineering output, allowing comparison and analysis. An example of the type of information for a commit captured by one popular repository manager (git) that can be recovered is as shown in Listing 4.2.

Listing 4.2. Git Commit Log

```
commit 95fcedd248107ab5de29ec28d4477b6e0cba8ce9
Author:     davidcutting <David.Cutting@uea.ac.uk>
AuthorDate: Mon Aug 12 16:57:28 2013 +0100
Commit:     davidcutting <David.Cutting@uea.ac.uk>
CommitDate: Mon Aug 12 16:57:28 2013 +0100

    Association support for XMI (rhapsody tested)

M       src/org/purplepixie/xmi/ModelRelationship.java
M       src/org/purplepixie/xmi/XMIFile.java
M       src/org/purplepixie/xmi/XMIParserGenericSAX.java
```

4.3.1 Information from Repository Mining

Having implemented software to mine relationships from source code repositories, before moving into combinational analysis with reverse engineering, it was important to check that additional semantic links were indeed contained. This was accomplished in two ways. Firstly manual checking of relationships recovered and secondly projection by network analysis which allowed the visualisation of larger projects in an efficient and easily understood manner.

The manual analysis consisted of checking through detected relationships and making a judgement on the efficacy of the relationships. Owing to the size of the projects analysed this could only be done on a small basis of random sampling. The sampling did, however, show apparently sensible and valid relationships.

Network analysis is a system whereby an attempt is made to build a visual representation of a relationship network between nodes. This uses a similarity matrix based upon how similar or related nodes are; the number of interactions or commonalities they share. This was performed, using third-party tools in R, against the repository information mined for two projects. Figure 4.7 shows an annotated (highlighted and named for clarity) network analysis of the D-UEA-ST project showing closely grouped components. These, on further analysis, represent discrete sections of the software, either built by different developers or part of a common function. In addition some interface structures occur at the bottom right, showing a form of abstraction with interfaces and their concrete implementations.

An annotated network analysis of the XMI Analyser software (Figure 4.8) shows not only levels of interface from the central main class (GUI, controllers and data classes). The duplication left to right is the result of a refactoring and renaming which, from a repository perspective, resulted in the appearance of two distinct sets of files and an interesting illustration of repository data. The right-hand side of the image contained the original classes. The packages had then been changed to bring the project into a common namespace with other tools resulting in the apparent twin sets. At the same time as the rename, refactoring was completed to increase software quality; reducing coupling and an increased use of abstraction and layers. This can be seen in the left-hand side set of Figure

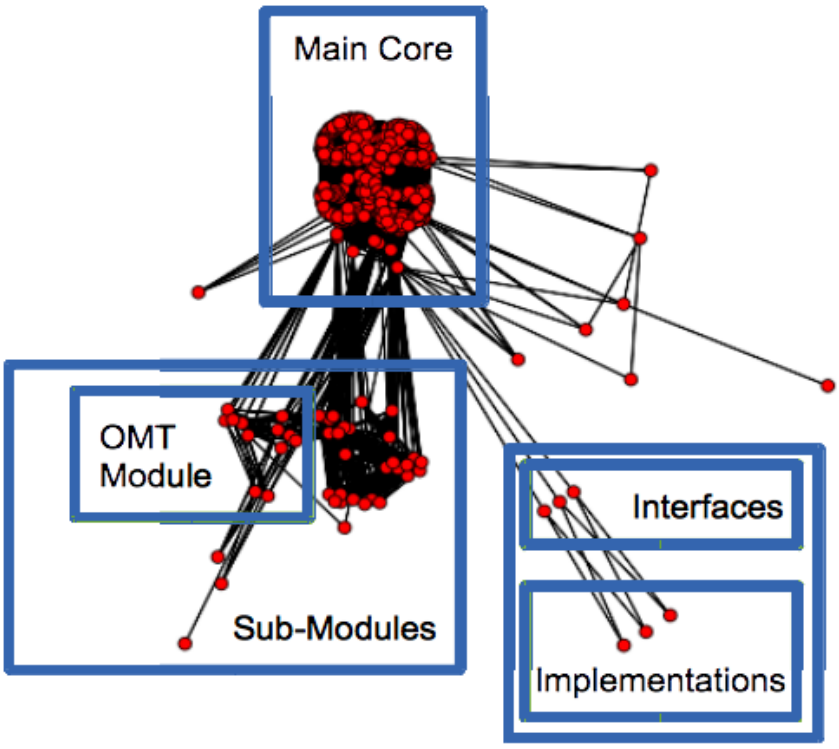


Figure 4.7. Annotated Network Analysis Output in R for D-UEA-ST Commit Data

4.8 compared to the right, with three distinct layers and lines of communication.

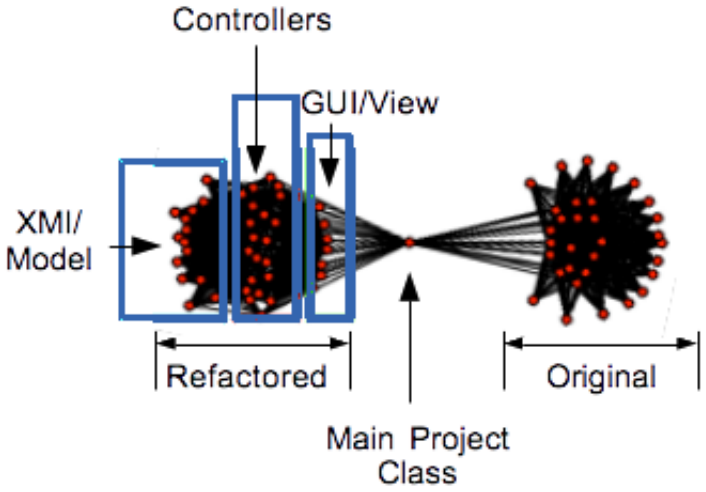


Figure 4.8. Annotated Network Analysis Output in R for XMI Analyser Commit Data

Owing to the computational complexity of building network diagrams it was infeasible to attempt this on larger projects. Consequently a simpler visualisation method was also used. Multi-dimensional scaling (MDS) was performed in R on the same similarity matrices (Section 2.5.1) to produce a two-dimensional representation of constituent classes. Two dimensions were chosen for output as they could be easily represented as an image without complexity and were in the same dimensions as the existing network diagrams. This was initially performed on the same projects, D-UEA-ST (Figure 4.9) and XMI Analyser (Figure 4.10), for comparison.

Each component (class) is represented by a mark (circle) on the figure, and the closer components are to each other the more strongly they are related. It appears some relationships continue to be visible with closely-located elements (possibly suitable for clustering - see Section 4.4), but this is not presented as clearly as with the network analysis because no relationship links are shown and many of the markers overlay with each other. For clarity some grouped elements are annotated in Figure 4.9 (much as in Figure 4.7) where the core class set and the OMT sub-module classes can be clearly identified as discrete closely grouped classes.

The MDS projection of XMI Analyser (Figure 4.10) shows some close location of classes, but mainly overdrawn on top of each other and so not as clearly decomposable as D-UEA-ST.

Because MDS is much less computationally expensive than network diagrams, large projects could also be analysed, such as the Eclipse project shown in Figure 4.11, containing over 16,000 classes. Owing to the sheer scale of Eclipse no attempt has yet been made to segment and analyse this projection. Again some semantic constructs are clearly visible through the close positioning and arrangement of components, certainly a core corpus and some discrete offshoot elements.

The groupings of components, shown in both the network diagram and MDS, especially showing the package rename so clearly, combined with the manual analysis, shows there is clear and useful information contained within the repository data. This data therefore, represented as sets of relationship strengths, does have the potential to be used in combination with other information and give fuller

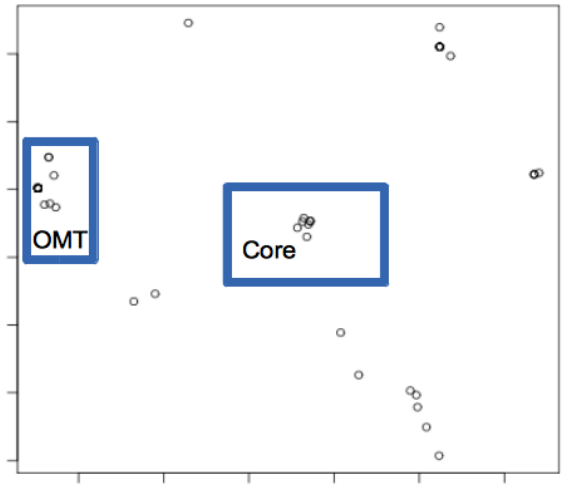


Figure 4.9. Annotated MDS Output in R for D-UEA-ST Commit Data

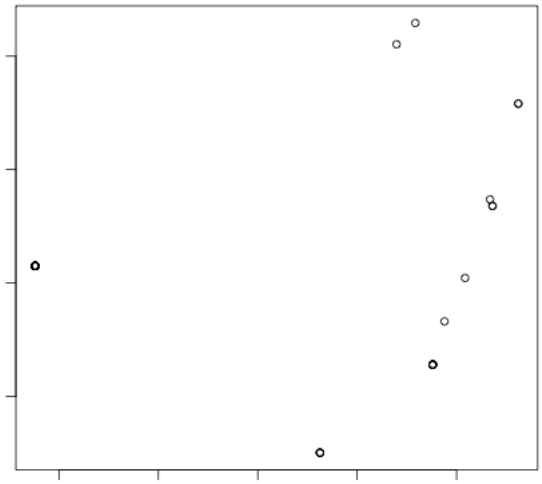


Figure 4.10. MDS Output in R for XMI Analyser Commit Data

insight or validation.

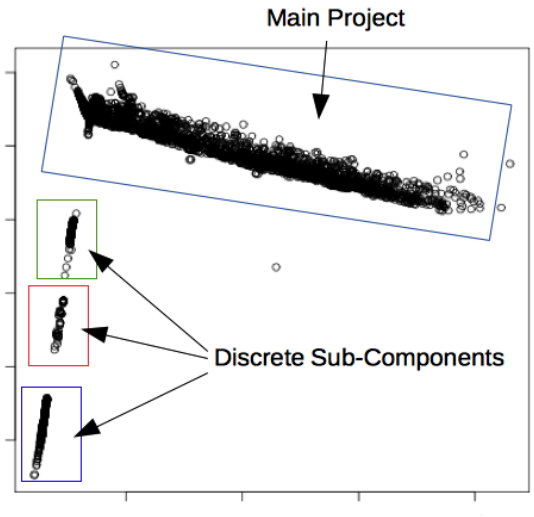


Figure 4.11. MDS Output in R for Eclipse Commit Data, augmented to show some architectural components

4.4 Application to Clustering

One potential application of recovered rich relationship information is the clustering of related elements or components within software. At the simplest level this will enable related components to be easily distinguished but also has the possibility of establishing additional information such as parts of architectural styles or semantic constructs (e.g. model-view-controller). To explore this possibility a series of experiments were conducted to apply the relationship data to clustering.

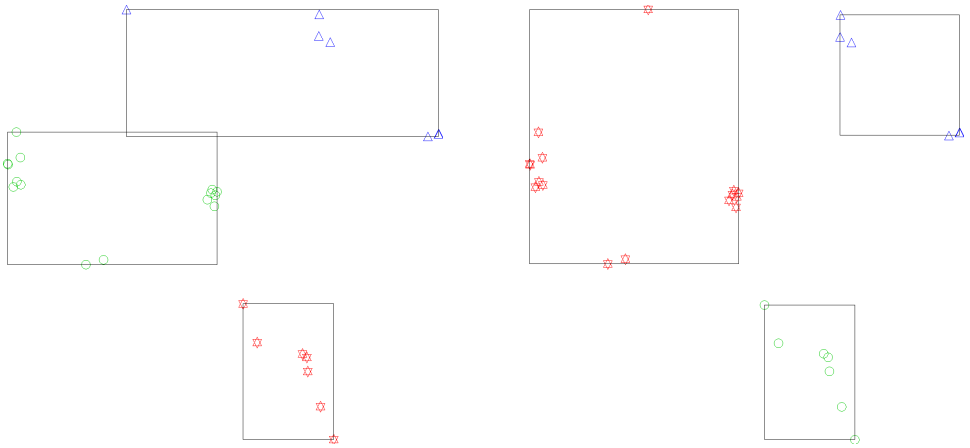
Having shown, using R, that relationships do seem to be present in the commit data (Section 4.3.1) and that multi-dimensional scaling (MDS) can be used to aid visualisation, MDS was then implemented in Java. Using data scaled to two dimensions with MDS offers benefits beyond n-dimensional data, primarily in terms of visualisation and comprehensible output which can be validated. Implementation in Java rather than pipelining into R is desirable to reduce requirements and dependencies of any resulting analysis application as well as providing this processed data internally for further use without reading it back in. A Java library is available for MDS processing [Pilch, 2009] and this was integrated within the experiment application. To validate output from the Java MDS it was compared

to the R output previously created and found to match.

With the data from the relationship detection leading to a two-dimensional MDS projection, the ability to perform clustering was added to the codebase. Clustering based upon repository patterns has previously been shown to be both possible and useful in showing relationships beyond static reverse engineering [Beyer and Noack, 2005; Bieman *et al.*, 2003].

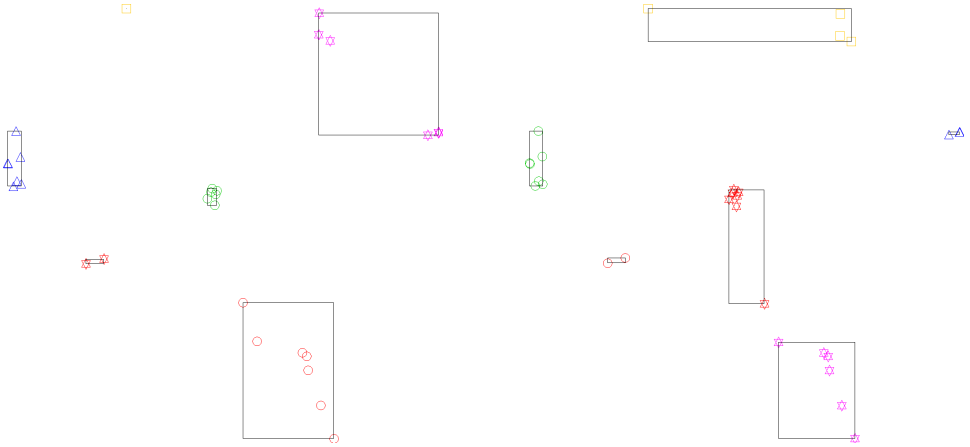
Initially K-Means (an approach with a defined number of randomly positioned centroids for K clusters) was implemented as an initial proof-of-concept to check the MDS projections and also to see how applicable this clustering technique would be to software similarity data. As identified in the literature, K-Means can be highly variable in output and not always suited to all data types. Although the software did prove capable of clustering as expected, a high level of uncertainty was seen in the output clusters. This was prevalent at relatively low levels of K, such as in Figure 4.12 where K=3, but clusters failed to stabilise at higher values of K either, as seen in Figure 4.13 for K=6. As a clear visualisation for each incrementing value of K a series of 100 runs was performed and put into a 15 frame-per-second video to allow easy time-based visualisation of stability (an example of these video projections can be found in Cutting [2016]). Not only was K-Means found to be unstable but even at K=6, the groupings of components into clusters were rarely accurate compared with, for example, the manual analysis shown in Figure 4.9.

As K-Means was found to be inconsistent, rather than expending effort trying to improve this technique, such as by smart centroid identification or averaging cluster runs, an alternative technique of the Expectation Maximisation/Gaussian Mixture Model (EM/GMM) was implemented. To implement the EM/GMM clustering approach WEKA was integrated into the experiment and it's EM methods were used. The WEKA/EM strategy class converted the supplied two-dimensional data into a form suitable for WEKA, ran the EM algorithm with set parameters, and returned the cluster assignments. The WEKA/EM process returned more stable clusters (and without a need to pre-specify the number to be found) but needs more investigation as to the most appropriate settings (see Section 7.2) as some inconsistencies and unexpected output can be seen. The stability relative to K-Means can be seen in Figure 4.14 and a much larger sam-



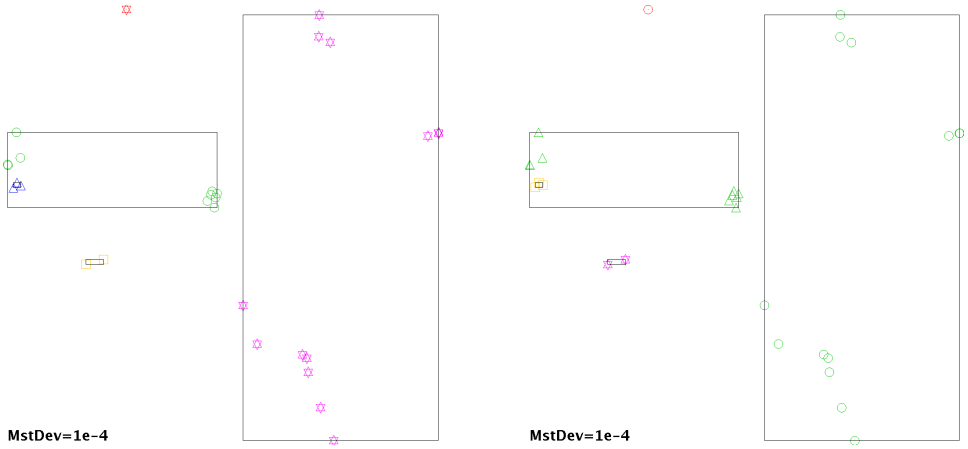
(a) K-Means with K=3 (A) (b) K-Means with K=3 (B)

Figure 4.12. K-Means Clustering of D-UEA-ST Co-Commit Data with K=3: Each point (class) is put into one of 3 clusters but the results are highly unstable i.e. for exactly the same input parameters the left and right images were produced owing to the initial randomisation of K centroids



(a) K-Means with K=6 (A) (b) K-Means with K=6 (B)

Figure 4.13. K-Means Clustering D-UEA-ST Co-Commit Data with K=6: Each point (class) is put into one of 6 clusters with results more stable than K=3 (Figure 4.12) but still showing variation for the same input parameters



(a) EM Clustering (A) (b) EM Clustering (B)

Figure 4.14. EM Clustering of D-UEA-ST Co-Commit Data with Minimum Standard Deviation 1x10^{-4}: EM delivers stable consistent clustering but at a resolution (number of clusters) unsuitable for detailed analysis (e.g. see the right-hand cluster group)

ple (Eclipse) can be seen in Figure 4.15 highlighting an inconsistency issue with the central data point being included in a cluster along with other closely-related components.

4.5 Conclusion

Having demonstrated that reverse engineering alone can be imprecise, the possibility of mining source code repositories for useful additional information was investigated. The first step was to test the hypothesis that through the measurement of co-commits (components being changed at the same time), relationships between components could be established. This experiment showed that not only could co-committal data be successfully mined from repository meta data but that it also contained links and delineated groups of components (Section 4.3.1).

As an experimental proof-of-concept a theoretical worked example demonstrated how relationships could be detected and recovered from both static reverse engineering and repository mining (Section 4.2.3). This example introduced *confidence values*, numerical representations of the apparent strength of a relationship in relation to others, from repository data and the concept of treating

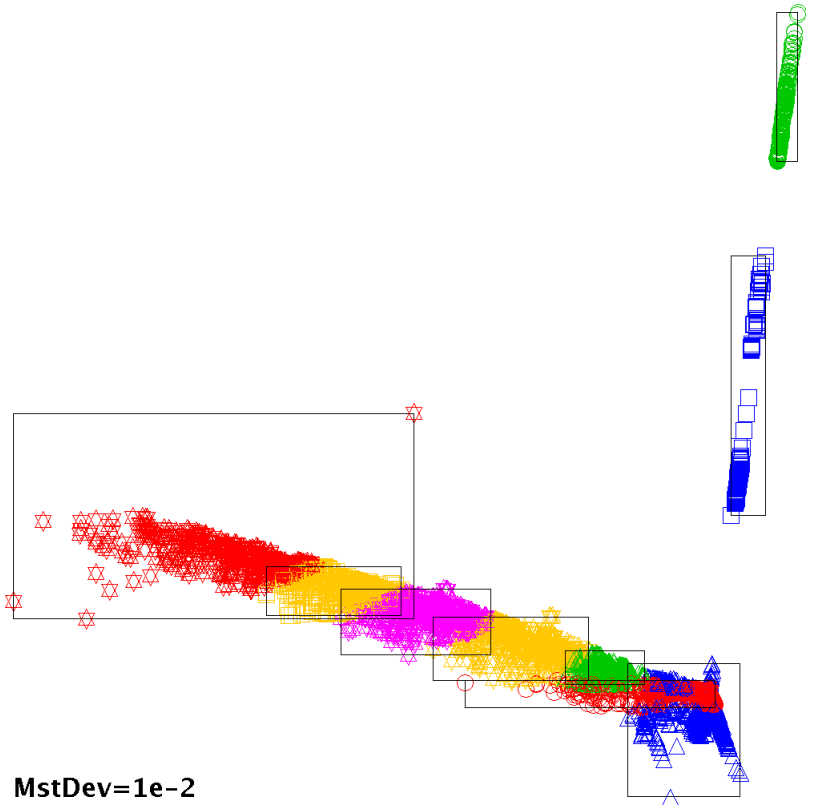


Figure 4.15. EM Clustering of Eclipse Co-Commit Data with Minimum Standard Deviation $1x10^{-2}$: Segmentation through clustering occurs, in a stable fashion, but with similar resolution issues to those seen in Figure 4.14

reverse engineering input as a *ground truth*. The two sources of information were then brought together to validate certain relationships and demonstrate others' are only provided by a single source.

This framework and worked example experiment forms a basis within which real world information from different information sources, on the same software target, can be brought together and used in conjunction, and demonstrates that this can be done consistently and effectively.

As an example application of these information sources clustering was examined in Section 4.4, using both K-Means and EM/GMM. Though limitations were found when applying these traditional methods, the overall conclusion can be that useful information could be generated and closely related components clustered together. Further work to consider different and more applicable forms

of clustering, perhaps fuzzy approaches in which nodes can belong to 0, 1 or more than 1 clusters, could be more successful. The application of such clustering to the software domain would be a major undertaking in itself and so not part of this chapter. This work has however found that clustering showed potential for the grouping of related components but the results were inconclusive with the techniques available.

Chapter 5

A Uniform Approach to Combining Multiple Traceability Information Sources

5.1 Introduction

In order to comprehend software structure and form, reverse engineering is often used to analyse code and generate output such as Unified Modelling Language (UML) diagrams [Arcelli *et al.*, 2005; Rasool and Streitfdert, 2011; Roscoe, 2011]. When benchmarking reverse engineering in Chapter 3, it was found that reverse engineering output can be imprecise and even when complete may only find structural components [Cutting and Noppen, 2014, 2015]. It is also the case that the large diagrams which can often result may be overly complex or cumbersome to use [Fowler and Highsmith, 2001; Grossman *et al.*, 2005]. However, there are also a number of alternative potential sources of understanding beyond reverse engineering.

One example of an alternative source of insight is through the mining of source code repositories, analysing the meta data logging the evolutionary changes of source code [Kagdi *et al.*, 2007b], as demonstrated in Chapter 4. Other possible sources include dynamic runtime analysis such as call tracing or stack analysis, expert opinion on related matters, other forms of static analysis such as coupling detection, and natural language processing of documentation or comments. The appropriateness and insight of each of these methods would naturally vary depending on the source project; perhaps with better documentation, structured in a more logical way for static analysis or with a higher granularity in the source code repository. Varied combinational use as applicable, and using weightings for importance of different sources, is therefore an essential part of the wider use of multiple information sources.

Whatever the sources of information chosen they must be compatible and comparable in order to be used successfully together in a common format. Failure to accomplish this would lead to two or more information sources suitable for independent use only.

5.1.1 Problem Statement

Is it possible to define an information framework to consider any information source in a uniform manner, to allow the comparative and combinational use.

5.1.2 Solution Definition

To allow combinational use the approach is to identify the *lowest common denominator* between information sources, a format that all information sources can be decomposed to and which still holds useful information.

The *lowest common denominator* is defined as: the lowest level of detail (highest level of granularity) which the information sources hold in common. For example source X has lowest level data at the class level whereas source Y has the statement level. The *lowest common denominator* in this case would be the class level (statements are more detailed than and can be decomposed upwards into classes).

In software terms this would be a model of component interaction. The question being: for a given level of granularity, which can be generated by all information sources, what is the count and significance of interactions found between components?

The granularity level represents the form of the components used, which may be files, classes, methods, or even variables or statements, whatever can be represented by all information sources. To offer an analogy in communication analysis; records might be held of phone calls and email exchanges which can be combined to build up a network analysis of interactions. The level of granularity in this example could be the individual, if both sources of information were personal enough, or could be the household as the landline phone records do not distinguish individuals. Only the highest common level of granularity could be used.

Although the approach presented is entirely generic, so open to any level of granularity, the implementation is based around the class level so all the components are classes constituent to a project.

In general the solution, as shown in Figure 5.1, is to take a series of different information sources pre-processed to be comparable, apply sets of filters and weighting before combining them. A separate set of "ground truth" relationships are also maintained and then merged into the output.

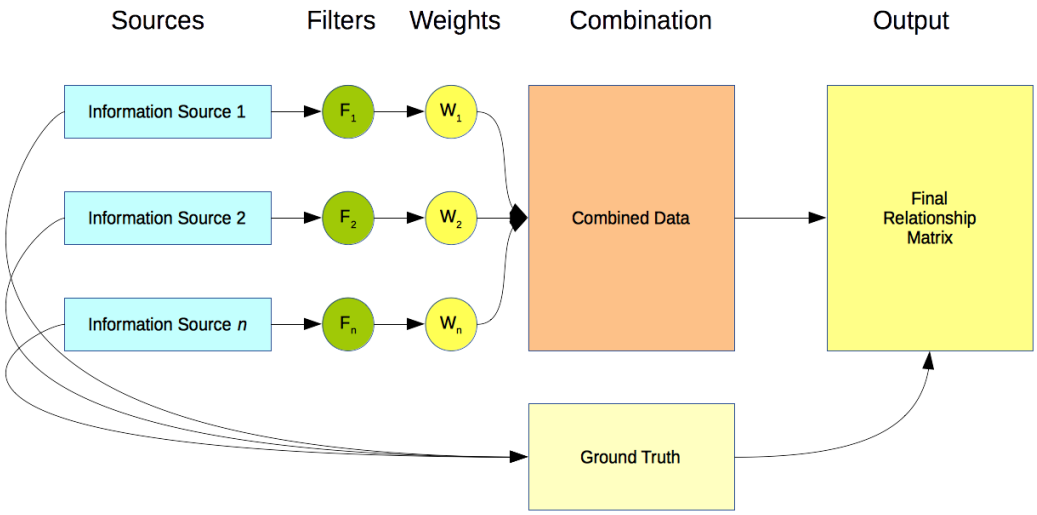


Figure 5.1. Process Overview of the Uniform Approach to Combining Multiple Traceability Information Sources

5.2 Approach

The approach to the combination of data sources is based around the concept of component interactions. For any given data source a set of interactions between each component in the target system can be obtained, a set S. Each intersection between components is represented by a value and possibly some additional information, forming a relationship matrix. In its simplest form each intersection is an integer value, initialised at 0, which is incremented every time an interaction between the two components is found. This is a uniform matrix that can be generated for any given information source, containing all the intersections of constituent components. An example of a 6-element relationship matrix is in Table 5.1, which would represent the example class diagram shown in Figure 5.2 where there is not yet any relational information at all between the classes. Intersections on the diagonal (between a component and itself) are excluded from the data as they are meaningless (a component is clearly related to itself in the strongest possible fashion). The matrices are completely symmetrical containing two intersections for each pair. For clarity only one half of the matrix is completed in the examples the other half mirroring the data.

Each information source will have its own relationship matrix of component

	A	B	C	D	E	F
A	-					
B	0	-				
C	0	0	-			
D	0	0	0	-		
E	0	0	0	0	-	
F	0	0	0	0	0	-

Table 5.1. Example 6 Element Relationship Matrix

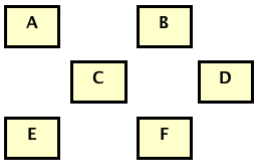


Figure 5.2. Example Six Element Class Diagram with No Relational Information

interactions, for example for two sources X and Y, matrices S_X and S_Y would be formed.

5.2.1 Formal Definition

Let the software system under study S consist of a number of components $\{c_1, c_2, ..., c_n\}$ between which the intent is to establish semantic relationships and their confidence value.

Let S_C be the Cartesian product of S with itself, i.e. S_C contains all the pairs of components.

$$S_C = S \times S = \{(c_1, c_1), (c_1, c_2), ...\} \tag{5.1}$$

R_X is defined as a similarity function for system S, information source X and pair of components $(c_x, c_y) \in S_C$. R_X calculates the confidence of the relationship (similarity) between c_x and c_y.

$$R_X : (c_x, c_y) \rightarrow [0..1] \tag{5.2}$$

under the following conditions:

- the output of the function is expressed as a number on the domain [0..1] where 0 means "*no confidence they are part of the same semantic concept*"

and 1 means "*certain they are part of the same semantic concept*"

- R_X must be commutative (i.e. $R_X(c_x, c_y) = R_X(c_y, c_x)$)

A unique definition of R_X needs to be defined for each information source describing S. Note that R_V (Section 4.2.2) is an example instance of R_X for version management systems as information sources and fulfils the conditions listed above.

AC is defined as a function outline for the aggregate confidence value of semantic relationship between two components c_x and c_y given a set of relationship functions R_X.

$$AC : (c_x, c_y, R_X) \rightarrow [0..1] \tag{5.3}$$

under the following conditions:

- the output of the function is expressed as a number on the domain [0..1] where 0 means "*no confidence they are part of the same semantic concept*" and 1 means "*certain they are part of the same semantic concept*"
- AC must be commutative
- AC must be associative

This function needs to be explicitly defined for a set of information sources and conform to the definitions given above. A suitable example definition for this function can be the use of the max operator to aggregate confidence values of semantic relationships. The example function below attaches the highest confidence value that can be found in R to the semantic relationship between c_x and c_y.

$$AC_{\max}(c_x, c_y, R) = \max\{r(c_x, c_y) | r \in R\} \tag{5.4}$$

5.2.2 Filtering

Without pre-judging the information sources chosen it is clear that some sources may be more or less sparse, or more or less noisy than others. It is important

therefore to build into the generic approach some consideration for filtering. Filtering can happen pre- or post-matrix.

Pre-matrix filtering is the decision about what interactions to include within the matrix. For example an information source may contain numerous types of interactions of widely varying significance; certain of these may be excluded as being insignificant and therefore the intersections are not incremented. This filtering happens within the realm of the information source data collection and happens before the resultant products are provided as a similarity function (equation 5.2).

Post-matrix filtering is setting parameters by which intersections are removed or deemed insignificant so not taken forward into the analysis. This can be accomplished, for example, by the application of a naive significance filter; any values below a threshold are set to 0 and so discarded. More complex filtering based on the distribution of values throughout the matrix, or external considerations, is also possible.

Therefore a general filter function F can be defined that for all the pairs of components c_x, c_y removes (sets to 0) those pairs which do not meet a significance value n:

$$\begin{aligned} &F((c_x, c_y), n) = R_X(c_x, c_y), && if \;\; R_X(c_x, c_y) >= n \\ &F((c_x, c_y), n) = 0, && otherwise \end{aligned} \tag{5.5}$$

5.2.3 Weighting and Truth

All information sources are not created equally. Some will naturally be more significant, more reliable, or have a higher fidelity to the actual nature of component relationships. Some may also be classed as *completely reliable* e.g. they can be taken as a *positive ground truth*; if they show a relationship between components that relationship is said to definitely exist in the strongest possible sense.

Consider for example a static code analysis (static reverse engineering) which identifies an inheritance (generalisation) relationship between two component classes. That this relationship exists is clear and the strength of that relationship, one class being derived from the other, is also clear and very strong. Compared to some identification of a relationship from another information source, such as call traces or semantic analysis of the repository, it is not only a much stronger

but a *definite* relationship. In this case the static relationship could be considered a *ground truth* and other relationships indicative at best.

To allow for this disparity in information sources the approach allows for both weighting values when combining matrices (one information source is of a greater weight and therefore higher importance) and the inclusion of additional information such as *ground truth* flags for intersections (for which there is said to definitely be a relationship regardless of resultant value following amalgamation of matrices).

To record such ground truth another instance of the Cartesian product of S with itself (S_C, equation 5.1) is used, thereby containing all the pairs of components with values representing 1 for *ground truth relationship* and 0 where no such relationship is said to exist.

$$S_T = S \times S \tag{5.6}$$

A truth function T is defined that for a given pair of components (c_x, c_y) and information source X indicates if a *ground truth* relationship exists. In an information source designated as a *ground truth* any relationship value above zero would be regarded as true:

$$\begin{aligned} T((c_x, c_y), X) &= 1, \qquad if \;\; R_X(c_x, c_y) > 0 \\ T((c_x, c_y), X) &= 0, \qquad otherwise \end{aligned} \tag{5.7}$$

under the following conditions:

- Information source X is a *ground truth* source

Combining Weighted Sources

To combine different information sources into a single set (the combined relationship strength over all sources), the function TO is defined that for pairs of components (c_x, c_y) aggregates the information source relationships for each information source X in accordance with weighting value w relative for each source,

within the domain [0..1].

$$TO(c_x, c_y) = \frac{\sum_{i=1}^{n} w_i R_{X_i}(c_x, c_y)}{\sum_{i=1}^{n} w_i} \tag{5.8}$$

In order to combine the truth data a further two functions are defined which together combine the sets of truth relationships for each information source. First the function TC which sets the value to be carried forward.

$$\begin{aligned} &TC((c_x, c_y), X) = 1, \qquad if\ \ T((c_x, c_y), X) > 0 \\ &TC((c_x, c_y), X) = 0, \qquad otherwise \end{aligned} \tag{5.9}$$

And then the function TR which combines the sources

$$TR(c_x, c_y) = \sum_{i=1}^{n} TC((c_x, c_y), X_i) \tag{5.10}$$

5.2.4 Boolean Conversion

For some purposes a discrete level of relationship strength is not required, merely that if the relationship exists (has passed the various filtering and combination threshold) it is recorded as present or not. This is represented in a Cartesian product of the classes but containing just 1 if a relationship is present or 0 if a relationship is not.

Functionally in this definition the function is the same as that shown in equation 5.7, but the purposes of the two are different and may be rewritten independently and so are kept as separate functions. Equation 5.7 is intended to generate a new matrix set containing boolean values for a data source *we know to be true* i.e. the ground truth. The resultant matrix can then be used to preserve relationships represented through later filtering stages. The standard Boolean conversion however is applied to any matrix and converts any non-zero values to 1. Let the function B to produce a Boolean result 1/0 for a pair of components (c_x, c_y). This can be used with any suitable source (such as an information source

R_X, but here is used with a combined total Tr).

$$\begin{aligned} B(c_x, c_y) &= 1, \qquad if\ \ Tr(c_x, c_y) > 0 \\ B(c_x, c_y) &= 0, \qquad otherwise \end{aligned} \tag{5.11}$$

5.2.5 Combination and Simple Example

Once the matrices for each information source have been generated and filtered, they can be used together through the formulae defined in Sections 5.2.1 to 5.2.4. The combinational function To (equation 5.8), with associated weighting values w, allows the formation of a combinational set. For two information sources A and B the relationship matrices would be S_A and S_B. An example of this process is shown in Table 5.2. Pseudo-code outlining the processes following to manipulate the data in Table 5.2 is contained in Listing 5.1.

	A	B	C	D	E	F
A	-					
B	1	-				
C	1	0	-			
D	0	0	0	-		
E	0	0	0	2	-	
F	0	0	0	1	0	-

(a) X Relationship Matrix (S_X)

	A	B	C	D	E	F
A	-					
B	0.5	-				
C	0.5	0	-			
D	0	0	0	-		
E	0	0	0	1	-	
F	0	0	0	0.5	0	-

(b) Normalised X Matrix (from R_X)

	A	B	C	D	E	F
A	-					
B	4	-				
C	6	0	-			
D	0	1	0	-		
E	0	1	0	10	-	
F	0	4	0	0	0	-

(c) Y Relationship Matrix (S_Y)

	A	B	C	D	E	F
A	-					
B	0.4	-				
C	0.6	0	-			
D	0	0.1	0	-		
E	0	0.1	0	1.0	-	
F	0	0.4	0	0	0	-

(d) Normalised Y Relationship Matrix (from R_Y)

	A	B	C	D	E	F
A	-					
B	0.4	-				
C	0.6	0	-			
D	0	0	0	-		
E	0	0	0	1.0	-	
F	0	0.4	0	0	0	-

(e) Normalised Y Relationship Matrix (S_Y), with Significance Filter Applied; $R_Y = F(R_Y, 0.2)$

	A	B	C	D	E	F
A	-					
B	0.9	-				
C	1.1	0	-			
D	0	0	0	-		
E	0	0	0	2.0	-	
F	0	0.4	0	0.5	0	-

(f) Combined Matrix; $To = \{S_X, S_Y\}$

	A	B	C	D	E	F
A	-					
B	1	-				
C	1	0	-			
D	0	0	0	-		
E	0	0	0	1	-	
F	0	0	0	1	0	-

(g) Boolean X Matrix $B(S_X)$

	A	B	C	D	E	F
A	-					
B	1	-				
C	1	0	-			
D	0	0	0	-		
E	0	0	0	1	-	
F	0	1	0	0	0	-

(h) Boolean Y Matrix $B(S_Y)$

	A	B	C	D	E	F
A	-					
B	0	-				
C	0	0	-			
D	0	0	0	-		
E	0	0	0	0	-	
F	0	1	0	0	0	-

(i) B Source Only Relationships Matrix $S_{UY} = |B(S_Y) - B(S_X)|$

Table 5.2. Data Source Matrix Creation and Combination Process for Example Sources X and Y

Listing 5.1. Pseudo-code Matrix Example

```
// Panel a
Load matrix X

// Panel b
For each i in X
  i = i divided by maximum value in X
endfor

// Panel c
Load matrix Y

// Panel d
For each i in Y
  i = i divided by maximum value in Y
endfor

// Panel e
For each i in Y
  if i value is less than 0.2
    i value set to 0
  endif
endfor

// Panel f
New matrix Z initialised as 0
For each i in X
  Z i value incremented by X i value
endfor
For each i in Y
  Z i value incremented by Y i value
endfor

// Panel g
For each i in X
  If i value greater than 0
    i value set to 1
  endif
endfor

// Panel h
For each i in Y
  If i value greater than 0
    i value set to 1
  endif
endfor

// Panel i
New matrix W initialised as copy of Y
```

```
For each i in Z
  If i value greater than 0
    W i value set to 0
  endif
endfor
```

In this example two information sources A and B are provided as matrices before being normalised to the range 0..1. A naive significance filter (F) is applied to S_B before the two sources are combined.

In this instance the combination of matrices is made with equal weighting ($w_i = 1$) and so the values are just added together for each intersection (and the combined matrix can be re-normalised as required).

Generally it is foreseen that there would be significant overlap between information sources, e.g. the majority of detected relationships would be found in multiple sources. To demonstrate the efficacy and value of a single source, it is important to be able to identify relationships found from it alone. To demonstrate this the two example matrices S_A and S_B are converted into *boolean matrices* with function B for simplicity, where each intersection is represented by 0 for no relationship and 1 for some form of relationship. By subtracting one from the other, just the relationships contained in a single form can be found, in this case those found by information source B represented as $S_{UB} = |B(S_B) - B(S_A)|$.

5.2.6 Illustrative Example

Returning to the illustrative example in Chapter 4 (Section 4.2.3) used to demonstrate incorporation of reverse engineering data, a set of matrices can be generated.

From the reverse engineering data, extracted from XMI, the matrix S_R is formed as shown in Table 5.3. Similarly the matrix S_M is formed from the semantic relationships found through repository metadata mining, shown in Table 5.4.

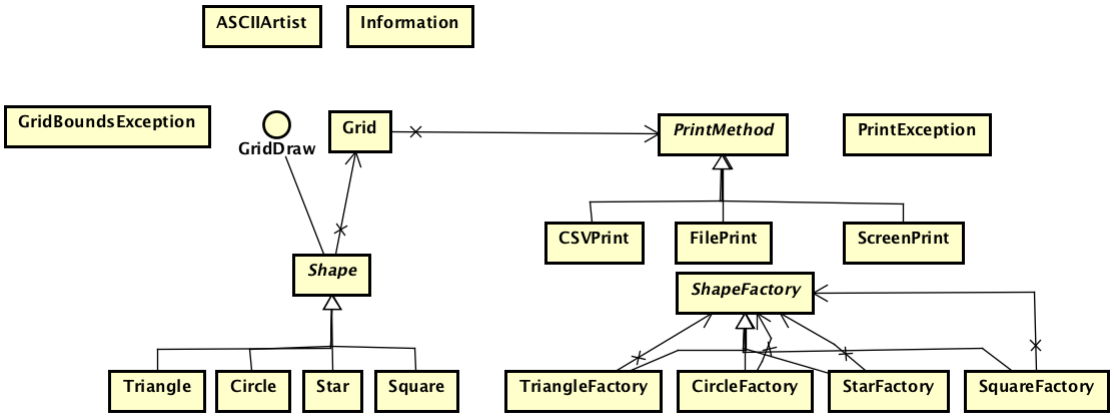


Figure 5.3. ASCII Artist Illustrative Example: Classes with Static (Reverse Engineering) Relationships

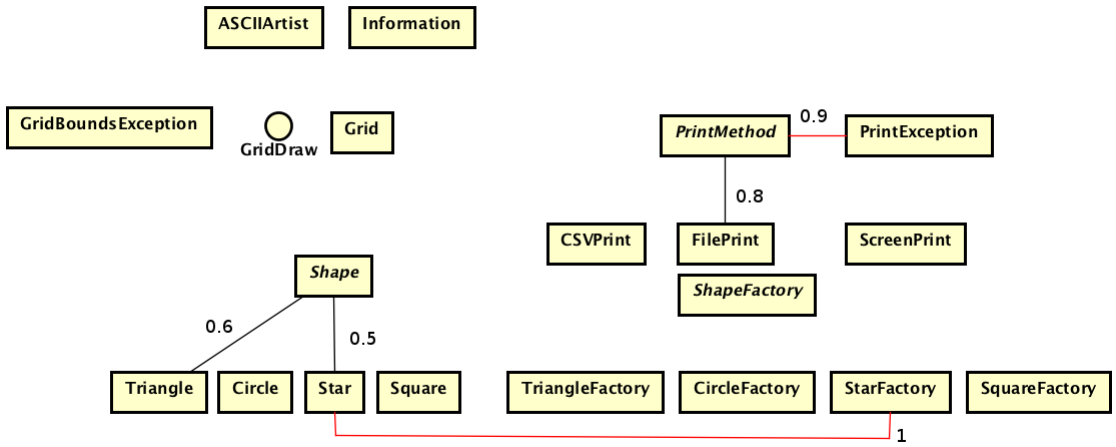


Figure 5.4. ASCII Artist Illustrative Example: Classes with Repository Mining Semantic Relationships (Novel Relationships in Red) and Confidence Values

The two matrices S_M and S_R can be combined through the + operator to include all relationships found, where $R_{MR} = S_M + S_R$, as shown in Table 5.5.

5.3 Discussion and Limitations

To make regular and replicable use of multiple information sources in a comparable form it is clear that a standard framework is required. This has been defined,

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	19
3 Grid	-															
4 GridDraw		-														
5 Shape	1	1	-													
6 Triangle			1	-												
7 Circle			1		-											
8 Star				1		-										
9 Square			1				-									
10 TriangleFactory								-								
11 CircleFactory									-							
12 StarFactory										-						
13 SquareFactory											-					
14 PrintMethod	1											-				
15 CSVPrint												1	-			
16 FilePrint												1		-		
17 ScreenPrint												1			-	
19 ShapeFactory								2	2	2	2					-

Table 5.3. ASCII Artist Illustrative Example: Reverse Engineering Matrix S_R of Figure 5.3 (components 1,2, and 18 with no relationships removed for clarity)

	5	6	8	12	14	16	18
5 Shape	-						
6 Triangle	0.6	-					
8 Star	0.5		-				
12 StarFactory			1	-			
14 PrintMethod					-		
16 FilePrint					0.9	-	
18 PrintException					0.9		-

Table 5.4. ASCII Artist Illustrative Example: Repository Mining Matrix S_M of Figure 5.4 (components 1, 2, 3, 4, 7, 9, 10, 11, 13, 15, 17, and 19 with no relationships removed for clarity)

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
3 Grid	-																
4 GridDraw		-															
5 Shape	1	1	-														
6 Triangle			1.6	-													
7 Circle			1		-												
8 Star				1.5		-											
9 Square			1				-										
10 TriangleFactory								-									
11 CircleFactory									-								
12 StarFactory						1				-							
13 SquareFactory											-						
14 PrintMethod	1											-					
15 CSVPrint												1	-				
16 FilePrint												1.8		-			
17 ScreenPrint												1			-		
18 PrintException												0.9				-	
19 ShapeFactory								2	2	2	2						-

Table 5.5. ASCII Artist Illustrative Example: Combined Matrix $R_{RM} = S_R + S_M$ of Figure 5.5 (components 1 and 2 with no relationships removed for clarity)

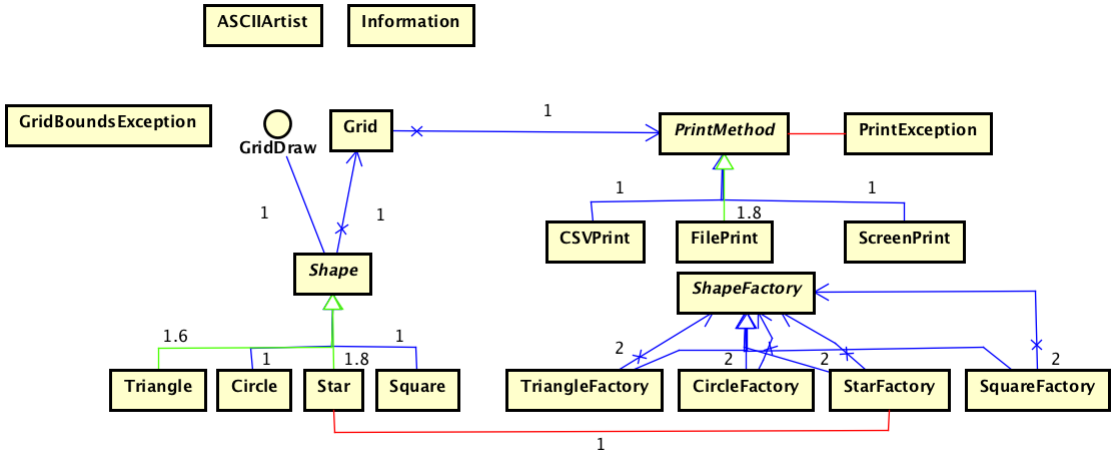


Figure 5.5. ASCII Artist Worked Example: Classes with Both Relationship Sets and Strength Values (Static only in blue, repository only in red, found in both in green)

with examples, in this chapter. Although clearly suitable to meet the basis of the problem statement, which is worded in a general manner, there exist some limitations to the approach. These are discussed below.

5.3.1 Level of Granularity

For comparison all the information sources must provide data at a common level of granularity, referred to as the *lowest common denominator* e.g. the most specific level of granularity shared by all sources. For example, a source with information at the method level cannot be directly used with a source at the class level; it must be converted to a higher comparable level. In this example, methods being contained within classes, relationships can be easily converted from the method to the class level (A::x – B::y becomes a A – B relationship).

This necessitates both a loss of precision and also a loss of specific accuracy; it may be the case that the level of granularity becomes *so high* that the entire process becomes unwieldy or less useful. For example if a source operates at the architectural level, all other sources are limited to this level.

One possible solution to this would be to "devolve" high sources to the desired level of granularity, using weighted means or another conversion, e.g. a relationship between elements A – B would "devolve" into a less weighted relationship

evenly spread between A::x – B::y and A::z – B::w.

5.3.2 Pre-Matrix Filtering

The filters included in the approach are limited to those applied to the generalised data once presented as sets of relationships (e.g. a significance filter). Different information sources would themselves require significant processing to be brought (a) into the set form for comparison and (b) to potentially filter out "noise" at the raw data level.

For example, when considering converting a source code repository history into a set of relationships; the starting point is a list of commits and their constituent changed elements. These are then processed and relationships are strengthened when both constituent parts of the pair are contained within a commit. However not all commits would want to be included within the analysis, e.g. those in which a large proportion of overall classes are updated as to be meaningless, or those to outdated historic files or structures which have since been removed from the project. Likewise, how would renaming of files be dealt with? Are relationships with the previously named file ignored or "merged" into the relationships with the current names?

Such source-specific filtering and processing is outside the terms of the generic approach detailed in the chapter. For full repeatability some generalised forms of pre-processing could be defined, certainly for common types or sources of relationship information.

5.3.3 Maintaining Traceability of Relationships

In the generic approach data sources are used in *weighted combination* e.g. a relative strength or importance is applied to each source and then are combined. Further the use of *ground truth* means that relationships determined to be *definitively true* are kept protected even if other sources are combined which do not show such a relationship and would naturally weaken its strength.

Using the pure form of the approach however means that the source of the relationship and strength are lost. After a series of combinations there is no record

at the final stage of where each came from. The original data sets from the individual sources could, of course, be referred back to. However, in an ideal scenario a "traceability matrix" would be maintained alongside the "relationship matrix" and "ground truth matrix", containing a narrative history of each relationship set. This would allow for the generation of "rationale reports" detailing exactly why the approach determined such a relationship exists. This in turn could be used with expert knowledge to determine the credence to put into such a relationship, perhaps with reference to a specific use.

5.4 Conclusion

In this chapter a generic model for the comparison and combination of disparate data sources through the use of *lowest common denominator* relationship matrices has been introduced. Once, after appropriate filtering, a matrix is created from a given data source it can be used in isolation or in combination with other matrices targeting the same subject (with the same set of intersections).

Owing to the heterogeneous nature of these disparate sources, post-matrix filters and weightings can be applied, as well as a *ground truth* nominated. The matrices can then be mathematically combined or subtracted to provide a full set of found relationships or only those uniquely found in a single source respectively.

After the creation of a richer combined information matrix for target software, this data can be potentially used in a variety of ways. As a simple set of relationships it allows easier understanding of couplings or dependencies within a code corpus, for general or specific use. The clearest specific potential use is for change impact analysis, identifying which components are closely linked and therefore likely to be impacted by changes made. Subtraction of data sources leaving only relationships found in a single source can also be used as a validation of the technique; highlighting relationships found only in one, perhaps less common, source.

Chapter 6

Augmenting Change Impact Analysis with Semantic Relation Information from Version Management Systems

6.1 Introduction

Having established that co-commit information from repositories seems to hold useful relationship information beyond static analysis (Chapter 4), and defined a generic framework for the use of multiple information sources to determine component relationships (Chapter 5), it is now desirable to implement and evaluate this novel approach. One problem domain in which such dependency traceability is clearly beneficial is that of predictive change impact analysis (Section 2.6), so in this chapter the approach is implemented in this domain.

Predictive Change Impact Analysis (PCIA) is the ability to determine, for a given change to a system, the probable and/or likely impacts (ripple effects) to or within other components of the system. Commonly engineers know what needs updating to make a change, the challenge is without deep understanding as to how the components are interrelated, to predict the effect that change will have. PCIA is never perfect but acts as a starting point for engineers to know which elements they need to consider that may be impacted and, as importantly, which ones are suitably isolated.

To apply the combinational approach to this domain the process outlined in Section 5.2 and shown in Figure 5.1 will be applied. This implementation will use static reverse engineering output and semantic source code repository links as the two information sources. The approach will be implemented and the strength of relationships found between components used to determine coupling hence the likely impact of a change of one having an effect on the other.

6.1.1 Problem Statement

Can semantic relationships, mined from a source code repository, be used in combination with static reverse engineering to augment, e.g. perform in a better or more accurate manner, predictive change impact analysis?

6.1.2 Experiment

In order to investigate the problem statement an experiment was designed. This involved the implementation of a tool compatible with the generic approach out-

lined in Chapter 5 (Section 6.2). To evaluate performance of the augmented data (Section 6.3) a series of real-world test subjects were identified (Section 6.3.1) and performance was validated and evaluated against manual analysis (Section 6.3.2), an industry standard tool JRipples (Section 6.3.3), and the actual follow-up changes made to the project (Section 6.3.5).

A discussion (Section 6.5) and conclusion (Section 6.6) finds that the approach of combining data does work and certainly does identify a greater set of potentially impacted components than JRipples. An apparent problem however is that, in some circumstances, the new approach appears to widely "over-match" leading to a large number of false positives and a unacceptably high coverage of all contained classes within the impact set. Some potential ideas for future work to address these issues and increase precision while maintaining high recall are identified.

6.2 Implementation

Although reverse engineering can be imperfect, some tools are capable of a complete detection of structural elements and straightforward relationships within source code (Chapter 3). These elements include constituent classes and interfaces, in addition to generalisation, composition, aggregation and, in some cases, association relationships.

In simple terms these reverse engineered relationships can be represented as a set of intersections S_R as detailed in Section 5. For each structural element there is a recorded intersection for every other element with which it has a relationship of some type or other, and an integer value representing the number of shared relationships. S_R is then normalised to contain intersection values in the range [0..1]. Relationships contained in S_R have all been generated directly from the source code and can therefore be considered to be accurate; S_R is a set of *ground truth* relationships (every relationship in S_R is recorded in the truth matrix T_R, see Section 5.2.3, specifically equation 5.7).

Similarly a set of relationships S_M can be gained from mining a source code repository, and identifying times that files (hence, in Java, classes) are committed together. A series of pre-filters is applied to determine which commits are included in the set. For example, only those that have more than one file and less than a

upper threshold at which too many files are committed for it to be meaningful. S_M will then contain intersections of all files with discrete integer values representing the number of commits they share. Following normalisation a defined significance threshold is applied to the set S_M leaving only relationships deemed as above the given significance threshold, and the file classes identified (e.g. for significance x, $S_M = F(S_M, x)$, see Section 5.2.2).

These sets can then be combined or compared in a number of ways. To gain a picture of the overall set of relationships, the sets can be combined together, containing all the relationships found in either, or both, approaches. In this way a new result set R_{RM} is generated using the combinational formula from Section 5.2.3 so that, with equal weights ($w_i = 1$) for S_R and S_M, where $n = 2$ and $S_1 = S_R, S_2 = S_M$:

$$R_{RM} = \frac{\sum_{i=1}^{n} w_i S_i}{\sum_{i=1}^{n} w_i} = \frac{1 \times S_R + 1 \times S_M}{1 + 1} = \frac{S_R + S_M}{2} \tag{6.1}$$

As previously stated it is expected there will be significant overlap between these two sets, many of the relationships will be found both through reverse engineering and semantic mining. It is possible to demonstrate the effectiveness of repository mining by subtracting relationships found in S_R, the *ground truth* relationships, from S_M to build a result set R_{UM} ($R_{UM} = S_M - S_R$). In this way R_{UM} only contains relationships deemed to be semantically significant but not identified purely by reverse engineering.

6.3 Evaluation

In order to evaluate the augmentation technique using real-world code examples three different approaches were used. These were:

1. Comparison with an industry standard.
2. Analysis of actual follow-up changes.
3. Manual analysis.

Evaluation was performed against a set of real-world test subjects (eight subjects from four different projects) with available Java source code repositories. The full set of targets and analysis including necessary binaries to run the experiment can be found and freely downloaded at [Cutting, 2015a]. Having been identified as possible candidates, test subjects were analysed to find a specific change suitable for analysis (Section 6.3.1) and include the software as a test subject.

An analysis of the relationships contained within test subjects was then performed using two information sources (static reverse engineering and repository mining) in accordance with the processes detailed in Chapter 5. Strong relationships were deemed to be an indication of change impact between classes and these findings were then compared against an industry standard change impact tool.

To identify industry standard tools, to be used in evaluation, literature was analysed [Lehnert, 2011; Li *et al.*, 2013] to build a list of comparable change impact analysis tools, operating on Java at the class level of granularity. This search identified tools such as JFlex [Hoffman, 2003] and JRipples [Buckner *et al.*, 2005]. Of these JRipples was consistently still in use and available for analysis and thus was selected as the basis of comparison.

6.3.1 Test Subjects

Test subjects (targets T) were chosen on the basis of being medium to large Java codebases with available public source code repositories. Once a specific project was chosen as a candidate for inclusion, the current repository was cloned (T_x). To make the experiment as realistic as possible, a real change on a class would need to be identified. A manual check of the commit logs identified a number of potential points during development at which a specific change on a clearly identified class was committed (point y). A list of the targets used for testing is contained in Table 6.1.

The changed class then became the impact subject (S) for the experiment. The source tree was copied and then "reset" (git hard reset) to the commit at the point that file was changed (T_xC_y). In this way the source tree, and available change log/commit log information, was then representing the exact point at

Target Project	
Short Name	**Class Count**
Apache Cayenne	
cayenne-55a	2601
cayenne-55e	2774
Google Guice	
guice-1dd	407
guice-1f9	422
Hearthsim Game Simulator	
hearthsim-118	557
hearthsim-787	582
JFlex Java Lexical Analyser	
jflex-bdd	94
jflex-de1	85

Table 6.1. Target Projects Used for Validation

which the change was made. Reverse engineering (using IBM Rational Rhapsody) was then performed on the source in its target state and the resultant XMI structural definition saved along with the change logs to point y.

6.3.2 Validation Against Manual Analysis

All of the previous validation techniques set out in this Section have relied on automated analysis and comparison using a set of test subjects and comparing, in different ways, against an industry standard or subsequent actual changes. Although very useful much of this does assume the accuracy of the industry standard tool and says little about the remaining positives, e.g. those impacts identified by jcRA but not by JRipples. These impacts could be false positives but could also be correct impacts not found by JRipples.

To attempt to resolve this question a manual analysis on one target (jflex-1dd) was performed in which other coupled classes were identified. As the relationship detection for both jcRA and JRipples are bi-directional classes which made use of, or were highly used by, the target, were included. Note that as with all the data in this section the full results of the manual analysis are available for download at Cutting [2015a].

Description	**Manual Analysis**	**JRipples**	**jcRA**
Class Count	25	17	51
Contained in Manual Analysis (Positive Positives)	-	17	25
Not in Manual Analysis (False Positives)	-	0	26
Precision	-	1.00	0.51
Recall	-	0.68	1.00

Table 6.2. Manual Analysis Results for jflex-1dd

Once the manual analysis had identified those classes *correctly* believed to be related to the target, standard information retrieval metrics could be used to calculate precision and recall. These results are contained in Table 6.2.

6.3.3 Wider Validation with JRipples

To compare and contrast the output from JRipples it was first necessary to generate a change impact set. For each target and change set (every T_xC_y) the Java code was imported into JRipples, and an Impact Analysis set (I) generated with the subject class S set as changed. This simulated the real-world application of JRipples as a change impact tool, where the class to be changed (S) was known, and the user wishes to see the other classes JRipples believes may be impacted by the change to S (the impact set).

The resultant change impact set (I) is a set of classes JRipples identifies as being potentially or likely to be impacted by a change to class S for codebase T_xC_y. This set is represented as: $I_r(S, T_xC_y)$.

To generate the impact from the implemented tool the process was repeated in a similar fashion using the *jcRelationAnalysis* tool with the same change target S and source state T_xC_y. This then generated a change impact set represented as: $I_j(S, T_xC_y)$.

To gauge relative performance with JRipples it was then necessary to calculate what proportion of impacted classes identified by each approach are identified by

the other (what the overlap is from JRipples to jcRelationAnalysis and vice-versa).

The proportion of classes identified by JRipples contained in the jcRelationAnalysis set is represented as: $I_r(S, T_xC_y) \cap I_j(S, T_xC_y)$

The proportion of classes identified by jcRelationAnalysis contained in the JRipples set is represented as: $I_j(S, T_xC_y) \cap I_r(S, T_xC_y)$

A further consideration for each is the *coverage* of the change impact set within the total codebase, i.e. what proportion of the total classes contained (CC) are within the impact set. Such a calculation is required as a high coverage in terms of overall classes may indicate lower precision even if recall is good, and hence "noise" in the presented data. At higher coverages the output becomes unusable as such a large proportion of classes are identified as potentially impacted. At the extreme it is clear that if 100% of classes are identified as potentially impacted, all the true positives are contained within this set, but the size is so large as to make it unmanageable and meaningless.

The proportion of classes identified by JRipples of the total class count is represented as: $I_r(S, T_xC_y) \cap CC(T_xC_y)$

Likewise, the proportion of classes identified by jcRelationAnalysis of the total class count is represented as: $I_j(S, T_xC_y) \cap CC(T_xC_y)$

The results of these experiments and comparisons, shown in Table 6.3, demonstrate a range of performance depending on target. In all cases the majority of the classes identified as potentially impacted by JRipples were also identified by jcRelationAnalysis with the coverage given as $I_j(S, T_xC_y) \cap I_r(S, T_xC_y)$ ranging from 66.67% to 100%. For half of the targets 100% coverage was found, and in a further quarter coverage was above 94%.

One evident difference between JRipples and jcRelationAnalysis was in total class coverage, the proportion of the total class count $CC(T_xC_y)$ identified as potentially impacted by an approach. JRipples consistently showed a much lower coverage whereas jcRelationAnalysis had not only a higher coverage in all cases but in some was so high as to be impractical (in excess of 90%) although this was very target-dependent. At this early stage only naive tuning has been performed and further analysis of tuning results holds the possibility of decreasing coverage and perhaps increasing detection rates (Section 6.3.4).

Target	**Class Count** $CC(T_xC_y)$	**JRipples Coverage** $I_r(S,T_xC_y) \cap CC(T_xC_y)$	**jcRA Coverage** $I_j(S,T_xC_y) \cap CC(T_xC_y)$	**jcRA found of JRipples** $I_j(S,T_xC_y) \cap I_r(S,T_xC_y)$
cayenne-55a	2601	89 3.42%	231 8.88%	75 84.27%
cayenne-55e	2774	12 0.43%	28 1.01%	8 66.67%
guice-1dd	407	6 1.47%	67 16.46%	6 100%
guice-1f9	422	1 0.24%	52 12.32%	1 100%
hearthsim-118	557	86 15.44%	513 92.10%	84 97.67%
hearthsim-787	582	99 17.01%	419 71.99%	94 94.95%
jflex-bdd	94	4 4.26%	16 17.02%	4 100%
jflex-de1	85	18 21.18%	52 61.18%	18 100%

Table 6.3. Validation Results Against Targets

The differences between targets is very marked and so it is also likely that the current jcRelationAnalysis approach is sensitised to certain development practices or coding styles; be that the way in which the codebase evolution has been managed or how interrelated elements are. Beyond tuning it is also possible a pattern could be defined as to what elements affect jcRelationAnalysis performance, and this could be used to ascertain potential use before application.

6.3.4 Tuning and Sensitivity Levels

Although filters were applied to the data, such as exclusion of commits including over a set threshold of classes, at most stages of the process there is scope for further tuning or adjustment of input parameters. For example both data sources are normalised in the same linear fashion with the value 1 representing the highest level found (equation 4.11) and every value then mapped between 0 and 1 as a proportion of its scale to the highest value. Actually, distributions vary sig-

nificantly and there are a number of alternative approaches to the normalisation that could be taken. Such alternatives would include techniques based around deviation from the mean, setting of outlier bounds, or similar processes.

Although it may be expected for example that the distribution would be Gaussian in nature, similar to that shown in Figure 6.1, the real world examples are very sparse and with much lower general values as shown in Figure 6.2. From this it can be seen some specialist tuning in the creation and filtering of the normalisation parameters may be required. Such extensive modification provides for potential rich future work investigating the most appropriate mechanisms to use (see Section 7.2).

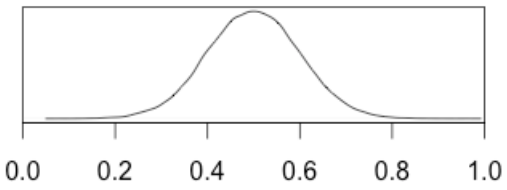


Figure 6.1. Centralised Distribution

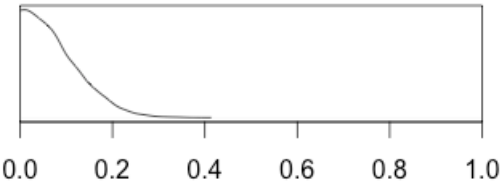


Figure 6.2. Low Centred Distribution

For this series of experiments the only tuning input used is a naive significance filter (F; Section 5.2.2, equation 5.5) applied to the normalised committal data. This sets a value above which the interaction is viewed as significant enough for inclusion, and below which would be ignored.

To visualise tuning performance and compare its effectiveness a series of runs were performed with varying values of significance filter and plotted as coverage of JRipples impacts found $I_j(S, T_xC_y) \cap I_r(S, T_xC_y)$ and total class contents $I_j(S, T_xC_y) \cap CC(T_xC_y)$. The intention being to find a value at which the lowest overall class coverage $(I_j(S, T_xC_y) \cap CC(T_xC_y))$ yields the highest JRipples

coverage ($I_j(S, T_xC_y) \cap I_r(S, T_xC_y)$); the highest recall with the highest precision. The results for some of the targets can be seen in Figure 6.3. The "stepped" values shown are possibly a result of linear normalisation causing loss of precision leading to very tightly grouped values.

As can be seen there is a reduction in the proportion of JRipples impact classes found as the significance filter is increased. The exact relationship varies widely depending on the target in question, with some apparently offering potential to reduce overall coverage while keeping JRipples coverage high (Figure 6.3a) and others with little or no potential (Figure 6.3d). Where potential benefit was seen the experiment was repeated using that significance filter value and those results used.

6.3.5 Actual Follow-up Changes

To further validate the approach an investigation was performed into the actual (*real world*) follow-on changes made. This process involved building a set of the classes included in the next N commits after T_xC_y on the basis they as all commits chosen as y were single-class changes may indicate an impact of the change.

A straightforward analysis could then determine the number of classes contained in the next N commits and the number of those found by the JRipples and jcRA approaches. Of course such a simplistic analysis is somewhat limited and assumes there are further impacted changes within the next N commits. Other activity, for example a parallel development effort on an unrelated subject, may severely affect the efficacy of this validation. It is clear that further work and better analysis is needed before this validation can be fully meaningful and accurate but include it here as an indication of the efficacy of the approach.

The results for $N = 10$, shown in Table 6.4, show that jcRA always includes a higher proportion of the next 10 changes in its impact set. In some ways this is unsurprising as in every case the total class coverage of jcRA was higher than JRipples. In several cases however the difference in recovered proportion was higher than the increase in overall coverage.

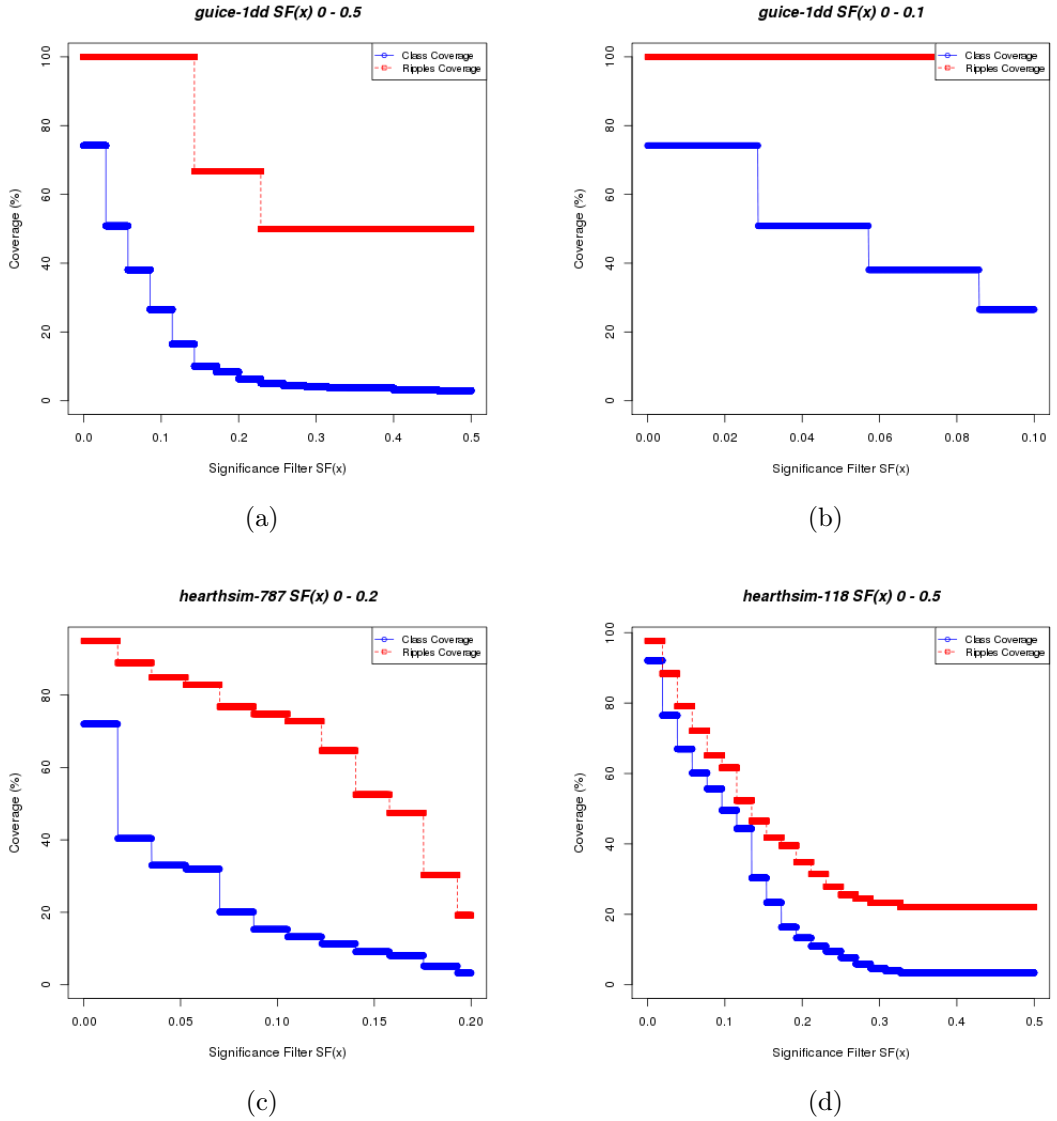


Figure 6.3. Naive significance filter tuning parameters applied to target samples. The x axis shows the increasingly significance filter (as it is higher, relationships with a value lower than it are discarded removing noise). The y axis shows the percentage of class coverage. The blue series indicates class coverage of the overall target (what percentage of all the classes in the target are included in the change set) while the red series shows what proportion of the classes in the JRipples change set are in the jcRA change set. An ideal is for the blue series (overall coverage) to be as low as possible and the red series (JRipples coverage) to be as high as possible for the same values.

Target	**Next 10 Change Count**	**JRipples**	**jcRA**
cayenne-55a	36	6	8
cayenne-55e	50	4	7
guice-1dd	10	1	8
guice-1f9	36	1	26
hearthsim-118	102	42	93
hearthsim-787	143	84	123
jflex-bdd	73	2	13
jflex-de1	57	17	51

Table 6.4. Next 10 ($N = 10$) changes contents (classes contained in next 10 updated) and of those classes the number identified in impact set generated by JRipples and jcRA

6.4 Presentation of Change Impact Results

As previously explained, presentation of a raw relationship matrix (as a huge table) is far from ideal, being largely incomprehensible and more complex than even large structural diagrams. The intention of the insight process is to aid developers in comprehension, not overload them further. The tooling (*jcRelationAnalysis*) therefore contains a number of provided report types (and is highly extensible for more types to be defined as needed), which are available to the developers.

The most prescient of these are the dependency/change impact trees. These trees are sets of data ordered by a subject class showing which other classes are determined to have a relationship with the subject, and therefore are potentially impacted by changes to the subject. Inbuilt reporting can both generate a general set showing all links for all constituent classes, exported in plain-text or interactive HTML, as well as a targeted change impact tree for a given class. This is implemented through the targeted impact tree (report name *AnalysisChangeImpactTree*) with a configured subject class and a depth. Dependencies directly linked to the target are at depth one (first order dependencies) and the report will continue, if so configured, to then look in turn for dependencies of these not previously included in the report, to the configured depth.

Such a report format allows a developer to assess the impact of a change within

a specific class, a much more appropriate real-world use case than understanding all relationships at once [Wloka, 2006]. Such a model is similarly used in various change impact approaches including that compared in Section 6.3.

6.5 Discussion

Following the evaluation of the approach in Section 6.3 it seems that the implemented tool *jcRelationAnalysis* (jcRA) is able to produce relevant impact sets for a given target, using the two sources of information and the combinational matrix model. When compared to an industry standard tool, JRipples, in the majority of cases jcRA could find nearly all of the impacts identified by JRipples. The overall coverage however, the proportion of total classes contained which are included in the potential impact set (the overall coverage), was higher in jcRA, often significantly so. In some cases the overall coverage was itself so high as to make the results from jcRA meaningless, though this was highly target-dependent.

In addition to the automated analysis a manual code analysis for a sample target was performed and used for comparison between JRipples and jcRA. As perhaps could be expected given the larger overall coverage, jcRA's precision suffers in comparison to JRipples, but there are cases in which classes which are identified in the manual analysis are not found by JRipples but are by jcRA. This leads to a situation in which each tool outperforms the other in one metric, JRipples with a higher precision and jcRA with a higher recall.

It is clear from the results jcRA does generate relevant impact sets (when compared with JRipples based on manual analysis of the code in Section 6.3.2) but with a larger general coverage of the codebase. Generally it could be the case that, excepting overly large general coverage to the point of confusion, erring towards inclusion e.g. including all impacts at the cost of some false positives, would be a better outcome than excluding impacted classes but this would require further research. In this case false positives refer to classes not subject to change which are erroneously included in the change impact set. With no consideration for classes not changed there are no false positives possible. As previously stated however a variety of tuning possibilities exist within jcRA, with the possibility of additional sources of data to clarify the results further. It is therefore possible

that the precision issue could be addressed without lack of recall. Alternatively a decision could be made early in the analysis as to the overall sensitivity values, as to which side to err on, to include or not to include. This informed user decision would then be used to set the necessary parameters and adjust overall sensitivity.

A significant amount of work can still be performed in terms of tuning the approach and also incorporation of additional sources of information, either just in addition or as filters.

At many points during the analysis process there is the opportunity for configurable tuning parameters such as sensitivity levels, cut-offs, weightings, or other methods of normalisation to be applied. Built in a highly extensible manner jcRA further offers the option of different methods of analysis and processing to be included. At this stage, wanting only to validate the concept and efficacy of multiple sources used in this fashion, little of this has been considered save some naive significance filtering. Such a high degree of flexibility and customisation, especially in light of the very target-dependent performance, may also involve adding a stage to the process in which the target is analysed and the "best" parameters for tuning automatically generated.

The flexible and entirely generic nature of the matrix comparison model used in the approach is not just highly customisable and tunable through weighting but easily extended to include further sources of information. Any data source which can be translated into a quantifiable relationship between elements (classes) in a software base can be generated as a matrix model within jcRA. Once the matrix is generated it can be manipulated directly, filtered, modified, and normalised. Further, the matrix can be used in combination with other matrices from different data sources in a variety of ways such as addition, weighted multiplication, or subtraction for unique relationships. A simple combination allows relationships to be summarised together, as used for the two sources in this chapter. Additionally support is available to use matrices as a weight, a filter, or to subtract rather than add, removing relationships identified as unnecessary.

There is, therefore, significant opportunity for future work in a variety of ways to refine or extend the approach further (see Section 7.2). The ultimate aim is to release the entire toolset and source code to the community, raising the possibility of third-party collaboration. Currently the binary release is available with the

dataset from these experiments [Cutting, 2015a] and the user documentation is also freely available [Cutting, 2015b].

6.6 Conclusion

In this Chapter the implementation of a change impact analysis use of the approach is presented through the jcRelationAnalysis (jcRA) tool. To validate this approach and tool it was applied to a number of targets, generating change impact sets, and then these were compared against an industry standard tool, JRipples, in a number of automated and manual tests. The results show that the approach is capable of generating relevant change impact sets and appears to have a higher recall than JRipples but with a lower precision. A number of avenues of future work have been identified which may be able to improve the efficacy of the tool using current data sources, or indeed add more data sources.

Traditional reverse engineering, based on static code analysis, is still the standard method to analyse code bases for developer insight [Roscoe, 2011] although there are problems with the consistency and standard of the output [Cutting and Noppen, 2014, 2015].

The mining of software repositories for meta information is an exciting and still growing area [Allamanis and Sutton, 2013; Kagdi *et al.*, 2007a,b; Williams and Hollingsworth, 2005] offering rich potential insight. The challenge remains how to best analyse and use this information to generate useful insights [Hassan, 2008].

Use of such data sources in combination is by no means new and a growing area of focus with studies such as Bieman *et al.* [2003] indicating that relationships found through repository mining can exceed those found by static analysis alone.

The most recent comparable approach to the one presented is a tool called ImpactMiner [Dit *et al.*, 2014], which uses a Subversion (SVN) repository combined with static analysis, to produce change impact analysis. ImpactMiner is available as a dropin for Eclipse and includes a well featured user interface. The precise approach taken is less generic than jcRA, requiring use within the Eclipse platform and analysis of code within an Eclipse project. ImpactMiner operates as a lower level than the jcRA approach, operating at the method rather than

class level. Some other shortcomings in ImpactMiner such as the iterative nature (it runs an analysis for every variant of the software contained in the repository) and limitation to SVN have been identified by the ImpactMiner team as areas for improvement and likely to be addressed [Dit *et al.*, 2014].

In conclusion, it can be seen that the approach presented offers a number of potential benefits and has been demonstrated to perform well on different targets when creating change impact sets. Unlike either the industry standard, JRipples, or the more recent ImpactMiner, the implementation of the approach (jcRelationAnalysis) runs outside of Eclipse as a standalone, configurable, extensible, and scriptable tool. This makes it suitable for more detailed programmatic or iterative use, with the input and output changed and used as required. A range of future work tuning the framework and extending functionality is also possible, and by using a generic matrix approach, further data sources can be easily integrated.

In reference to the problem statement (Section 6.1.1) it has clearly been demonstrated that it is possible to use multiple, weighted, sources of information together to find relationships other than those expressed in static reverse engineering alone. Further, it is possible to apply these relationships to the domain of predictive change impact analysis and offer better performance than current tools.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The fundamental challenge set out in Section 1.1.1 was "is it possible to reconstruct richer traceability structures through the combinational use of multiple sources of information?".

To that end, in this thesis a number of research topics have been introduced with the goal of enhancing legacy software system analysis by combining behavioural and semantic information sources. Initially, in Chapter 2, the current problem with legacy software and the solutions to comprehend poorly documented existing systems through traceability and reverse engineering techniques were investigated.

Source code reverse engineering to recreate design artefacts such as UML projections were found to be the most commonly used processes with a wide variety of tools and approaches to accomplish the task. Lacking, however, was any benchmark against which these varied tools could be compared against each other and a set standard, and/or which could further provide validation for any new tools or approaches. To accomplish this, in Chapter 3, the creation of the Reverse Engineering to Design Benchmark (RED-BM) was described. RED-BM consisted of a number of real-world software projects to be used as targets, along with a gold standard output (what structural and relationship elements are contained, and should be detected, within the source) and a number of defined metrics. This benchmark was then applied to a range of industry standard tools to judge their performance.

Performance was varied and the main metric (the compound measure which took into account both structural elements and relationships across all sizes of provided targets) recorded results ranging from 8.82% to 100% os structural elements successfully recovered. This shows that many of the tools provided for reverse engineering are imprecise and also demonstrates the effectiveness of the benchmark at being able to distinguish a wide range of performance.

In the process of creating the benchmark a number of tools were created to automate the process of collecting and measuring software and reverse engineering output as much as possible and to aid replication or repetition by third-parties. Part of this was to facilitate the programmatic use of output by reverse engineer-

ing tools. There exists a standard (XMI) which supposedly allows interoperability but in practice the implementations between tools are so varied they are neither compatible nor standard. A generic parsing component was therefore created capable of understanding all the common XMI dialects and also UML expressed in XML. This was then reused in a variety of further tools and approaches in the rest of the research.

Even when it works well, static reverse engineering alone is only able to recover clear structural relationships contained within the code. More information can be found through analysis of the historical data contained within source code repositories which record the evolutionary history with which artefacts have been changed throughout the life of the project. A specific concept with relation to dependency traceability, identifying semantic relationships, is that of co-commits; artefacts that are often changed together may have an inferred relationship.

The possibility of using such relationships was investigated in Chapter 4, where an approach and method to mine them was defined and implemented. Analysis of the recovered semantic relationships found within the repositories using a number of techniques, such as network diagrams and clustering, found that clear relationship groupings and logical sectioning did exist. An analysis of these groups found that logical artefact collections were present and visible within the recovered data. Although more work is required to stabilise or find more applicable forms of grouping/clustering, the purpose of demonstrating strong sets of semantic relationships, in addition to the static relationships recovered by reverse engineering, was fulfilled.

Having shown the variation in information contained from different sources and that sources differed in terms of the relationships contained, it was clear that combining multiple diverse sources could offer richer traceability information. Therefore, with multiple information sources being used, and the possibility of more sources being included, it was desirable to define a generic approach through which disparate sources could be used in combination. This was carried out in Chapter 5. Using a number of examples to illustrate and theoretically test the model, a set of formal functions and equations were created to allow repeatable and consistent application of the method to different sources. Any data source which could, at the *lowest common level* of granularity, be expressed

as an $n \times n$ matrix of component intersections with relationship values could be included. Support was added for the concept of ground truth, e.g. relationships sets which are known to be true regardless of other data, for example where strong static relationships are recovered. Manipulation, filtering, and combination operations were defined including weighted combinations in which different information sources could be given relative weights, or importance, before being combined into a summary matrix.

To demonstrate the efficacy of the generic approach, in Chapter 6 it was implemented in the domain of change impact analysis, using static reverse engineering and source code semantic relationships as the two sources. This implementation was in the form of the *jcRelationAnalysis* (jcRA) tool. The tool was applied against a number of real-world software projects, reverted to a historical point where a real change was about to take place, and this change was theorised to build a projected impact set. To evaluate the approach an industry standard tool, JRipples, which uses source code analysis alone was also asked to generate an impact set for the same state and change. While this research was being carried out a similar approach was implemented by another research team (ImpactMiner), though when analysed this was found to operate at a very different level of granularity. Further, ImpactMiner requires a significant processing overhead, reconstructing every intermediate step of a system, which the implemented approach does not and in fact runs quickly on standard destkop hardware.

Evaluation showed that the new method, applied to change impact, outperformed JRipples in terms of recall, but had a lower precision which in some cases over-matched to the extent that the coverage was so large as to be un-practical. An investigation into simple tuning based on *naive significance filters* proved some benefit but it is clear there is still work to be done on this specific implementation. The implementation as a proof of the generic method's feasibility and scope however was a complete success with all the constituent components and processes being implemented successfully.

In summary, following a review of current practice, a series of problems were identified and framed. Ultimately, a generic framework for the comparative use of multiple information sources in an empirical and repeatable way was defined. This framework was then implemented within the change impact analysis domain

and generated useful and practical results which in many cases outperformed an industry standard method. This work proved that is it indeed possible to reconstruct richer, more informed, traceability structures through the combinational use of multiple sources of information.

7.2 Future Work

> "There will always be more questions. Every answer leads to more questions. The only way to survive is to let some of them go."
> – *David Levithan, Every Day*

To aid readability the individual areas for future work are split into their thematic (chapter) areas.

Reverse Engineering Benchmark - Chapter 3

Other Languages: Expand the RED-BM target set to include other object-oriented languages, specifically C++ and repeat looking for language variation on tools that support multiple versions.
Additional Metrics: Define additional metrics with required supporting gold standards and output, specifically looking at design pattern and "code smell" detection.
Wider Range of Targets: Include a wider range of targets covering a number of different architectural paradigms (the current set include a range, but provide a specific grouping of targets for a number of paradigms). Also include more targets at different code sizes, larger and smaller than the current set as well as more within the set range.

Source Code Repository Mining - Chapter 4

Refactoring Detection: Currently each artefact is seen as a single discrete entity for the analysis but often refactoring occurs through, for example, renaming of packages which then renames the files (changes their absolute path). With the current approach these two files are included in the data as two entries so, when used in combination with the current picture (reverse engineered) the "old version" relationships are lost.

It should be possible to detect these changes, and similar depending on the exact repository software used and migrate the relationship data to the "new version" of the artefact.

Generic Approach to Combination - Chapter 5

Granularity Level: Allow for the inclusion of data sources at a higher level of granularity without raising the overall level, for example "decompose" such sources to a mean relationship shared with their constituent sub-components (e.g. class relationships "decompose" to relationships between their constituent methods).
Traceability: Add a specific support within the framework for a third set of data carried through which records a narrative of where the relationships came from, and any other relevant meta data, allowing end-to-end traceability for the presented relationships (e.g. a presented rationale for the decisions made within the approach).

Predictive Change Impact Analysis Implementation - Chapter 6

Further Information Sources: Combine further information sources, potentially a technical source such as call-tracing through dynamic code analysis and a documentary source such as existing documentation parsed through object-modelling technique or code comment analysis.
Greater Range of Tuning Types and Filters: Current filters are *naive significance filters* and operate in a very straightforward way. Given the widely varied form of the data and the often sparse nature of the matrices, a very good potential improvement avenue would be through building and testing more complex data-aware filters. This could include the option to run an analysis with different types and parameters of filters to find a "best fit" for the particular target and information sources provided.

References

AgileJ (2015). AgileJ structure views. http://www.agilej.com/. [Online; accessed December 2015]. 26

Ali, N., Gueheneuc, Y., and Antoniol, G. (2011). Trust-based requirements traceability. In *2011 IEEE 19th International Conference on Program Comprehension (ICPC)*, pages 111–120. 32

Ali, N., Gueheneuc, Y.-G., and Antoniol, G. (2013). Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Transactions on Software Engineering*, 39(5):725–741. 30, 32, 87

Allamanis, M. and Sutton, C. (2013). Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, page 207–216, Piscataway, NJ, USA. IEEE Press. 4, 29, 31, 76, 131

Altova (2012). Altova umodel. http://www.altova.com/umodel.html. [Online; accessed December 2012]. 26

Anquetil, N. and Lethbridge, T. C. (1999). Experiments with clustering as a software remodularization method. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 235–255. IEEE. 34, 39

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983. xii, 30, 32, 33

Arcelli, F., Masiero, S., Raibulet, C., and Tisato, F. (2005). A comparison of reverse engineering tools based on design pattern decomposition. In *Software*

Engineering Conference, 2005. Proceedings. 2005 Australian, pages 262–269. IEEE. 21, 22, 25, 47, 48, 72, 99

ArgoUML (2016). Argouml. http://argouml.tigris.org/. [Online; accessed February 2016]. 26, 59

Asuncion, H., François, F., and Taylor, R. (2007). An end-to-end industrial software traceability tool. In *Foundations of Software Engineering: Proceedings of the 6 th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, volume 3, pages 115–124. 10, 11

Bellay, B. and Gall, H. (1997). A comparison of four reverse engineering tools. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 2–11. IEEE. 48, 53, 63

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591. 48, 72

Beyer, D. and Noack, A. (2005). Clustering software artifacts based on frequent common changes. In *13th International Workshop on Program Comprehension, 2005. IWPC 2005. Proceedings*, pages 259–268. 4, 34, 76, 93

Bieman, J. M., Andrews, A. A., and Yang, H. J. (2003). Understanding change-proneness in OO software through visualization. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 44–53. IEEE. 4, 34, 76, 93, 131

Boehm, B. W., Brown, J. R., and Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press. 10

Bohner, S. and Arnold, R. (1996). *Software Change Impact Analysis*. Wiley-IEEE Computer Society, 1 edition. 40

Bohner, S. A. (2002). Extending software change impact analysis into COTS components. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 175–182. IEEE. 45

Briand, L. C., Labiche, Y., and Soccar, G. (2002). Automating impact analysis and regression test selection based on UML designs. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 252–261. IEEE. 45

Brooks, F. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19. 2, 40

Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V. (2005). Jripples: A tool for program comprehension during incremental change. In *IWPC*, volume 5, pages 149–152. 45, 120

Cerulo, L. (2006). On the use of process trails to understand software development. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 303–304. IEEE. 21

ChangeVision (2016). Astah professional. `http://astah.net/editions/professional`. [Online; accessed February 2016]. 26, 83

Chikofsky, E., Cross, J., *et al.* (1990). Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17. 3, 13, 14, 15

Cleland-Huang, J., Gotel, O., and Zisman, A. (2012). *Software and Systems Traceability*. Springer. 2, 10, 11, 12, 13

Cleland-Huang, J., Gotel, O. C., Huffman Hayes, J., Mäder, P., and Zisman, A. (2014). Software traceability: trends and future directions. In *Proceedings of the Conference on the Future of Software Engineering*, pages 55–69. ACM. 2, 10, 11

Counsell, S., Newson, P., and Mendes, E. (2004). Design level hypothesis testing through reverse engineering of object-oriented software. *International Journal of Software Engineering and Knowledge Engineering*, 14(02):207–220. 3, 14, 15, 22

Cutting, D. (2013). Reverse engineering to design benchmark. http://www.uea.ac.uk/computing/machine-learning/traceability-forensics/reverse-engineering. [Online; accessed May 2013]. xv, 3, 48, 49, 50, 51, 71, 72

Cutting, D. (2015a). jcRelationAnalysis Data Set. http://go.purplepixie.org/jcra-data-set. [Online; accessed December 2015]. 120, 121, 131

Cutting, D. (2015b). jcRelationAnalysis Documentation. http://go.purplepixie.org/jcra-documentation. [Online; accessed December 2015]. 131

Cutting, D. (2016). MDS and K-means video example. http://www.purplepixie.org/davestuff/MDS-Cluster-Video-DC.zip. [Online; accessed October 2016]. 93

Cutting, D. and Noppen, J. (2014). Working with reverse engineering output for benchmarking and further use. In *Procedings of the 9th International Conference on Software Engineering Advances*. IARIA. xi, 4, 47, 87, 99, 131

Cutting, D. and Noppen, J. (2015). An extensible benchmark and tooling for comparing reverse engineering approaches. *International Journal in Advances in Software*, 8(1&2):115–124. xi, 3, 4, 47, 99, 131

Dit, B., Wagner, M., Wen, S., Wang, W., Linares-Vásquez, M., Poshyvanyk, D., and Kagdi, H. (2014). Impactminer: a tool for change impact analysis. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 540–543. ACM. 4, 45, 131, 132

Dorfman, M. and Thayer, R. H. (1990). Standards, guidelines and examples on system and software requirements engineering. *IEEE Computer Society Press Tutorial, Los Alamitos: IEEE Computer Society Press, 1990, edited by Dorfman, Merlin; Thayer, Richard H.*, 1. 11

Du, H. (2010). *Data Mining Techniques and Applications: An Introduction*. Course Technology Cengage Learning. 34, 38, 39

Edwards, M. and Howell, S. L. (1991). A methodology for systems requirements specification and traceability for large real time complex systems. Technical report, DTIC Document. 10

Fenton, N. E. and Pfleeger, S. L. (1998). *Software metrics: a rigorous and practical approach*. PWS Publishing Co. 53, 63

Ferenc, R., Beszedes, A., Tarkiainen, M., and Gyimothy, T. (2002). Columbus - reverse engineering tool and schema for C++. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 172 – 181. 25

Fischer, G., Lusiardi, J., and von Gudenberg, J. (2007). Abstract syntax trees-and their role in model driven software development. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 38–38. IEEE. 16, 17

Flores, N. and Aguiar, A. (2005). Reverse engineering of framework design using a meta-patterns-based approach. *Software Stability: Timeless Architectures and System of Patterns*, page 10. 21, 22

Fowler, M. (1997). Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*. 3

Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9(8):28–35. 99

Fujaba (2012). Fujaba tool suite. `http://www.fujaba.de`. [Online; accessed December 2012]. 25

Fulop, L., Hegedus, P., Ferenc, R., and Gyimóthy, T. (2008). Towards a benchmark for evaluating reverse engineering tools. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 335–336. IEEE. 73

Galvao, I. and Goknil, A. (2007). Survey of traceability approaches in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 313–313. IEEE. 11

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional. 22, 24

Gaul, A. (2012). Function call overhead benchmarks with MATLAB, Octave, Python, Cython and C. *arXiv preprint arXiv:1202.2736*. 72

Gherardi, L., Brugali, D., and Comotti, D. (2012). A Java vs. C++ performance evaluation: a 3D modeling benchmark. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172. 72

Gotel, O. and Finkelstein, C. (1994). An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101. IEEE. 2, 11, 12

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661. 31

Grossman, M., Aronson, J. E., and McCarthy, R. V. (2005). Does UML make the grade? insights from the software development community. *Information and Software Technology*, 47(6):383–397. 77, 99

Guéhéneuc, Y.-G. and Antoniol, G. (2008). Demima: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.*, 34(5):667–684. 25

Harrington, G. and Rondeau, K. (1993). An investigation of requirements traceability to support systems development. Technical report, DTIC Document. 12

Hassan, A. E. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, page 48–57. xv, 4, 29, 30, 31, 32, 76, 131

Herzig, K. and Zeller, A. (2013). The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE. 29

Hoffman, M. A. (2003). Automated impact analysis of object-oriented software systems. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 72–73. ACM. 120

Huang, L. and Song, Y.-T. (2007). Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*, pages 374–384. IEEE. 44

Huang, L. and Song, Y.-T. (2008). A dynamic impact analysis approach for object-oriented programs. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 217–220. IEEE. 44

IBM (2015). Rational rhapsody. http://www-142.ibm.com/software/products/us/en/ratirhapfami/. [Online; accessed December 2015]. 27

IEEE (1984). IEEE guide to software requirements specifications. *IEEE Std 830-1984*. 11, 12, 13

Jiang, J. and Systä, T. (2003). Exploring differences in exchange formats-tool support and case studies. In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 389–398. IEEE. 57

Jirapanthong, W. (2015). Requirements traceability on web applications. In *2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 18–23. IEEE. 11

Jones, J. (2003). Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*. 16

Kagdi, H., Collard, M. L., and Maletic, J. I. (2007a). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131. xv, 4, 29, 30, 31, 34, 76, 77, 131

Kagdi, H. and Maletic, J. I. (2006). Software-change prediction: Estimated+ actual. In *Software Evolvability, 2006. SE'06. Second International IEEE Workshop on*, pages 38–43. IEEE. 4, 43, 45

Kagdi, H. and Maletic, J. I. (2007). Software repositories: A source for traceability links. In *Proceedings of 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07), Lexington, KY, USA*, page 32–39. 45

Kagdi, H., Maletic, J. I., and Sharif, B. (2007b). Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, page 145–154. 4, 30, 32, 45, 99, 131

Khan, S. S. and Lock, S. (2009). Concern tracing and change impact analysis: An exploratory study. In *Aspect-Oriented Requirements Engineering and Architecture Design, 2009. EA'09. ICSE Workshop on*, pages 44–48. IEEE. 45

Koschke, R. (2003). Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109. 52

Kumar, K., Kaur, P., and GNDU, A. (2015). A generalized process of reverse engineering in software protection & security. *International Journal of Computer Science and Mobile Computing*, 4(5):534–544. 3, 13, 14, 15, 47

Labiche, Y., Kolbah, B., and Mehrfard, H. (2013). Combining static and dynamic analyses to reverse-engineer scenario diagrams. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 130–139. IEEE. 21, 47

Laurent, P., Cleland-Huang, J., and Duan, C. (2007). Towards automated requirements triage. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 131–140. IEEE. 11

Lehman, M. M. and Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc. 5, 40

Lehnert, S. (2011). A review of software change impact analysis. *Ilmenau University of Technology, Tech. Rep.* 4, 5, 40, 43, 45, 120

Li, B., Sun, X., Leung, H., and Zhang, S. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646. xii, 40, 43, 44, 120

Mäder, P. and Egyed, A. (2015). Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441. 2, 11, 12

Mamas, E. and Kontogiannis, K. (2000). Towards portable source code representations using XML. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 172–182. IEEE. 14, 18

Marcus, A. and Maletic, J. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th International Conference on Software Engineering, 2003. Proceedings*, pages 125–135. 30

Merriam-Webster (2012). Merriam-webster dictionary. http://www.merriam-webster.com/dictionary/reverse%20engineering. [Online; accessed December 2012]. 13

Meyer, M. (2006). Pattern-based reengineering of software systems. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 305–306. IEEE. 3, 14, 15, 16, 48, 72

Microtool (2012). objectif. http://www.microtool.de/objectif/en/. [Online; accessed December 2012]. 27

Misek, J. and Zavoral, F. (2010). Mapping of dynamic language constructs into static abstract syntax trees. In *Computer and Information Science (ICIS), 2010 IEEE/ACIS 9th International Conference on*, pages 625 –630. xii, 17

Modeliosoft (2012). Modelio. http://www.modeliosoft.com. [Online; accessed December 2012]. 27

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE. 31

Naur, P. and Randell, B. (1969). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO. *Report of a conference sponsored by the NATO Science Committee*. 2, 10

NoMagic (2012). Magicdraw uml. http://www.nomagic.com/products/magicdraw/magicdraw-enterprise.html. [Online; accessed December 2012]. 27

Olivier, P., Boukhobza, J., and Senn, E. (2012). On benchmarking embedded Linux flash file systems. *arXiv preprint arXiv:1208.6391*. 72

OMG *et al.* (2005). Unified modelling language: Infrastructure. http://www.omg.org/spec/UML/2.0/. [Online; accessed December 2012]. 19, 20

OMG *et al.* (2011a). Architecture-driven Moderization: Abstract Syntax Tree Metamodel (ASTM). http://www.omg.org/spec/ASTM. [Online; accessed December 2012]. 16, 17

OMG *et al.* (2011b). OMG Meta Object Facility (MOF) Core Specification. http://www.omg.org/spec/MOF/2.4.1. [Online; accessed December 2012]. 19

OMG *et al.* (2011c). OMG MOF 2 XMI Mapping Specification. http://www.omg.org/spec/XMI/2.4.1. [Online; accessed December 2012]. 18, 19, 57, 63

Pages, B. (2016). BOUML. http://www.bouml.fr. [Online; accessed May 2016]. 26

Pettersson, N., Lowe, W., and Nivre, J. (2010). Evaluation of accuracy in design pattern occurrence detection. *Software Engineering, IEEE Transactions on*, 36(4):575–590. 72

Philippow, I., Streitferdt, D., Riebisch, M., and Naumann, S. (2005). An approach for reverse engineering of design patterns. *Software and Systems Modeling*, 4(1):55–70. 3, 14, 15, 16, 26, 55, 72

Pilch, C. (2009). *Applications of Multidimensional Scaling to Graph Drawing*. PhD thesis, University of Konstanz, Konstanz, Germany. 37, 38, 92

PINOT (2012). Pattern inference and recovery tool (pinot). `http://www.cs.ucdavies.edu/~shini/research/pinot`. [Online; accessed December 2012]. 25

Pinzger, M. and Antoniol, G. (2013). Guest editorial: reverse engineering. *Empirical Software Engineering*, 18(5):857. 3, 14, 15

Queille, J.-p., Voidrot, J.-F., Wilde, N., and Munro, M. (1994). The impact analysis task in software maintenance: a model and a case study. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 234–242. IEEE. 40

Raghavan, S., Rohana, R., Leon, D., Podgurski, A., and Augustine, V. (2004). Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 188–197. IEEE. xii, 18

Ramadani, J. and Wagner, S. (2016). Are suggestions of coupled file changes interesting? 4, 29, 34

Ramesh, B. and Edwards, M. (1993). Issues in the development of a requirements traceability model. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 256–259. IEEE. 11

Rasool, G. and Streitfdert, D. (2011). A survey on design pattern recovery techniques. *International Journal of Computing Science Issues*, 8. 14, 21, 22, 47, 48, 99

Rodina, D. (2012). Software ideas modeller. `http://www.softwareideas.net`. [Online; accessed December 2012]. 27

Rogers, S. and Girolami, M. (2011). *A first course in machine learning*. CRC Press. 34, 38, 39

Roman, G.-C. and Cox, K. C. (1993). A taxonomy of program visualization systems. *Computer*, 26(12):11–24. 52

Roscoe, J. (2011). Looking forwards to going backwards: An assessment of current reverse engineering. *Current Issues in Software Engineering*. 13, 14, 15, 47, 72, 99, 131

Sarkar, M. K., Chatterjee, T., and Mukherjee, D. (2013). Reverse engineering: An analysis of static behaviors of object oriented programs by extracting UML class diagram. *International Journal of Advanced Computer Research*, 3(3):135. 3, 15, 19

Shi, N. and Olsson, R. (2006). Reverse engineering of design patterns from Java source code. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 123–134. IEEE. 3, 21, 22, 25

Sim, S. E., Easterbrook, S., and Holt, R. C. (2003). Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pages 74–83. IEEE Computer Society. 48, 73

Spanoudakis, G. and Zisman, A. (2005). Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, 3:395–428. 2, 11, 12

SparxSystems (2012). Enterprise architect. `http://www.sparxsystems.com/products/ea/index.html`. [Online; accessed December 2012]. 27

SPOOL (2012). SPOOL. `http://www.iro.montreal.ca/~keller/Spool/main.html`. [Online; accessed December 2012]. 25

StarUML (2008). StarUML. `http://staruml.sourceforge.net/en/`. [Online; accessed December 2012]. 28

Uchiyama, S., Washizaki, H., Fukazawa, Y., and Kubo, A. (2011). Design pattern detection using software metrics and machine learning. In *First International*

Workshop on Model-Driven Software Migration (MDSM 2011), page 38. 3, 15, 16, 22, 24, 72

Umbrello (2012). Umbrello UML modeller. http://uml.sourceforge.net. [Online; accessed December 2012]. 28

VisualParadigm (2012). Visual paradigm for UML. http://www.visual-paradigm.com/product/vpuml/. [Online; accessed December 2012]. 28

Williams, C. and Hollingsworth, J. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480. 4, 29, 30, 31, 76, 131

Wloka, J. (2006). Towards tool-supported update of pointcuts in AO refactoring. In *Proceedings of the Workshop on Linking Aspect Technology and Evolution Revisited at AOSD 2006*. 129

Ying, A., Murphy, G., Ng, R., and Chu-Carroll, M. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586. 4, 34