Predicting Change Propagation Using Domain-Based Coupling

A thesis submitted for the degree of

Doctor of Philosophy

Amir Aryani

School of Computer Science and Information Technology

College of Science, Engineering, and Health

RMIT University

Melbourne, Victoria, Australia.

May 27, 2013

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Amir Aryani
School of Computer Science and Information Technology
RMIT University
2nd December 2012

Acknowledgments

This thesis would not have been possible without the support of many people to whom I am very grateful. First and foremost, I would like to thank my PhD supervisors, Dr Margaret Hamilton and Dr Ian Peake for their advice, guidance and continued confidence in my work. They supported me in pursuing many interesting ideas and provided encouraging words and advice when it was needed.

Also I want to thank Associate Professor Michael Winikoff and Professor Heinz Schmidt for their support and valuable advice that helped to enlighten the core of this research. Moreover, thanks to the other colleagues and collaborators who co-authored papers with me in the last few years: Professor Oscar Nierstrasz, Fabrizio Perin and Dr Mircea Lungu from the Software Composition Group in Bern, Assistant Professor Denys Poshyvanyk and Dr Malcom Gethers from the College of William and Mary, Assistant Professor Dr Chanchal K. Roy and Md Saidur Rahman from University of Saskatchewan, and finally Dr Abdun Naser Mahmood, Dr Venki Balasubramanian and Dr Ian Thomas from the RMIT University.

Special thanks to my friend Nicholas May who spent a lot of time with me brainstorming research ideas, and helped me with the proofreading of my papers.

I would like to thank Mercury Computer Systems (Australia) Pty Ltd. for allowing access to the BEIMS maintenance logs and domain information, and special thanks to Garry Busowsky and Norbert Riedl for their support of my research.

Part of this research work has been funded by an Australian Postgraduate Award (APA). In addition, I would like to thank the RMIT School of Graduate Research (SGR) for funding my travels to the international conferences that lead to the significant collaborations in this thesis.

Most of all, many thanks to my family and specially my beloved wife whose understanding and kindness gave me the freedom and energy to concentrate on my research.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- Md Saidur Rahman, Amir Aryani, Chanchal K. Roy, Fabrizio Perin. On the Relationships between Domain-Based Coupling and Code Clones: An Exploratory Study; 35th International Conference on Software Engineering (ICSE 2013), IEEE, San Francisco, CA, USA, May, 2013

- Malcom Gethers, Amir Aryani, Denys Poshyvanyk, Combining Conceptual and Domain-Based Couplings to Detect Database and Code Dependencies; in Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM12), IEEE, Trento, Italy, September, 2012

- Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Naser Mahmood, Oscar Nierstrasz, "Can we predict dependencies using domain information?," in Proceedings of the 18th Working Conference on Reverse Engineering (WCRE11), IEEE, Limerick, Ireland, October, 2011.

- Amir Aryani, Ian D. Peake, Margaret Hamilton, "Domain-Based Change Propagation Analysis: An Enterprise System Case Study," in Proceedings of the 26th International Conference on Software Maintenance (ICSM10), IEEE, Timisoara, Romania, September, 2010.

- Amir Aryani, Ian D. Peake, Margaret Hamilton, Heinz Schmidt, Michael Winikoff, "Change propagation analysis using domain information," in Proceedings of the 20th Australian Software Engineering Conference (ASWEC09), IEEE, Gold Coast, Australia, April, 2009.

Contents

List of Figures

List of Tables

Abstract

Most enterprise systems operate in domains where business rules and requirements frequently change. Managing the cost and impact of these changes has been a known challenge, and the software maintenance community has been tackling it for more than two decades. The traditional approach to impact analysis is by tracing dependencies in the design documents and the source code. More recently the software maintenance history has been exploited for impact analysis.

The problem is that these approaches are difficult to implement for hybrid systems that consist of heterogeneous components. In today's computer era, it is common to find systems of systems where each system was developed in a different language. In such environments, it is a challenge to estimate the change propagation between components that are developed in different languages. There is often no direct code dependency between these components, and they are maintained in different development environments by different developers. In addition, it is the domain experts and consultants who raise the most of the enhancement requests; however, using the existing change impact analysis methods, they cannot evaluate the impact and cost of the proposed changes without the support of the developers.

This thesis seeks to address these problems by proposing a new approach to change impact analysis based on software domain-level information. This approach is based on the assumption that domain-level relationships are reflected in the software source code, and one can predict software dependencies and change propagation by exploiting software domain-level information. The proposed approach is independent of the software implementation, inexpensive to implement, and usable by domain experts with no requirement to access and analyse the source code.

This thesis introduces *domain-based coupling* as a novel measure of the semantic similar-

ity between software user interface components. The hypothesis is that the domain-based coupling between software components is correlated with the likelihood of the existence of dependencies and change propagation between these components. This hypothesis has been evaluated with two case studies:

- A study of one of the largest open source enterprise systems demonstrates that architectural dependencies can be identified with an accuracy of more than 70% solely based on the domain-based coupling.
- A study of 12 years' maintenance history of the five subsystems of a significant sized proprietary enterprise system demonstrates that the co-change coupling derived from over 75,000 change records can be predicted solely using domain-based coupling, with average recall and precision of more than 60%, which is of comparable quality to other state-of-the-art change impact analysis methods.

The results of these studies support our hypothesis that software dependencies and change propagation can be predicted solely from software domain-level information. Although the accuracy of such predictions are not sufficiently strong to completely replace the traditional dependency analysis methods; nevertheless, the presented results suggest that the domain-based coupling might be used as a complementary method or where analysis of dependencies in the code and documents is not a viable option.

Chapter 1

Introduction

By 2020, more than 60% of software programmers will be working on software maintenance.

Applied Software Measurement, Global Analysis of Productivity And Quality [91]
Capers Jones

This thesis is a contribution to software evolution and maintenance. Software maintenance is often considered as keeping a system functional without changing its design or any major functionality. However, software does not deteriorate, wear, tear or break as a result of the usage or passage of time. Software repair actually involves fixing errors of implementation with respect to the design. Most enterprise software systems operate in domains such as finance, human resource and administrations where business rules change all the time. For these systems to function properly and avoid software ageing [146], it is required to change their functionality in respect to changes in their environment.

Change in the software environment often leads to new software functionalities. The IEEE standard 1219 [86] defines software maintenance as the correction of errors and modifications needed to allow an existing system to perform new tasks, and to perform the old ones under the new conditions. Therefore, “Software Maintenance” and “Software Evolution” are often used interchangeably.

Frequent change in requirements is the nature of enterprise domains and challenge for software maintainers. The connections between software elements make this challenge even harder as a change to a part of a system might lead to failure or inconsistency in other parts. Such a phenomenon, known as change propagation [157], has been tackled by software maintenance community for more than two decade. However, most approaches have been targeting dependencies in software source code and design documents [35, 21, 80, 73, 66].

The problem is that dependency analysis is not a viable option in many enterprise software environments. Large-scale enterprise systems often include heterogeneous source code (e.g VB and Python) whilst most code analysis tools support C++ and Java. Also, in many cases custom development is required to use these tools for analysing systems with hybrid or complex architecture. These challenges make code analysis a costly method that requires advanced skills beyond the knowledge of typical software developers. Dependency analysis based on design documents for most software systems is even more difficult than code analysis, since, missing or outdated documents are common problems.

For typical enterprise systems, there are domain experts who accumulated the knowledge of system functionality. These experts are the primary requestors for the software new features and changes in requirements. Without an understanding of costs and impacts of the requested changes, these experts cannot effectively collaborate with software maintainers to evaluate the trade off between the cost and the benefit of the prospective changes. This issue potentially can increase the cost of changing software, and lead to unsatisfactory software functionality.

Imagine a tool which enables domain experts to estimate change impact without requiring technical knowledge of software engineering and without access to the source code. If such estimations could be derived by domain experts, and were sufficiently accurate, then this would assist software maintainers to save time and effort in making decisions about prospective changes.

This thesis introduces a new methodology for software change impact analysis based on only software domain information. This work is based on the hypothesis that change propagation results of additive or corrective changes can be predicted using only domain information visible to software end users.

This research project focuses on data driven enterprise systems and management information systems (MIS). The scope of this thesis is limited to domain related software changes such

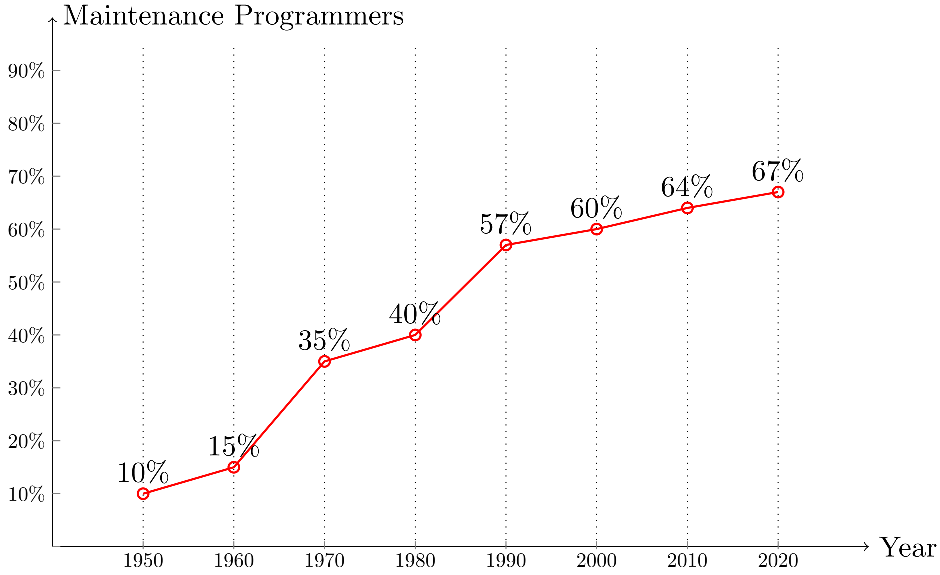


Figure 1.1: Percentage of Maintenance Programmers at Ten-year Intervals.

as changes to business constraints, and ignores software changes which do not change the functional properties of the system such as refactoring.

The rest of this chapter is organised as follows: The next section discusses the rationale behind this research. Section 1.2 presents the research questions, Section 1.3 shows the contributions of this thesis, and Section 1.4 describes the organisation of this thesis.

1.1 Rationale

The more software intensive systems blend in day to day human life, the more difficult they will be to replace or redevelop. Capers Jones in his well-known book “Applied Software Measurement, Global Analysis of Productivity And Quality” suggests that by 2020, more than 60% of software programmers will be working on software maintenance [91]. Figure 1.1 summarises the data presented by Jones about approximate percentage of world programmers working on software maintenance between 1950 and 2020. He argues that there is evidence of a critical phenomenon which occurs when an industry approaches 50 years of age, which is more workers perform maintenance jobs than build new products. Figure 1.1 suggests that in the software industry this turn over happened between 1980 to 1990.

Change in the software environment and consequently in requirements is a vital aspect of the software life cycle for most software systems [158, 12]. In order to keep a software application functional, it must evolve with respect to changes in its environment. However, the cost of change is often disproportionately high because of inadequate change impact analysis tools and techniques. In addition, manual change impact analysis performed by searching the source code or design artefacts is a tedious and labour intensive job.

During enterprise practices, it has been observed that users and domain experts are the main requesters for software changes and enhancements. However, it is difficult for domain experts to estimate potential side effects of requested software changes as existing change impact analysis methodologies require in-depth knowledge of software source code and understanding about software architecture. It is not even a trivial task for software engineers to recognise the scope of change propagation. In the literature, there are three main approaches to change impact analysis:

- Firstly there are document-based methods which rely on tracing dependencies in the design artefacts [76, 196, 167, 27, 46]. This approach has been one of the earliest forms of impact analysis, and provides great flexibility and accuracy in identifying where and how a change would affect the system. However, it is not practical for systems where the design artefacts are not accessible or reliable, such as legacy systems whose documents do not reflect many enhancements and ad-hoc developments.

- Secondly there are code-based methods which trace dependencies between system elements in the source codes [61, 181, 35, 37, 139]. This approach can be automated, and can be fairly accurate. However, it is often difficult to apply to hybrid systems with heterogeneous source code (e.g. parts of the system are in C++ and parts in Perl), or legacy systems with missing source code. In addition, these methods are complex and rarely usable by domain experts. Consequently, code-based impact analysis discourages any contribution from consultants, domain experts or managers.

- Finally there are history-based methods, which use maintenance history records to find dependencies between system components [168, 194, 77, 200, 56]. These methods are based on the assumption that if two components are frequently modified in a close timeframe and by the same programmer, it is likely that there is a relationship between them, and changing one of them might require alteration of the other one. This approach is reasonably time efficient and simpler than source code analysis. However, it is not as accurate as code-based or document-based methods, and not practical for systems which are in the initial development stage, or where the maintenance history is inaccessible.

The state of art research in the area of change impact analysis is focused on increasing the accuracy of change propagation analysis in order to reduce the risk of software failure and increase the reliability of systems. However, the provided methods usually require technical expertise, understanding of the software source code, or even tools specified for different architectures. Such factors make these approaches difficult to use for typical enterprise systems. What is needed is a pragmatic and inexpensive methodology for change propagation analysis, which conforms to the following criteria:

Simplicity and usability: The proposed methodology should be simple and usable by non-technical domain experts who have limited understanding of software source code. Simplicity of the proposed methodology reduces the required skills for change impact analysis, and might decrease the time spent by software maintainers on making decisions about prospective changes.

Practicality: The proposed methodology should be applicable to typical enterprise systems. It is common for such systems to have outdated design artefacts, heterogeneous source code and inaccessible maintenance history. These factors have to be considered for a change impact analysis methodology to make it usable for mainstream enterprise systems.

Generality: In order to make the proposed methodology work for general enterprise systems, it should not require a specific tool that is dependent to a particular programming language, software architecture, framework and implementation technology.

Efficiency: The proposed methodology should provide a sufficiently reliable estimation of the change impact in an acceptable timeframe with respect to scale and complexity of the system.

In order to achieve a pragmatic method, which conforms to these criteria, we need guidelines. The next section provides the guidelines in the form of research questions which provide us with goals for each stage of this research project, and assist us to evaluate the outcome.

1.2 Research Questions

The following are the research questions that we investigate in this thesis. These questions specify the requirements and expected outcomes of this research project:

RQ1. *What kind of model can we derive from domain experts' knowledge about relationships between software elements?*

What kind of information can be used to develop such a model?

What sources can be used to collect the required information?

This question is important because a model of the relationship between software elements is the prerequisite to many change propagation analysis methods.

RQ2. *How accurately can we identify architectural dependencies using such a model?*

It is a common understanding that change to one component might affect other architecturally dependent components. Ability to identify these dependencies without access to the source code is an important step towards the *generality* and *practicality* of the proposed method. This is specifically vital for software environments where conventional code analysis tools are not usable such as systems with hybrid source codes.

RQ3. *How accurately can we predict change propagation using such a model?*

The answers to this question is important to evaluate *efficiency* of the proposed method.

RQ4. *How does such a prediction compare with the well-established co-change coupling derived from maintenance history?*

It is essential to evaluate the *efficiency* of the propose method against the well established methods in the literature.

RQ5. *What is the required effort and cost of making the prediction?*

How well can we reduce the cost using tool support?

What is the trade off between cost and accuracy?

These questions evaluate the *usability* and *efficiency* of the proposed methodology.

The next section summarises the outcome and contributions of this thesis, derived from the proposed research questions.

1.3 Research Contributions

The main achievement of this thesis is a pragmatic method for change impact analysis based only on information visible and understandable to domain expert users. The benefit of this method is to provide an adequate estimation of a change impact on software functionality independent of software implementation. This method is applicable to software environments where tracing dependency based on source code, design artefacts and maintenance history is not accessible.

In this thesis, we answer the proposed research questions as follows:

- To answer RQ1 we propose a novel methodology for analysing software systems at the domain level, creating a model of relations between software elements, and demonstrating how such a model can be used for predicting the change propagation. The benefits of the proposed methodology are:
  - This methodology is agnostic to software implementation; therefore, it is applicable to software environments where source code analysis is not available such as systems with heterogeneous source code.
  - This methodology is not dependent on the software maintenance history; therefore, it is applicable to systems with inaccessible maintenance logs such as systems at their initial development stage.
  - This methodology is usable by non-technical domain experts who do not have access to software source code. As such this approach enables domain experts to predict the impact of proposing software changes without the support of programmers or software engineers.
- To complete the proposed methodology, we introduce the domain-based coupling as a novel metric for measuring the semantic similarity between software components at the domain-level. This metric allows us to capture, analyse and visualise relationships between these components.

- We answer RQ2 by providing the results of a case study on ADempiere; a large scale open source ERP[1] system, where we demonstrate how domain information can be used to predict source code dependencies and database relationships which might lead to change propagation.
- We answer RQ3 and RQ4 by a case study on BEIMS[2], a significant size enterprise system. In this study, we examine the software maintenance history in comparison to the domain-based coupling between software components. The results demonstrate how the domain-based coupling is correlated with the well-established evolutionary coupling derived from maintenance history. In addition, we will present how domain-based coupling can assist programmers to avoid software bugs arising from imperfect change propagation.
- We answer RQ5 by investigating the effort required for various tasks as part of domain-based analysis. Then we discuss the opportunities for automating the process and how a tool support can improve the speed and accuracy of the domain-based coupling analysis.

The enterprise case studies support the hypothesis that software dependencies and change propagation can be predicted solely from software domain-level information. The results suggest that the domain-based coupling can be used as a complementary method where analysis of dependencies in the code and documents is not a viable option.

In addition, this thesis examines the cost and accuracy of the domain-based coupling analysis, and it presents a semi-automated approach that provides an inexpensive analysis process with an accuracy comparable to state of art change impact analysis methods.

1.4 Thesis Organisation

The rest of this thesis is structured as follows: Chapter 2 discusses the background and related work. Chapter 3 addresses the first and second research questions by introducing domain-based coupling and a methodology for domain-based analysis. Chapter 4 addresses the third research question by a case study on an ERP system, where it is demonstrated that how architectural dependencies could be derived from domain information. Chapter 5

[1] Enterprise Resource Planning

[2] Building and Engineering Information Management System.

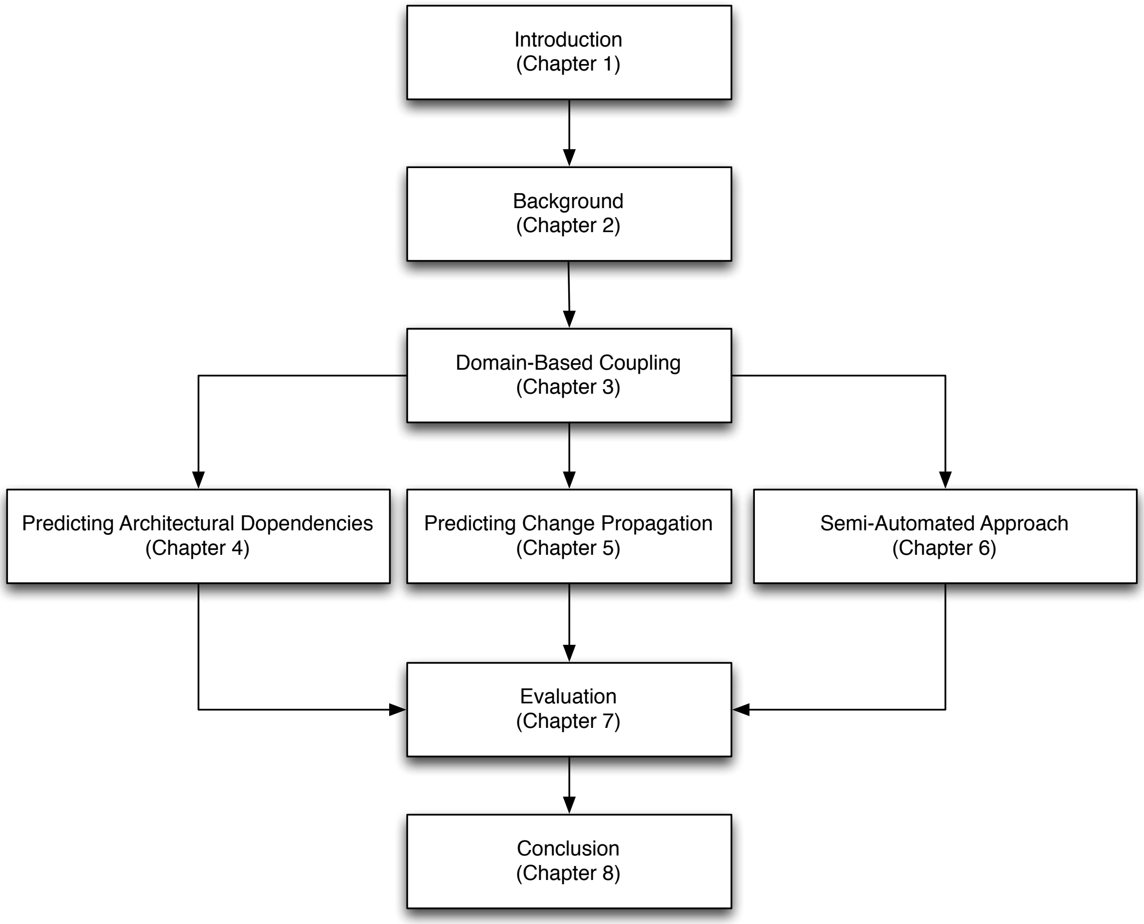


Figure 1.2: The logical sequence of the chapters of this thesis

addresses the fourth research question by providing the results of a case study on a large scale enterprise system where the domain information is used to predict the change propagation. Chapter 6 addresses the fifth research question by representing a semi-automated process for domain-based coupling analysis. Chapter 7 evaluates the outcome and the contributions of this thesis and discusses the limitations of the proposed methods. Finally, Chapter 8 summarises the research contributions, and presents the future areas of investigation. The structure and logical sequence of the chapters of this thesis are illustrated in Figure 1.2.

Chapter 2

Background

> Evolution requires expertise that is equivalent to or perhaps even greater than the expertise required to create a program from scratch.
>
> *A staged model for the software life cycle* [158]
> Václav T. Rajlich and Keith H. Bennett

This thesis is a contribution to the field of software evolution and maintenance. In particular, it provides a novel approach for predicting change propagation based on software domain information. This chapter provides an overview of software maintenance, its relationship to software evolution, and their role in the software life cycle (SLC). The phenomenon of change propagation and the practice of change impact analysis are described. Furthermore, a survey of various impact analysis methods is provided.

Figure 2.1 shows the logical sequence of this chapter's sections. The rest of this chapter is organised as follows: Section 2.7.3 provides an overview of software evolution and maintenance. Section 2.2 discusses the taxonomies of maintenance activities. Section 2.3 introduces a classification of software types based on evolutionary characteristics, and introduces the E-Type systems as evolving software. Section 2.4 describes the laws of software evolution, and Section 2.5 discusses how they impact the SLC. Section 2.6 introduces the phenomenon of change propagation, and Section 2.7 describes the practice of impact analysis. Sections 2.7.1, 2.7.2 and Section 2.7.3 provide a survey of existing methods on impact analysis methods. Section 2.8 provides an overview of domain engineering, and finally Section 2.9 summaries

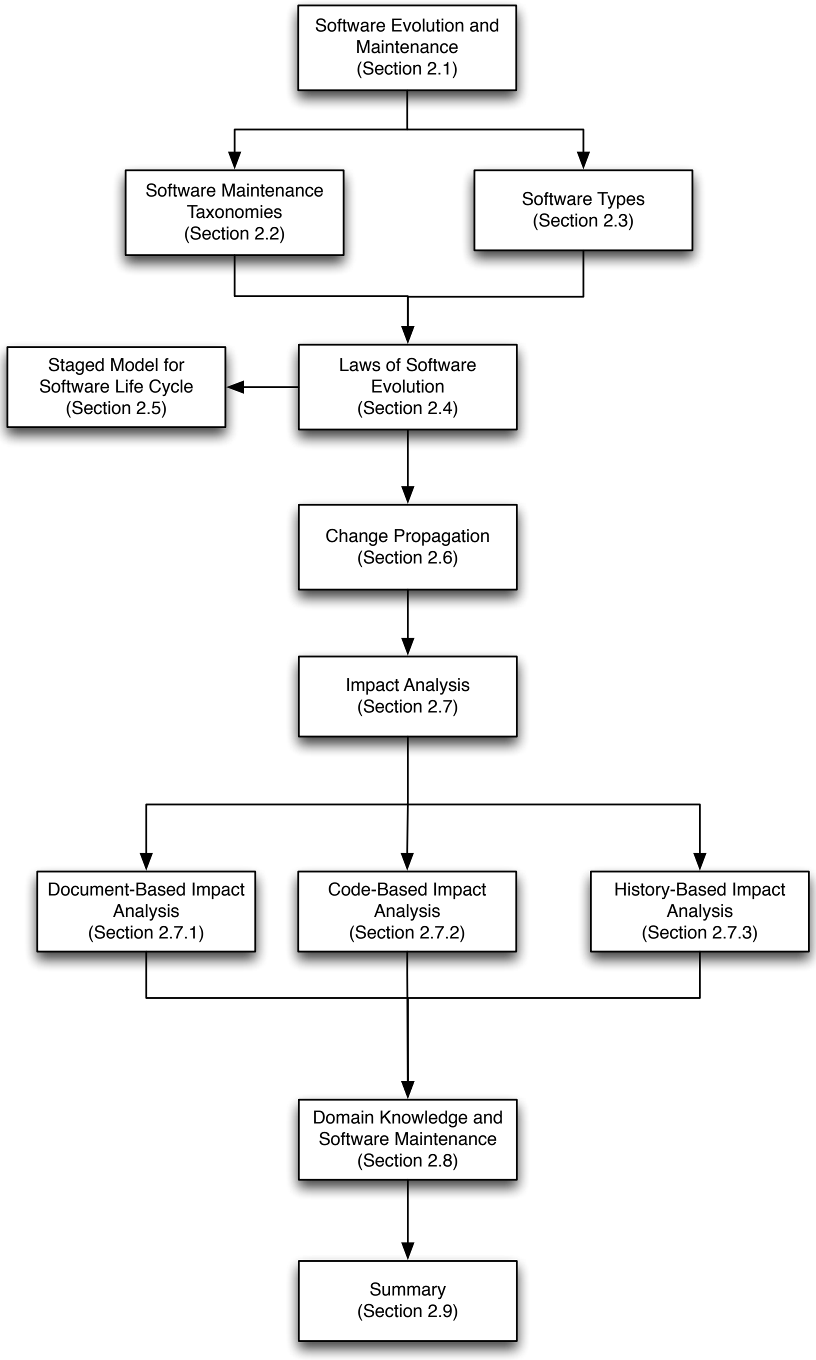

Software Evolution and Maintenance (Section 2.1)
Software Maintenance Taxonomies (Section 2.2)
Software Types (Section 2.3)
Staged Model for Software Life Cycle (Section 2.5)
Laws of Software Evolution (Section 2.4)
Change Propagation (Section 2.6)
Impact Analysis (Section 2.7)
Document-Based Impact Analysis (Section 2.7.1)
Code-Based Impact Analysis (Section 2.7.2)
History-Based Impact Analysis (Section 2.7.3)
Domain Knowledge and Software Maintenance (Section 2.8)
Summary (Section 2.9)

Figure 2.1: The logical sequence of sections of this chapter

this chapter.

2.1 Software Evolution and Maintenance

This thesis is a contribution to software evolution and maintenance. Hence, this section describes software maintenance, software evolution, and how they are related to the practice of software engineering.

Software ageing is not a new phenomenon, and the practice of software maintenance is well known as part of the SLC [146]. Maintenance is often considered as keeping a system functional without changing its design or any major functionality. However, software does not deteriorate, wear, tear or break as a result of the usage or passage of time. Software repair actually involves fixing errors of implementation with respect to the design; in addition, for most software, the environment is continuously changing, and for software to function properly, it is required to change its functionality in respect to changes in its environment. Hence, terms “Software Maintenance” and “Software Evolution” are often used interchangeably. However, to be more precise, in the literature these terms are defined as follows:

- *Software Maintenance* is defined by the IEEE standard of software maintenance, IEEE 1219 [86], as the correction of errors and modifications needed to allow an existing system to perform new tasks, and to perform the old ones under the new conditions [100].
- *Software Evolution* is defined by Belady and Lehman [10] as the dynamic behaviour of software systems as they are maintained and enhanced over time [100].

The above definitions indicate that software evolution is derived by maintenance activities which are enforced by changes in the software environment. However, software evolution can be affected by architectural properties and quality factors such as interoperability [87]. Godfrey and German compare software evolution with biological evolution [71]. They have demonstrated how software source code and software systems can be compared respectively with genotype and phenotype, two notions in biological evolution. They argue that despite differences between software and biological systems, in the software world, we can observe evolutionary mechanisms that encourage changes such as requests for new features, or to create new platforms, and the desire to improve quality attributes. In addition, there are forces which limit the change such as system complexity and legal concerns. The balance between the evolutionary mechanisms and the limiting forces, guides the SLC. The impact of software evolution on software development and the SLC will be further discussed in Section 2.4.

The cost of software evolution and maintenance has been one of the prime concerns in the SLC. It is commonly accepted by the software community that 70% of software expenditure is on software maintenance and 30% on new development. In the literature there are a number of empirical studies which examine the cost and processes involved in maintaining open source or proprietary systems [10, 100, 72]. These observations confirm that maintenance expenses are 60-80% of the initial development. Lehman [114, 119] argues that the high ratio of effort spent on maintenance to initial development does not necessarily have to be depreciated as maintenance covers a wide range of activities, and also continuous change is intrinsic to the nature of computer usage. However, programs should be more alterable, and the unit cost of change must be made as low as possible.

Software maintenance includes a wide range of activities. These activities have been focus of a number of studies, resulting in various software maintenance taxonomies. The next section explains the taxonomies relevant to this research, and discusses various types of maintenance activities.

2.2 Software Maintenance Taxonomies

In the literature, there are three well-known classifications for maintenance activities. These classifications are based on maintainer intentions, evidence, and characteristics of the change. This section presents these classifications and describes how they are related to the scope of this thesis.

It was Swanson and Lientz [172, 118, 117] who categorised software maintenance changes into four types :

Corrective are software changes in response to failure and software bugs such as performance failure, and process failure.

Adaptive are software changes in response to changes in data and the process environment. This includes all enhancements to make software function properly in a new environment.

Perfective are software changes which are intended to improve the system such as eliminating process inefficiency, enhancing performance and improving maintainability.

Preventive are software changes intended to avoid future maintenance problems, for example, restructuring internal dependencies to improve cohesion and coupling.

In this view, perfective and preventive software changes are intended to not change the functionality of the system whilst corrective and adoptive changes explicitly alter system functions in response to new requirements. This thesis focuses on changes derived directly by new environments or changes in existing requirements; hence, it only considers corrective and adaptive changes, and ignores the perfective and preventive changes.

The proposed categorisation by Swanson and Lientz, considers maintenance activities from perspective of software developerswhilst others proposed taxonomies from alternative perspectives. Chapin et al. [34] extended this categorisation in to twelve evidence-based classifications. In this view, the types of maintenance activities are defined using a hierarchical evidence of changes as follows.

- *Changes to software:*
    - *Changes to source code:*
        - *Changes to software functions:* Enhancive, Corrective and Reductive
        - *Changes to properties:* Adaptive, Performance, Preventive and Groomative
    - *Changes to non-code artefacts:* Updative and reformative
- *Changes to software environment:* Evaluative, consultive and training

This classification considers the observed activities and changes to software artefacts rather than the intentions of maintainers. This thesis only focuses on maintenance activities which lead to changes in software functions; hence, only considers enhancive, corrective and reductive activities.

Whilst the earlier research categorised maintenance activities based on their purpose, Buckley et al. [31] propose a taxonomy for maintenance activities based on the mechanisms of the change and the factors that influence these mechanisms. This view focuses on technical aspects, i.e., the how, when, what and where, of software changes, and derive a number of dimensions of software change mechanisms: time of change, change type, change history, degree of automation, activeness, change frequency, anticipation, artefact, granularity, impact, change propagation, availability, openness, safety and degree of formality. This taxonomy is an extension of prior work on software maintenance ontology by Kitchenham et al. [102]. They described ontology of software maintenance terms in the form of a UML model, aimed to identify factors which might affect the empirical studies in software maintenance. In this classification, this thesis only concerns change impact and change propagation.

The rules and dynamics of software evolution and maintenance activities are not the same for all systems [114]. The next section discusses a software classification based on evolutionary characteristics.

2.3 Software Types

The notion of software evolution is mainly associated with a specific software type known as E-Type [111]. This section describes E-Type software, and describes how it is distinguished from other software types.

It has been a view that software evolution is associated with large systems. In this view, a system is considered large if it includes more than an arbitrary number of source code lines. Lehman [113] was critical of this view on the grounds of its arbitrariness, and he believed large systems should be identified by the ways in which they are designed, developed and maintained. To address this concern, Lehman proposed a new classification [114] based on the realisation that there is a fundamental distinction between the evolution of systems which are implemented from a formal specification, and the ones that are developed to be part of day to day activities. He proposed three software types[1] as follows:

S-Type Programs: Lehman defined a program as Type S "if it can be shown that it satisfies the necessary and sufficient condition that it is *correct* in the full mathematical sense relative to a pre-stated formal *specification*" [111]. This definition assumes that

[1] Lehman [111] uses *program* and *software* interchangeably with more emphasis on programs. In this section, we follow his definitions.

the problem (requirement) can be formally *specified* prior to the implementation, the problem can be solved using an algorithmic method, and it is feasible to prove that the program is correct against the formal specification. These assumptions limit the domain of S-Type programs to mathematical applications, or formally defined transformations such as compilers.

E-Type Programs: Lehman initially defined a program as Type E if it mechanises a human or societal activity [114]. This definition was subsequently amended to "all programs that operate in or address a problem or activity of the real world" [111].

A characteristic of E-Type programs is their integration in a domain. Changes in their domain raise new requirements for these programs and necessitate their evolution with respect to their environment. Hence, software *evolution* is a direct consequence of the nature of E-Type programs, and one can not expect them to remain static.

P-Type Programs: Lehman defined this class as an intermediate between S-Type and E-Type [111]. The programs in this class address problems that can be fully specified, but the users are concerned with the execution results rather than validating the implementation against its specification.

An example of this type is a program that plays chess. The rules of the game can be fully specified; however, the decision tree at any given stage of the game is too large to be scanned by a personal computer, hence, the program must provide an optimum approximation of a good decision given the limited resources. A chess program is valued by its performance, not by validation against the specification.

Cook *et al.* refined this classification with an emphasis on the role of stakeholders in the evolution of system requirements [41]. Their classification is derived from the Kuhn's concept of normal science [105] and the concept of paradigm [132]. Kuhn explains that development of scientific knowledge consists of successive periods of what Kuhn called "normal science" that each take place within a paradigm [41]. In this view, a paradigmatic domain contains a stable and well structured body of knowledge. This implies that an analyst must use methodological hermeneutics[2] and the baseline model of the domain to validate the requirements. In contrast, non-paradigmatic domains lack such a rigid knowledge structure, and consequently the requirements are open to objective interpretation.

[2] The hermeneutics tradition in philosophy studies the process of interpretation [41].

Cook *et al.* [41] argue that E-Type programs are situated in non-paradigmatic domains. In such domains, sources to derive domain knowledge are less extensive and less reliable, and validation of requirements often wholly relies on interpretation of stakeholders' statements. This implies that stakeholders can define and redefine problems without any paradigmatic constraints, and the scope of the system is open to reinterpretation.

Type-P has been redefined as "Paradigm-based" programs which address problems in paradigmatic domains [41]. The evolution of these programs is restricted to changes in their paradigms, and the change to the system is constrained by the stakeholders' decision to keep the system consistent with the domain.

Type S programs are somewhat different. Cook *et al.* [41] argue that these programs do not evolve. Once the requirements are specified, then these programs should detach from the paradigm, and they will no longer be affected by the changes in their domain.

This thesis does not consider S-Type and P-Type programs, and it focuses only on E-Type programs where lack of rigorous specification makes the validation of software changes, a challenge for software maintainers. In these systems, often the implemented program is the only actual model which can provide reliable information about the potential impact of software changes. The uncertainty in requirements for E-Type systems makes them the default case of software evolution.

The laws of software evolution and how they are applied to E-Type systems will be further discussed in the following section.

2.4 Laws of Software Evolution

Lehman observed that E-Type programs must evolve in respect to changes in software domain or they risk an early death [114]. Based on an empirical study of IBM programming process in late seventies, Lehman defined five laws of software evolution [114], later extended to eight [112, 115] as presented in Table 2.1.

Lehman [113] explained how E-Type programs continuously adapt and grow (Law I and IV), and how such changes can increase complexity (Law II) and lower the quality of the system (Law VII). This phenomenon emphasises the challenge of evolution in the SLC and highlights

	Law Name	**Description**
I	Continuing Change	E-Type programs must be continually adapted else they become progressively less satisfactory.
II	Increasing Complexity	As an E-Type program evolves, its complexity increases unless work is done to maintain or reduce it.
III	Self Regulation	E-Type program evolution process is self regulating with distribution of product and process measures close to normal.
IV	Conservation of Organisational Stability	The average effective global activity rate in an evolving E-Type program is invariant over product lifetime.
V	Conservation of Familiarity	As an E-Type program evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution.
VI	Continuing Growth	The functional content of E-Type program must be continually increased to maintain user satisfaction over their lifetime.
VII	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII	Feedback System	E-Type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 2.1: Lehman's laws of Software Evolution [114]

the role of managing software changes in the longevity of E-Type programs.

The other four laws (Table 2.1) present the various aspects of the software evolution. Lehman argues that the software evolution is based on a feedback process (Law VIII), and for E-Type programs which are implemented in an organisation, the positive and negative feedback by the corporate management regulates the evolution of the system (Law III). The ability of the organisation to manage the software evolution is limited by forces such as availability of skilled staff and limited resources. These limitations make the average activity rate for a system constant over its lifetime (Law IV). In addition, the development team and all other associates should maintain their knowledge about the system during the process (Law V).

Lehman's laws laid a foundation for successful software maintenance and evolution, emphasising the necessity of continuing changes of E-Type programs. The next section describes how continuing software changes can be modelled in the SLC.

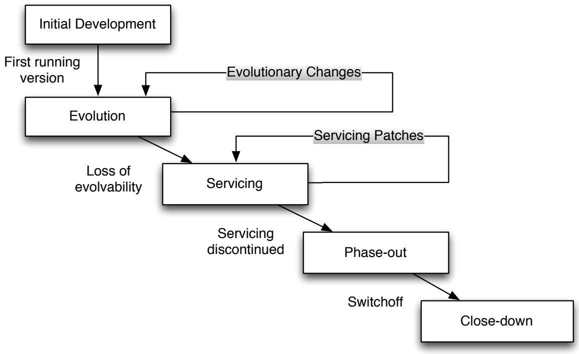


Figure 2.2: Staged model for the software life cycle [158]. Used with permission.

2.5 Staged Model For The Software Life Cycle

It is important to incorporate maintenance activities in the SLC. Mens *et al.* describe such an integration as a challenge in software evolution and maintenance [134]. They proposed the iterative and incremental software development (well-known as agile software development or extreme programming [9]) as a typical way to incorporate continuous change in the traditional software development process. The software maintenance activities can be integrated in to the SLC as a set of tasks including determining the maintenance objective, program understanding, maintenance planning and implementation [193]. This model considers any work after initial development as software maintenance. However, given the rapid software evolution, the traditional approach is no longer sufficient [12, 134]. Rajlich and Bennett [158] proposed a staged model (Figure 2.2) for the SLC which supports evolution of software in five distinct stages as follows:

- *Initial development:* The first stage of the SLC which aims to deliver the first version of the program. During this stage the programming team acquire a great deal of knowledge about the software domain such as requirements and business processes. Two primary outcomes of this stage are software architecture and development team knowledge.
- *Evolution:* The second stage of the SLC takes place once the first version of the software is completed. The aim of this stage is to adopt the new requirements and changes in the software environment.

- *Servicing:* During this stage only small changes (patches) are possible. The primary reason for moving from the evolution stage to the servicing stage is the loss of development team knowledge, leading to loss of software architecture coherence.
- *Phase-out:* During this stage no more servicing will be performed, but the system is still in production.
- *Close-down:* During this stage the system will be disconnected, and the users will be directed to a new system.

This model can be extended (Figure 2.3) for multiple versions of the software system. The developing team can create new branches of the system from existing source code leading to major changes in functionality and architecture. After creating each branch, and its release it will be stable (will not further evolve) and mostly serviced by minor enhancements and bug fixes.

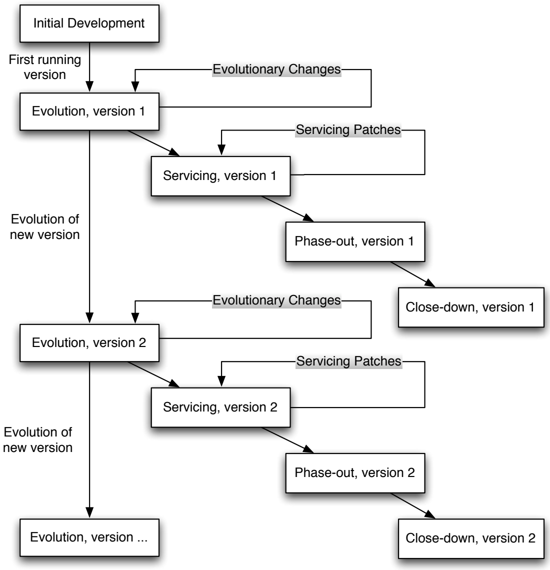


Figure 2.3: Versioned Staged model for the software life cycle [158]. Used with permission.

The staged software life cycle conforms to Lehman's laws of evolution by supporting continuous changes (Law I) in evolution and servicing stages. The evolution stage often incorporates major growth in software functionality (Law VI) which eventually leads to a new branch of software, also the growth in the evolution stage often leads to increased complexity (Law II).

Both the evolution and the servicing stages allow the feedback system where the development process is influenced by external forces (Laws III, IV, V, and VIII). Finally the decline in software quality (Law VII) happens in servicing and phase down stages before managers shut the system down and move its users to a new version or alternative systems at close down stage.

2.6 Change Propagation

As discussed in the previous section, E-Type software systems will be affected by continuous changes in their life cycle. However, as typical software systems are composed of connected parts, often change to one part of a system affects other parts, and leads to subsequent changes. Such a phenomenon, known as *change propagation* or *ripple effect*[3], can affect software development projects and maintenance activities by leading to unforeseen extra development costs. Thus, measuring the change propagation is of fundamental importance for the SLC.

The first law of software evolution describes continuous changes in the life cycle of E-Type systems. These changes and the potential change propagation can be used in various aspects of software maintenance. Black [19] argues that given the progressive changes in a system, change propagation measurements can be used to assess stability of the design and implementation. Also it can highlight highly volatile sections and candidate areas for restructuring (refactoring).

The second law of software evolution explains that the system complexity will grow unless a complexity control process is applied as part of the SLC. Complexity can be the result of architectural dependencies, or logical relationships that may or may not be visible at the source code level. These connections are often correlated with change propagation [179], and measuring change propagation can lead to a better understanding of complexity [192, 18].

Maintainability is one aspect of software quality [87, 69], and complexity can be negatively correlated with maintainability of a system [19]. Thus, complexity can lead to decrease in software quality. Black [19] argues that complexity control in the software life cycle can make a difference between survival or demise of the system.

[3] In the context of this thesis, for consistency we use the term *change propagation* instead of *ripple effect*.

Software engineering and maintenance communities have tackled the change propagation and growing software complexity for more than two decades. As a result, a number of design and development methodologies have been introduced to mitigate change propagation such as object oriented design [24], aspect oriented programming [101] and service oriented architecture [52]. The key characteristics of these methodologies are reducing the number of software dependencies, improving encapsulation and the separation of concerns [173]. However, even the most well designed software systems will still face some degree of change propagation during their evolution and maintenance. In the next section, we describe methods for measuring and managing the change propagation phenomenon.

2.7 Impact Analysis

The interdependent nature of software system composition means that any change to one software component often necessitates flow-on changes to other components. This effect is known as *change propagation* or *ripple effect* [157]. Hassan and Holt [79] describe change propagation as the central aspect of software development . When programmers change a software element, like a function or a variable, they have to search for other related elements and update them to avoid inconsistency. This is not a trivial task, and many software bugs are created by programmers who failed to properly propagate the change. It has been argued that the ill-effects of imperfect change propagation lead to the software ageing, and cause software failures and unforeseen extra development costs [146]. Thus, measuring the effects of change propagation plays an important role in the larger picture of the software development life cycle [157, 193].

The activity of identifying what to modify to accomplish a change, or of identifying the potential change propagation is *impact analysis* [3, 178, 153]. It is common to model change propagation based on software dependencies. In the literature, several formal models of change propagation and impact analysis have been explained. Luqi [125] presented a graph model for impact analysis, based on components and evolutionary steps. The proposed model uses a formal definition of indirect relationships between components. Rajlich [155, 156] introduced a model for change propagation based on graph rewriting, which uses a sequence of snapshots, with each snapshot representing a particular phase in the change propagation process. Predicting change propagation using this model requires an understanding of de-

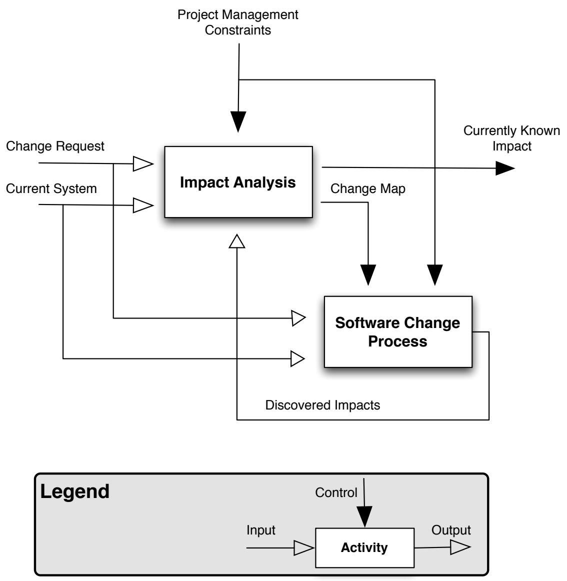


Figure 2.4: Impact Analysis Cycle by Shawn Bohner and Robert Arnold [22]. Used with permission.

pendencies between software elements such as classes and functions. Arnold and Bohner [22] describe change impact analysis as a cycle (Figure 2.4) whereby a user, analyst, or programmer submits a change request for approval. If the change is approved, it will be passed for *impact analysis* which might include analysis of the source code, design artefacts, or even test materials. The result is a change map based on relationships between software elements. The next stage is *software change process* where programmers change the software elements according to the change map. This stage can lead to discovery of a new set of affected elements. This cycle continues until the change requirements and the expected software quality [87] are satisfied.

In the literature, we identified three major impact analysis approaches:

- *Document-based approach:* This approach relies on tracing dependencies in design artefacts. This approach is based on the assumption that a model of software dependencies can be derived from design documents such as UML diagrams, and such a model can be used for impact analysis.
- *Code-based approach:* This approach is based on tracing dependencies between system

elements in the source codes.

- *History-based approach*: This approach uses the maintenance history to find the likelihood of the change propagation between system elements.

In the next three sections we describe these approaches in detail, and provide a summary of the related work.

2.7.1 Document-Based Impact Analysis

Tracing software dependencies based on the documented software model is a traditional approach to impact analysis. Pioneer methodologies for impact analysis took the assumption that the knowledge of software dependencies is available in the forms of design documents or dependency graphs. They include the Prism's process model of change management [126], frameworks for impact analysis [3, 23, 21], formal models for impact analysis [125, 76, 155] and a model for change propagation based on graph rewriting [155].

Model-Based Methods

It is common to model software systems using UML diagrams, and a number of impact analysis methods focuses on tracing software dependencies based on UML diagrams. Change propagation within UML models has been investigated in the areas of inconsistency resolution [122, 141, 20], automated impact analysis in UML models [28], and generating repair plans for UML documents [46]. In addition, there is a direction of research in tracing dependencies in design artefacts such as UML models using information retrieval techniques [167, 123]. Another application of tracing dependencies in UML models is supporting change propagation in agent-based systems [47, 45, 44].

Requirement-Based Methods

Tracing relationships between requirements, is another aspect of change propagation [74]. Hassine *et al.* [80] described a method for impact analysis at the requirement level. Goknil et al. [73] proposed a change impact analysis method based on formalisation of relations in software requirements. Lee *et al.* [110] extended these methods to a goal-driven method for managing requirements traceability and impact analysis. In addition, the traceability between requirements, source code and other artefacts has been identified as a challenge in software maintenance [38, 1]

Discussion

The document-based methods can perform well where there are recent and adequate design documents. However, missing or outdated design documents is a common issue in enterprise software environments. Often the initial design was not well documented, and the evolution of the source code has not been reflected in the design documents, resulting in inadequate documents about software dependencies. In such cases, the actual implemented software is considered as the most reliable source of information about software functionalities. In the next section, we discuss impact analysis methods based on the source code and software implementation.

2.7.2 Code-Based Impact Analysis

In many software environments, the source code is the most reliable software artefact. Hence, code-based analysis is one of the most investigated impact analysis approaches. In the literature, we identified five kinds of code-based analysis methods which can assist in identifying the impact of software changes. These methods are classified as static dependency analysis, program slicing, clone detection, coupling analysis and dynamic analysis.

Static Dependancy Analysis

Tracing dependencies in the source code has been part of software maintenance from the early stages [185]. After the invention of object-oriented design, the code analysis methods have been extended to incorporate the new relations such as inheritance and polymorphism. Kung *et al.* [106] classified the different kinds of change types in object-oriented classes and their impacts. Later researchers extended this classification to a comprehensive list of change types and automatic methods for estimating change impact [120, 109, 171, 35]. Also there is a trend of research in detecting hidden dependencies in the source code based on abstract system dependency graph (ASDG) [196, 179].

Program Slicing

One of the most well-known code analysis methods is *program slicing*, which has been exhaustively explored by many researchers, and extended to many programming paradigms [16, 187, 190, 169]. Program slicing was initially introduced by Weiser [182, 183] with the aim of assisting programmers in program understanding and debugging.

Program slicing has been defined as a decomposition technique that omits unrelated program components to a select software element (referred to as a slicing criterion) [60]. This method has been used for software maintenance [62], and there are number of empirical studies in this area [15, 14, 17]

Clone Detection

It is common for developers copy and paste code fragments from one module to the other one. As a result, there are sections of code that are similar. These sections are called code clones. There are two kinds if similarity between code fragments: textual similarity (Type 1 to 3) [11], and functional similarity (Type 4) [55]. Roy and Cordy [165] describe these types as follows:

Type 1 Identical code fragments except for variations in white space, layout and comments.

Type 2 Syntactically identical fragments except for variations in identifiers, literals, types, white space, layout and comments.

Type 3 Copied fragments with further modifications such as changed, added or removed statements.

Type 4 Code fragments that perform the same computation but are implemented syntactically differently.

Previous research shows that a significant fraction (between 7% and 23%) of the code in a software system has been cloned [164]. These clones can lead to problems in software maintenance, as inconsistent change to the cloned codes results in unexpected software behaviour [92]. Hence, detecting these code clones has an important role in impact analysis.

There are four categories of clone detection techniques [165]: textual, lexical, syntactic, and semantic. Textual methods apply little transformation on the source code, and in most cases use the raw source code for comparison [89, 90, 50, 129]. Lexical methods are based on transforming the code to a set of tokens and applying lexical analysis [7, 98]. Syntactic methods convert the source code to the abstract syntax tree and use tree matching or structural metrics to find the clones [8, 88]. Semantic methods use static program analysis or a dependency graph, and find clones by searching for isomorphic subgraphs [103, 104]

Coupling Analysis

Coupling metrics show the degree of semantic or syntactic relationships between software elements like classes. Structural coupling in object-oriented systems has received notable attention in the literature. Briand *et al.* classified these metrics within the unified framework for object-oriented systems [26]. Metrics like the Coupling Between Objects (CBO) or the CBO' [40] consider the inheritance between classes to measure the coupling among software elements. Other metrics like the Response For Class (RFC) and the RFC_∞ [39] consider indirect relations among classes based on a level of indirection in the invocation chain of the class methods. There are empirical studies which demonstrate the application of these metrics in impact analysis [29, 186, 151, 68].

Dynamic Analysis

The static coupling between object-oriented elements has been extensively studied in the literature. However, because of polymorphism, late binding and hidden dependencies the static coupling metrics do not perfectly reflect the coupling between classes and objects. Intially dynamic object-oriented coupling measures were proposed by Yacoub *et al.* [191]. Arisholm *et al.* [2] extended this approach to dynamic coupling metric based on analysis of runtime objects' interactions. They demonstrated that dynamic coupling captures different properties to static coupling measurements.

There are automated tools and techniques for dynamic impact analysis including CoverageImpact [144], PathImpact [107, 108] and online impact analysis [25]. These methods are based on counting the number of methods before and after execution of a method, and slicing the execution trace to identify the impact of the change to the method. Orso *et al.* [145] performed an empirical comparison between these methods and their results show that PathImpact is more precise, while CoverageImpact is less computationally expensive.

A recent direction of research is focused on reducing the cost and improving the accuracy of dynamic impact analysis by utilising the special object-oriented characteristics such as inheritance [83, 84, 85].

Cornelissen *et al.* [42] provided a comprehensive survey of dynamic analysis methods and their application in impact analysis and program comprehension.

Discussion

The code analysis methods can be automated, and in most cases, their outcome is a highly accurate set of dependencies between code elements. However, there are three obstacles to implement these methods in enterprise environments:

- Not all of these dependencies lead to change propagation [65]. Therefore, other complementary methods [137, 197] are required to filter out irrelevant dependencies and improve the precision of impact analysis results.
- These methods are difficult to apply to systems with hybrid source code (*e.g.,* parts of the system are in C++ and parts in Perl), or legacy systems with missing source code.
- These methods typically demand high level of technical skills and tools. Such requirements limit the users of these methods to software engineers and skilled developers, and discourage collaboration of domain experts (*e.g.,* consultants and managers) in impact analysis.

In summary, code-based dependency analysis techniques are a strong part of software engineering and development; however, complementary methods are required to achieve an effective impact analysis approach in software maintenance. In the next section, we discuss an alternative impact analysis method based on maintenance history, and independent from tracing dependencies from code or design documents.

2.7.3 History-based Impact Analysis

An alternative approach to dependency analysis is mining co-change coupling from software repository. This approach is based on the hypothesis that the frequency of change propagation between software elements in the past indicates the likelihood of change propagation between them in future.

Several tools and methods have been proposed to assist programmers in maintenance activities by using maintenance history records. Mockus and Votta [138] designed a tool to classify maintenance activities based on the description of changes, aiming to provide better understanding about why the changes were performed. Chen et al. [36] introduced CVSSearch, a tool which enables programmers to search for code fragments based on comments recorded in source code repository. Cubranic and Murphy [43] introduced Hipikat, a tool which uses the maintenance history records to provide a group memory for newcomers to software development projects. Hassan and Holt [78] proposed annotation of architectural dependencies with information mined from source control system, aiming to provide better understanding of architectural dependencies. German et al. [66] introduced a hybrid approach to identify the impact of prior code changes and if they caused any software failure. Their method is based on annotating the functions' dependency graph using the history of code changes.

Kagdi et al. [93] provided a comprehensive survey of mining software repository (MSR) approaches, and introduced a multi dimensional taxonomy based on information sources, purpose, methodology and evaluation methods. From their survey, we can identify the following related approaches to change propagation and impact analysis: logical coupling, evolutionary coupling, heuristics for predicting change propagation, and change patterns.

Logical Coupling

In a trilogy on software release history analysis Gall et al. [59, 57, 58] sought to discover the semantic relationships between classes based on source code version history. They called such relationships logical coupling. This approach is further extended based on using bug tracking reports as a source of maintenance history [53].

Evolutionary Coupling

Zimmerman et al. [198] demonstrated how to perform fine-grained analysis on CVS repositories, and discover co-change relationships between source code elements. They called such a relationship "Evolutionary Coupling", and presented a tool support called Rose [199, 200]. The aim of the provided tool is to assist programmers in predicting change propagation between source code elements based on maintenance history. The Rose prototype was evaluated against eight open source systems where it correctly predicted 26% of further files to be changed, and 15% of the functions or variables. A similar approach has been taken by Ying et al. [194, 195] for co-change coupling analysis. They evaluated their method against Eclipse and Mozilla, and demonstrated that valuable dependencies can be derived from co-change coupling which may not be derived from other analysis methods.

Heuristics for Predicting Change Propagation

Hassan and Holt [77] proposed a number of heuristics for predicting change propagation. They proposed that change propagation can be predicted based on history-based co-change records, code structure relations, code layout and developer data. The performance of these heuristics is evaluated against five open source systems, and the outcome shows that the best precision and recall could be derived from history-based co-change coupling[4]. They argue that "Our results cast doubt on the effectiveness of code structures such as call graphs as good indicators for change propagation" [77].

In more recent papers, heuristics have been used to visualise and understand development stages [67], to create a meta-model for software evolution [70], and identify code ownership [81].

Change Patterns

Kagdi *et al.* [96, 95] applied sequential-pattern mining to discover files that are frequently changed together. Their method is distinguished from evolutionary coupling [198] by extend-

[4] In this thesis, the term "history-based co-change coupling" identifies a wide range of research on the coupling metrics based on software maintenance history while "evolutionary coupling" specifically refers to the coupling metric derived from fine-grained analysis of version control repositories (Zimmerman et al. [198] and Ying et al. [194, 195]).

ing the analysis of co-change coupling beyond source code elements. Their method discover the links between source code and other types of artefacts using the ordering information in which files were changed in a change-set (this might not recorded in source code repositories).

In other related directions of research, the sequential-pattern mining method has been used to visualise the hierarchical order of changes in software elements [32, 184], report API-usage patterns [189], identify code ownership [82] and detect patterns of user activities [51].

Discussion

The history-based analysis methods are typically well-automated and have low execution costs. However, there are obstacles to implement these methods in typical software environments: Firstly, specific technical knowledge is required to implement these methods and tools. This knowledge is not commonly available to developers of small to medium software systems yet. Secondly, in some software environments, the maintenance history is inaccessible such as in recently developed systems. Finally, the history-based methods are not as accurate as document-based and code-based methods. There are recent research efforts in the area of improving the accuracy of history-based impact analysis including blending conceptual and evolutionary coupling [94] and using variable granularity [149].

In summary, software maintenance history is a valuable source of information about software evolution and change propagation; however, given the state of the art in this area, the history-based methods might be expensive or impractical to implement in some software environments such as recently developed systems.

2.8 Domain Knowledge and Software Maintenance

This proposed approach in this thesis is based on using domain knowledge for change impact analysis. This section discusses the related domain analysis approaches, and describes the terminology in this area:

Domain Analysis has been defined as the activity of identifying the objects and operations of similar systems in a particular domain [140]. Domain analysis is distinguished from system analysis, in that it is not concerned with functions of a specific system. Instead, it is concerned with similar functions and behaviours which occur in all systems in a domain.

Domain Model is defined as the definition of the functions, objects, data, and relationships in a domain [99].

Domain Expert provides information about the domain model of a system, and supports domain analysis [99, 135].

Knowledge of program structure and functionality is vital for modifying software systems [133, 170]. All activities by which knowledge is gained about a program is called *program understanding*, and *reverse engineering* is a systematic form of program understanding which provides an abstract view of the system [166]. Such an abstract view can assist maintenance activities such as impact analysis.

As we discussed in Section 2.3, the purpose of E-Type systems is to address a problem in a domain, and domain knowledge is critical for understanding the design and implementation of these systems. Most of the program understanding methods are based on source code analysis; however, Brooks[30] connected the program understanding and domain knowledge for the first time in 1983. Few years later, Letovsky *et al.* [116] elaborated this idea and suggest that programmers must understand the intention behind code in order to carry out maintenance on it. Such understanding can be acquired from code annotation and comments [180], design artefacts [130], or from domain experts.

In 1994, Lindvall and Sandahl [121] conducted a study on an industrial software project to describe how impact analysis is performed by professional software engineers. They observed that in practice software engineers prefer to interview domain experts rather than consult documentation in order to trace objects that need to be changed. Their findings suggest that dependency analysis methods based on documentation, models and accompanying tools, are not the preferred approaches to impact analysis. Since then, domain knowledge has been incorporated into reverse engineering and program understanding by a number of researchers. Petrenko *et al.* studied how programmers collect partial knowledge during software comprehension and later use it to guide the concept location process [150]. Michail [136] discussed the possibilities of using the application GUI to guide browsing and searching of its source code. Rugaber [166] showed how domain knowledge can be useful in program compression. Riebisch [162] introduced feature models based on domain analysis to supporting software evolution. Feature modelling was initially developed to structure domain properties from customers' point of view [147].

Mapping domain concepts to source code has been recognised as an important task in software maintenance by number of researchers [13, 154, 131]. There is a direction of research in concept location by mapping the source code to the domain entities [160, 159, 161]. User cognitive abilities and their understanding about a system has been used to develop reverse engineering tools and methods [177, 175, 176], later this method has been proposed to support software change processes [188].

These methods mainly aim to assist programmers in activities of understanding and changing the source code. Since, the primary objective of these methods is to incorporate domain knowledge in to the development environment, they are difficult to use by consultants, managers, or expert users who have little knowledge about the source code. In contrast, the proposed method in this thesis is independent from software implementation, and aimed to be usable by expert users who might have little or no skills in software engineering.

2.9 Summary

In this chapter, we described the background and the work related to this thesis. This thesis contributes to the practice of impact analysis in software maintenance, and introduces a novel impact analysis method based on domain information. As such, the first four sections provided an overview of software maintenance, described how software evolution is an inevitable part of software maintenance, and how various software systems evolve. In addition, the laws of software evolution have been described and how they relate to the software life cycle.

The phenomenon of change propagation is introduced, and we described how it challenges software maintainers to estimate the cost and impact of software changes. We provided an overview of impact analysis approaches including document-based, code-based and history-based methods. Moreover, we discussed the significant impact analysis methods, their advantages and their drawbacks. Finally, an overview of domain analysis methods is presented, and the role of domain knowledge in software maintenance is described. In summary, we described the problem of change propagation that is addressed in this thesis, related approaches in the literature, and how we distinguish our proposed method from the related work.

Chapter 3

Domain-Based Coupling

> Any program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse.
>
> *Programs life cycles and laws of software evolution*[114]
> M.M. Lehman

This chapter introduces domain-based coupling as a measure of semantic similarity between software user interface components. We hypothesise that such a coupling measure can approximate software dependencies in the source code and the database, and it can be used to predict change propagation.

As we discussed in Section 2.3, the E-Type software systems (known as evolving type) are derived from domains where requirements are uncertain, and are likely to change during the software's lifetime [114]. Most enterprise systems belong to this category. Therefore, for these systems the domain experts are the primary source of information for evaluating requirements [41]. These domain experts drive software evolution by continuously asking for new functionality or requesting changes to existing ones. Unfortunately, domain experts are in a poor position to estimate the impact of the changes they request since they typically do not have inside knowledge of the internal dependencies of the software system.

Enterprise software systems are constructed to model business domains [114]. It is reasonable to expect that real-world dependencies are therefore reflected in the software itself. This chapter reports on two case studies which demonstrate the processes of domain analysis, measuring the domain-based coupling, and visualising such a coupling as weighted graphs. In addition, these studies describe how the derived graphs approximate the dependencies in the source code.

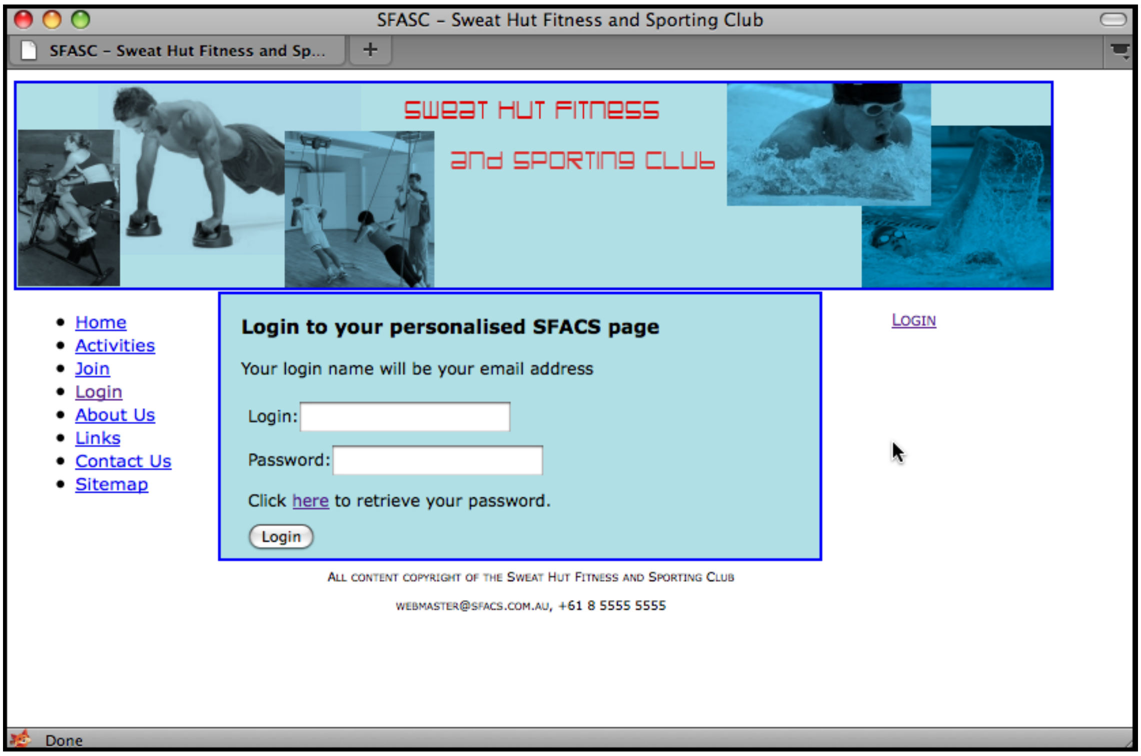
This approach is applicable to a subset of E-Type software which includes the data driven information system that provides most of their functionality through a number of user interface components, such as business applications, and management information systems (MIS). We apply the term information systems to such systems. In Section 3.6, we will further discuss the applicability of this approach to different software categories and types of software changes.

The rest of this chapter is organised as follows: Section Section 3.1 introduces a simple web application that we use for the demonstration of the proposed methodology. Section Section 3.2 provides the background on concepts and notations. We present the domain-based coupling analysis process in section Section 3.3. In Section 3.4 we demonstrate how this process might work on a simple web application, and we report on the results of a case study on an enterprise web application in section Section 3.5. Section 3.6 discusses the applicability of the proposed approach to various system types. Section 3.7 describes the further topics of investigation, and finally Section 3.8 summarises this chapter.

3.1 Example Website

In order to demonstrate how our hypothesis might work, we looked for an example of a software system where we could apply our approach (Figure 3.1). We wanted an example which typifies enterprise applications, but is smaller and hence more easily understandable and explainable. We chose a simple web application designed to promote a health club, and allow members to join and book activities online.

In this thesis, we refer to this application as the *example website*, and use it to explain our model, and describe the steps for deriving domain-based coupling between user interface components.



Login to your personalised SFACS page

Your login name will be your email address

Login:

Password:

Click here to retrieve your password.

All content copyright of the Sweat Hut Fitness and Sporting Club

webmaster@sfacs.com.au, +61 8 5555 5555

Figure 3.1: Example Website

3.2 Basic Concepts and Notation

In our work we use the following terminology:

- A *domain variable* is a variable unit of data which has a clear identity at the domain level.
- A *domain function* provides proactive or reactive domain-level behaviour of the system which includes at least one domain variable as an input or output.
- A *user interface component (UIC)* is a system component directly interacting with the system domain user and containing one or more domain functions.

For example, in a business software system, UICs are the software data entry forms. Each form provides one or more functions to the end user, and the data fields visible on the forms are the domain variables.

3.2.1 Notation

In this thesis, we use standard notation for binary relations. For $R, Q \subseteq A \times A$, we denote by $R.Q$ their composition, i.e., $xR.Qy$ iff $\exists z : xRz \wedge zQy$. R^{-1} denotes the inverse of R, and ID indicates the identity relation. Moreover we abbreviate $x.R = \{y|xRy\}$.

We use graph theory in our work, denoting by $G = (V, E, l)$ the graph G with vertices V, edges $E \subseteq V \times V$ and labelling $l : E \to L$ for some label set L. The edge set in a graph obviously defines a binary relation. It is common in computer science that any finite set X of pair-wise disjoint relations $R : A \times A$ on some set A can be equivalently represented by a graph with vertices A and directed edges $E = \bigcup_{R \in X} R$, the union of the given relations. This is achieved by naming relations and labelling the edges of the graph with corresponding relation names. More formally, let L be a finite set of relation labels and $l_R \in L$ the name of R for any $R \in X$. Then we define $REL(A, X)$ as the labelled directed graph $REL(A, X) = (V, E, l)$ with $V = A, E = \bigcup_{R \in X} R$ such that

$$(v, v') \in E \text{ and } l(v, v') = l_R \text{ iff } vRv' \text{ for some } R \in X.$$

We also call $REL(A, X)$ the *relation graph* of X over A. We note that relation application (dot notation in $x.R$ applying R to an object $a \in A$) and composition (dot notation $R.Q$) corresponds to path chasing in the relation graph.

3.2.2 Basic Concepts

The three basic concepts of our work are modelled by binary relations as follows:

1. *Domain variables* are simply modelled by a finite set V, called *variable symbols.*
2. *Domain functionalities* are modelled by a finite set F, so-called *function symbols.* The binary relations REF and $USE \subseteq F \times V$ represent elementary dependencies between functions and variables. For convenience, we define $REF \subseteq USE$ and interpret REF as input variables (read set) and USE as the input-output variables (read and write set). Because we are only interested in domain functionalities (interacting with an external user or external software) we assume moreover that $f.USE \neq \emptyset$ for all $f \in F$, i.e., a domain function uses one or more variables.

3. *User Interface Components (UICs)* are modeled by a finite set C called the *component symbols*. The associated function symbols are represented by the relation $HAS \subseteq C \times F$. We require that $c.HAS \neq \emptyset$ for all $c \in C$, i.e., a component has one or more functions.

Example

In the example website, the *LoginPage* is a UIC, and it has the following functions: *LoginPage*.HAS = {*authenticate*, *accept*, *reject*}. Informally, *authenticate* reads the name and password combination and excludes pathological cases such as empty strings entered by the user. The system then determines whether to *accept* or *reject* and produces the status *AuthenticationStatus* message. More formally:

authenticate.USE = *authenticate*.REF= *accept*.REF= *reject*.REF = {*UserName*, *Password*}, and *accept*.USE = *reject*.USE = *accept*.REF ∪ {AuthenticationStatus}.

Convention 1 *For the rest of the thesis, and without loss of generality, we assume the system under analysis (SUA) is fixed, that is,* V*,* F *and* C *are fixed and so are their* REF*,* USE *and* HAS *relations. We also call the graph* $REL(V \cup F \cup C, \{REF, USE \backslash REF, HAS\})$ *the* behavioural model *for the given SUA.*

Definition 1 *We define the* conceptual connection *relation* $CNC \subseteq C \times C$ *by*

$$CNC = HAS.USE.USE^{-1}HAS^{-1}$$

The CNC relations shows how user interface components are connected by sharing domain variables. CNC is reflexive, and since functions have a non empty variable set, the following corollary follows by definition.

Definition 2 *We call* $REL(C, \{CNC \backslash ID\})$ *the* conceptual connection graph *of the SUA.*

Corollary 1 *Two UICs* $c, c' \in C$ *are conceptually connected iff they are adjacent in the conceptual connection graph.*

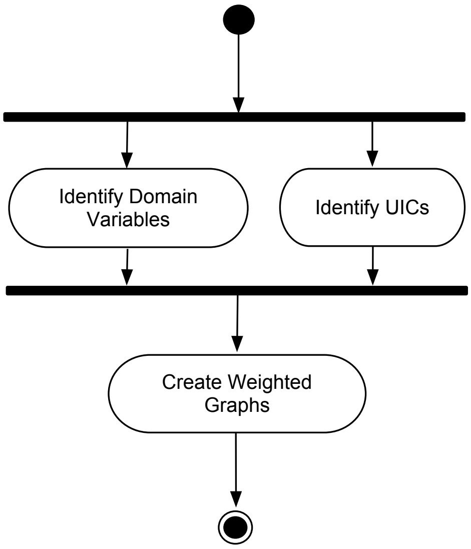


Figure 3.2: Domain-based Coupling Analysis Process

Example

In the example website, two UICs *ContactPage* and *LoginPage* are conceptually connected.

LoginPage.HAS.USE = {UserName,Password}

ContactPage.HAS.USE = {UserName,Age,Query,Email}

ContactPage.HAS.USE ∩ *LoginPage*.HAS.USE = {UserName} ≠ ∅

Definition 3 *A* weighted graph *is a labelled graph* $G = (C, CNC \backslash ID, f)$ *where* f *is a labelling function* $f : E \rightarrow [0..1]$ *assigning probabilities to edges.*

We use the definition of the weighted graph in the next section to describe how to create domain-based coupling graph for a SUA.

3.3 Methodology

In this section, we explain a process to analyse the behavioural model of an information system, and create weighted graphs from CNC relations. Figure 3.2 illustrates the process including three stages: identifying domain variables, identifying UICs, and creating weighted graphs. As illustrated in Figure 3.2 identifying domain variables and UICs are prerequisite for creating the weighted graph. These stages are described as follows.

3.3.1 Identify Domain Variables

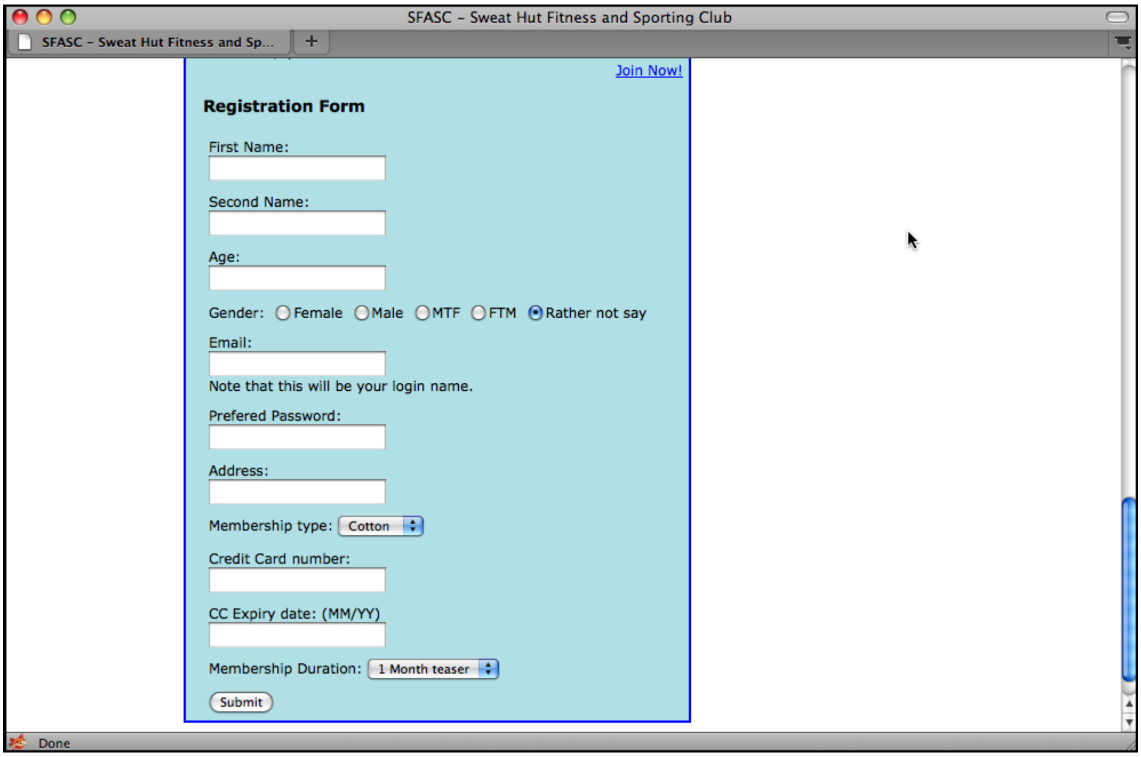
The aim of this stage is to identify the set of domain variables which are employed as part of the user interaction with the system. As defined in section 3.2, domain variables are variable units of data which have a clear identity at the domain level. In order to establish whether or not a particular data element is a domain variable, the answers to the following questions should be all true.

1. Is it variable data? An example of variable data in a business application might be the content of a data field on a screen (form), which can change based on the state of the application, business rules or constraints, e.g. *Account Code, User Name, or User Address.* As opposed to this, a footnote or tool-tip on the screen might provide non-variable data elements which do not change for a given version of the application.
2. Is the data understandable purely with domain knowledge? The answer to this question will indicate whether a domain user who has no familiarity with the architecture and source code of the given application can still understand the meaning and purpose of the given data within the domain.

In the example website, the registration form (Figure 3.3) contains the following domain variables: First Name, Second Name, Age, Gender, Email, Password, Address, Membership Type, Credit Card Number, Credit Card Expiry Date, and Membership Duration. This information is understandable without any knowledge of the specific software functionality and architecture.

3.3.2 Identify UICs

The aim of this stage is to identify the system's UICs. In order for a system component to be a UIC, the answer to the following questions should be all true:



SFASC – Sweat Hut Fitness and Sporting Club

SFASC – Sweat Hut Fitness and Sp...

Join Now!

Registration Form

First Name:

Second Name:

Age:

Gender: ○ Female ○ Male ○ MTF ○ FTM ● Rather not say

Email:

Note that this will be your login name.

Prefered Password:

Address:

Membership type: Cotton

Credit Card number:

CC Expiry date: (MM/YY)

Membership Duration: 1 Month teaser

Submit

Done

Figure 3.3: Example Website: Registration Form in Join Page

1. Is this component visible to the domain user?

2. Does this component provide at least one domain function?

   To identify UICs, knowledge of at least some of the system domain functionality is required. For a system function to be a domain function, the answer to the following questions should be true:

   2.a Does the function/feature change the external behaviour or property of the system at the domain level? This question helps to separate system functions from system non-functional properties such as reliability and visual characteristics (e.g. static background colour).

   2.b Does the given system function have at least one domain variable as part of its input or output? If the system function does not have any impact on domain variables, or is not affected by their value, then it is not a domain function, e.g. a software license control is not a domain function.

In the example website, LoginPage (Figure 3.1) and Join Page (Figure 3.3) are UICs.

3.3.3 Create Weighted Graphs

The aim of this stage is to create weighted graphs (Definition 3) which represent the strength of the relationship between UICs based on conceptual connections between UICs and domain variables.

Creation of weighted graphs is achieved as follows. Firstly, the HAS and USE relations are determined based on domain knowledge. The HAS and USE relations and the derived CNC relations are all compactly capturable in a *dependency matrix* defined as follows. Secondly, the weighted graphs is automatically derived from the captured dependency matrix.

Definition 4 *The* dependency matrix M *of a SUA is a matrix* $M_{c,v}$ *of true/false values where for* $c \in C$ *and* $v \in V$*,*

$$M_{c,v} = true : v \in c.HAS.USE$$

$$M_{c,v} = false : v \notin c.HAS.USE$$

Domain variables are the inputs and outputs of the domain functions; hence, where two UICs have common domain variables, this suggests the UICs have related domain functionality, and there might be some dependencies at the source code level which lead to change propagation. For example, two UICs with a lot of common domain variables may extend a common parent class, and if a software change requires alteration of the parent class, it will automatically alter the behaviour of other child classes.

The Domain-Based Coupling between two components is derived from shared domain variables, based on the following measurements:

Definition 5 *Number of common variables among two UICs is modelled by the function* $\vartheta : C \times C \to \mathbb{R}$ *where*

$$\vartheta(c, c') = |c.HAS.USE \cap c'.HAS.USE|$$

Note that the definition of common domain variables is symmetric, i.e., $\vartheta(c, c') = \vartheta(c', c)$*.*

Based on the asymmetric and symmetric domain-based coupling and definition Definition 3, we derive two weighted graphs as follows:

Definition 6 *Asymmetric domain-based coupling graph of a SUA is the asymmetric weighted graph* $G_a = (C, CNC \backslash ID, \omega)$ *where coupling weight function* $\omega_a : C \times C \to [0..1]$ *is*

$$\omega_a(c, c') = \frac{\vartheta(c, c')}{|c.HAS.USE|}$$

Definition 7 *Symmetric domain-based coupling graph of a SUA is the symmetric weighted graph* $G = (C, CNC \backslash ID, \omega)$ *where coupling weight function* $\omega : C \times C \to [0..1]$ *is*

$$\omega(c, c') = \frac{\vartheta(c, c')}{|c.HAS.USE \cup c'.HAS.USE|}$$

Note that by definition, (c, c') is an edge in these graphs iff $c\ CNC\ c'$ and $c \neq c'$. We use these graphs to visualise the conceptual connections between UICs.

Example

In the example website, the weight of the link between *ContactPage* and *LoginPage* is calculated as follows, where, as noted earlier, *LoginPage*.HAS.USE={UserName,Password}, the intersection of *ContactPage*.HAS.USE and *LoginPage*.HAS.USE is just {UserName}, and the union is {UserName, Password, Age, Query, Email}, therefore:

$$\omega_a(LoginPage, ContactPage) = \frac{1}{2} = 0.5 \qquad \omega(LoginPage, ContactPage) = \frac{1}{5} = 0.2$$

For a large SUA, the number of variables and functions may be large, and so the number of dependencies may not only be large but also dependencies between large-scale components of similar size (number of function points) may vary significantly in the number of domain variables shared. It turns out that it is practically very useful to weight dependencies by their level of sharing. A threshold λ can then be used to select *relevant* dependencies by their weight $w > \lambda$ only.

3.3.4 Visualisation

The aims of visualising the weighted graphs are: firstly to achieve a better understanding of domain-based coupling between pairs of UICs, and secondly to identify clusters of UICs based on domain-based coupling which can lead to change propagation.

We visualise the asymmetric graph using *Dot*, a utility tool, part of a well-known graph visualisation toolset called Graphviz [75]. *Dot* uses a four stage hierarchical graph drawing algorithm, based on the work of Gansner et al [64]. It breaks any cycles which occur in the input graph by reversing certain edges, then it ranks nodes and orders nodes inside each rank to avoid crossing, finally it positions nodes and routes edge splines [63]. The result is a directed graph with a minimum number of crossed edges.

Whilst the asymmetric graph is informative about individual UIC pairs, we use the symmetric graph to identify clusters of UICs paired by strong domain-based coupling. It is common to visualise a weighted graph as a spring graph, by constructing a physical model and running an iterative solver to find a low-energy layout. We use the *Neato* utility which is part of Graphviz. *Neato* uses an approach proposed by Kamada and Kawai [97] by placing a spring between each pair of vertices, and achieves a pleasing layout by minimising the energy of the spring system [143]. In our model the distance between vertices in the graph is calculated as follows:

Definition 8 *The ideal distance between two vertices is*

$$d(v, v') = \frac{k}{w(v', v)}$$

where k is a constant for the given graph, and w is the symmetric domain-based coupling.

Example

In the example website, Join Page, Login Page and Membership Page are the three UICs and the vertices of the weighted graphs, denoted by v_1, v_2, v_3. Tables 3.1 shows the domain-based coupling values between these vertices.

Figure 3.4 visualises the relations between these vertices based on asymmetric weight function

	v_1	v_2	v_3
v_1	▨	0.1	0.1
v_2	0.18	▨	0.10
v_3	0.91	0.50	▨

(a) w_a

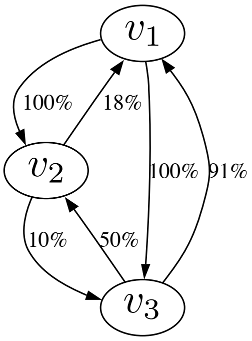
	v_1	v_2	v_3
v_1	▨	▨	▨
v_2	0.18	▨	▨
v_3	0.91	0.09	▨

(b) w

Legend: v_1: Join Page , v_2: Login Page, v_3: Membership Page

Table 3.1: Example Domain-based Coupling Values

(Table 3.1a). In this graph the distances between vertices do not represent the weight of edges, rather, they are aimed to improve graph presentation.



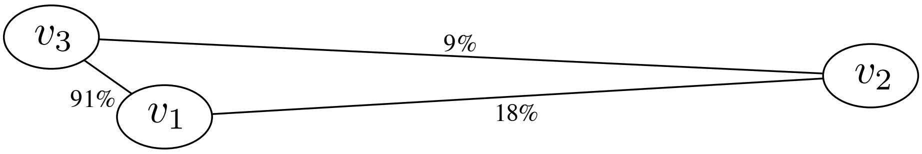
Legend: v_1: Join Page , v_2: Login Page, v_3: Membership Page

Figure 3.4: Example Asymmetric Weighted Graph

Figure 3.5 visualises the symmetric weigh function (Table 3.1b) between these vertices where the distance between them are measured as:

$$d(v_1, v_2) = \frac{1}{0.18} \qquad d(v_1, v_3) = \frac{1}{0.91} \qquad d(v_2, v_3) = \frac{1}{0.09}$$

In the next section, we use the example website to demonstrate how our methodology can be used to derived the asymmetric and symmetric weighted graphs from a web-based software application.



Legend: v_1: Join Page , v_2: Login Page, v_3: Membership Page

Figure 3.5: Symmetric Weighted Graph Sample

3.4 Case Study: Example Website

In this section, we use the example website (Section 3.1) to provide a detailed demonstration of how the proposed methodology might work on an information system.[1]. Given the simplicity of the SUA, we explain the processes in detail including identifying domain variables and UICs, and creating the weighted graphs. Moreover, we compare the derived graphs with the dependencies in the source code.

	Title	Description
UIC_1	Activities	Provides a list of current activities in the club.
UIC_2	Contact us	Provides a form for submit a request or a question.
UIC_3	Delete Account	Enable current users to delete their account.
UIC_4	Join	Provides a form for joining the club.
UIC_5	Login	Authenticate the user.
UIC_6	Membership	Provides account information.

Table 3.2: Example Website UICs

[1] This is a limited study, and it does not aim to equate the accuracy of our approach. In chapters 4 & 5, we will present large scale case studies which evaluate the efficiency and accuracy of the proposed methodology.

Domain Variable	UIC_1	UIC_2	UIC_3	UIC_4	UIC_5	UIC_6
ActivityDate	true	false	false	false	false	false
ActivityDay	true	false	false	false	false	false
ActivityInstructor	true	false	false	false	false	false
ActivityName	true	false	false	false	false	false
ActivityRemainedPlace	true	false	false	false	false	false
Address	false	false	true	true	false	true
Age	false	true	true	true	false	true
CreditCardExpiryDate	false	false	true	true	false	true
CreditCardNumber	false	false	true	true	false	true
Email	false	true	true	true	true	true
FirstName	false	false	true	true	false	true
Gender	false	false	true	true	false	true
MembershipDuration	false	false	true	true	false	true
MembershipType	false	false	true	true	false	true
Password	false	false	true	true	true	false
Query	false	true	false	false	false	false
SecondName	false	false	true	true	false	true
UserName	true	true	false	false	false	false

Table 3.3: Example Website: Dependency Matrix

3.4.1 Domain Analysis

We had three people[2] independently analysing the web application based only on domain information, primarily the user interfaces, i.e, the website architecture was *not* made available to them. Two people had the role of typical domain users with a superficial understanding of system functionality, and one had the role of a domain expert. The domain expert studied the system specification document which describes system features and functionality (excluding implementation data).

The domain expert and both domain users created a list of UICs, then listed domain variables related to UICs by the *HAS.USE* relationship. The results suggest that the accuracy of the derived relationship model is highly dependent on domain users' knowledge about the system functionality. We observed the following difficulties in the domain analysis: Firstly, domain users did not identify all UICs, and they only reported on UICs which are listed

[2] People who participated this case study had the computer science background, but they have not seen this website before the case study.

	UIC_1	UIC_2	UIC_3	UIC_4	UIC_5	UIC_6
UIC_1						
UIC_2	11%					
UIC_3	0%	15%				
UIC_4	0%	15%	100%			
UIC_5	0%	20%	18%	18%		
UIC_6	0%	17%	91%	91%	9%	

(a) Symmetric Graph

	UIC_1	UIC_2	UIC_3	UIC_4	UIC_5	UIC_6
UIC_1		25%	0%	0%	0%	0%
UIC_2	17%		18%	18%	50%	20%
UIC_3	0%	50%		100%	100%	100%
UIC_4	0%	50%	100%		100%	100%
UIC_5	0%	25%	18%	18%		10%
UIC_6	0%	50%	91%	91%	50%	

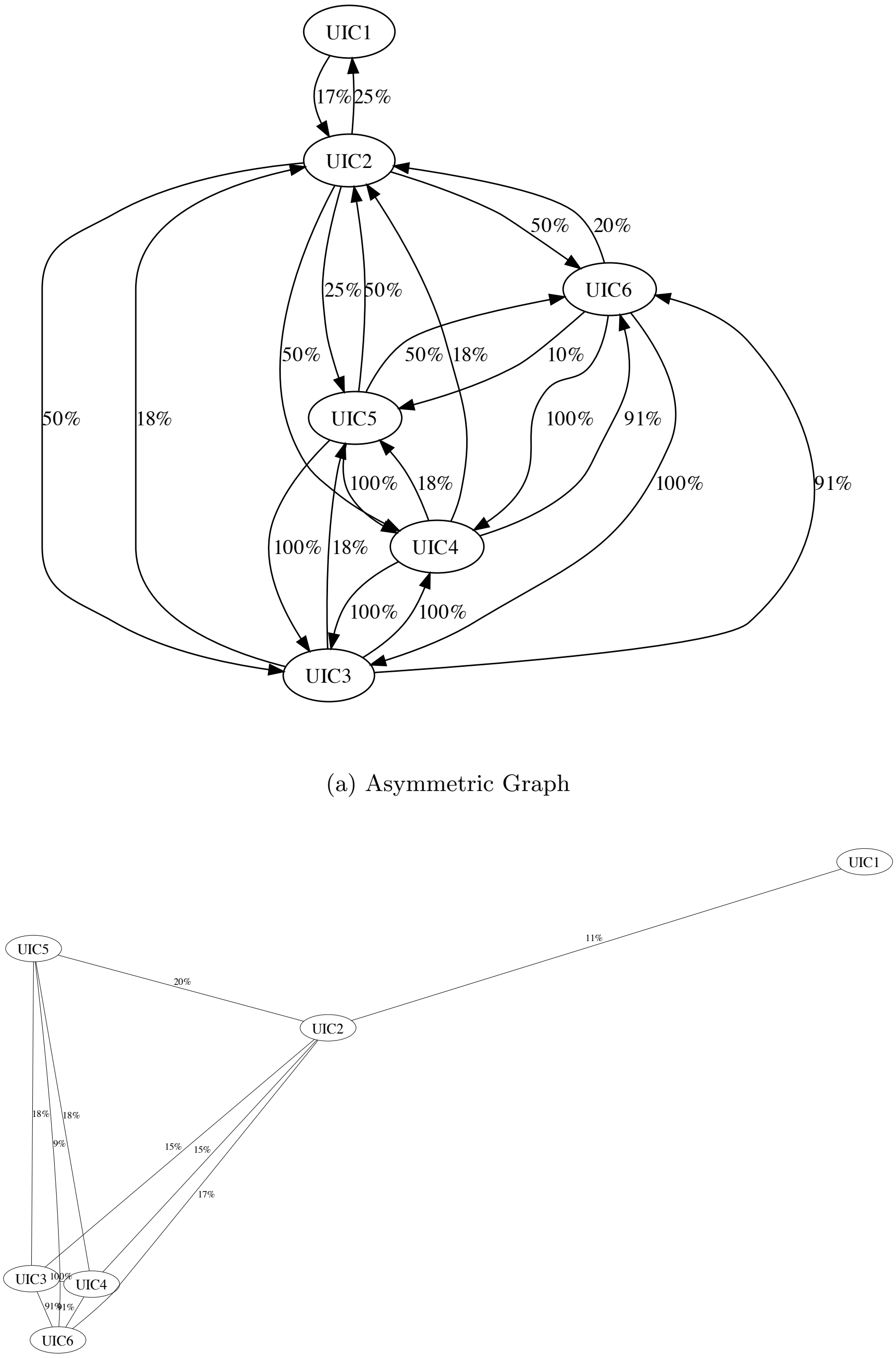
(b) Asymmetric Graph

Table 3.4: Example Website: Edge labels of weighted graphs

in the website main menu. However, there are some UICs in the system which could only be accessed through a button on a specific page, such as the lookup screens. Secondly, the website has two access modes: guest users and members. Some UICs and domain variables are only visible to the members. The domain users ignored the member mode, and only recorded the ones that were accessible to guest users. In comparison, the domain expert who studied the software functional specification identified all UICs and related domain variables.

The website includes ten web pages. There are four web pages which did not qualify as UICs: *Home*, *About*, *Related Sites* and *Site Map*. The first three web pages contain only static contents, and the *Site Map* provides only non domain related data (addresses and links to other websites). The six other web pages have domain variables, and provide the functionalities of querying or editing domain information. Hence the domain expert identified them as UICs. Table 3.2 shows the UICs in the example website. Table 3.3 shows the dependency matrix derived by the domain expert. The rows are domain variables, and columns are UICs. Each cell has a value of *true* or *false* that identifies whether there is a relationship between the corresponding domain variable and the UIC.

In the next step, an automated script is used to create the asymmetric and symmetric



(a) Asymmetric Graph

(b) Symmetric Graph

Figure 3.6: Example Website: Weighted Graphs

weighted graphs. Figure 3.6 illustrates the graphs. Each node represents a UIC, and each edge shows a CNC relationship (Definition 1) between two UICs, labelled with ω_a and ω (coupling weight functions) for the asymmetric and symmetric graphs respectively. For improving readability, the edge labels are listed in Table 3.4.

The asymmetric weighted graph (Figure 3.6a) includes six edges with strong coupling of ω_a =100%. This implies that all domain variables involved in the source UIC exist in the contents of the target UIC. Such a strong coupling value suggests that the functionality of the target UIC might be dependent upon the source UIC. In contrast, there are edges with weak coupling values including nine edges which are labelled with ω_a <30%. The weak coupling value is an indication that the majority of domain variables in the source UIC do not exist in the contents of the target UIC, hence we can not conclude that any dependency exists between them based on domain-based coupling. Later in Chapter 5, we demonstrate how the asymmetric coupling can be used to predict change propagation.

While the asymmetric weighted graph shows the detailed coupling values between pairs of UICs, the symmetric weighted graph can be used to identify clusters of UICs based on their domain-level commonality. These clusters can identify architectural dependencies which might lead to change propagation. For example, in the symmetric graph presented in Figure 3.6b, three components UIC_3, UIC_4, and UIC_6 are the most tightly coupled UICs which formed a cluster. In contrast, UIC_1 is loosely coupled to others and this might indicate that there is no architectural dependency between their source code. In the next section, we will demonstrate how these clusters in the graph represent source code dependencies.

3.4.2 Comparison with Architectural Dependencies

Now that we have completed domain-based coupling and created weighted graphs, we aim to analyse website source code and discover if the weighted graphs can lead us to architectural dependencies. Figure 3.7 shows the architecture of the website. Each component is marked by its name, UICs are tagged by $[UIC_i]$, code libraries are identified by white background, and data sources are represented by the name of XML file which holds the data.

The website has been developed using PHP including both web pages and libraries. The UICs recorded by the domain expert are pages that read and write some domain variables into one of the three XML data files. In addition, there are four web pages which are not

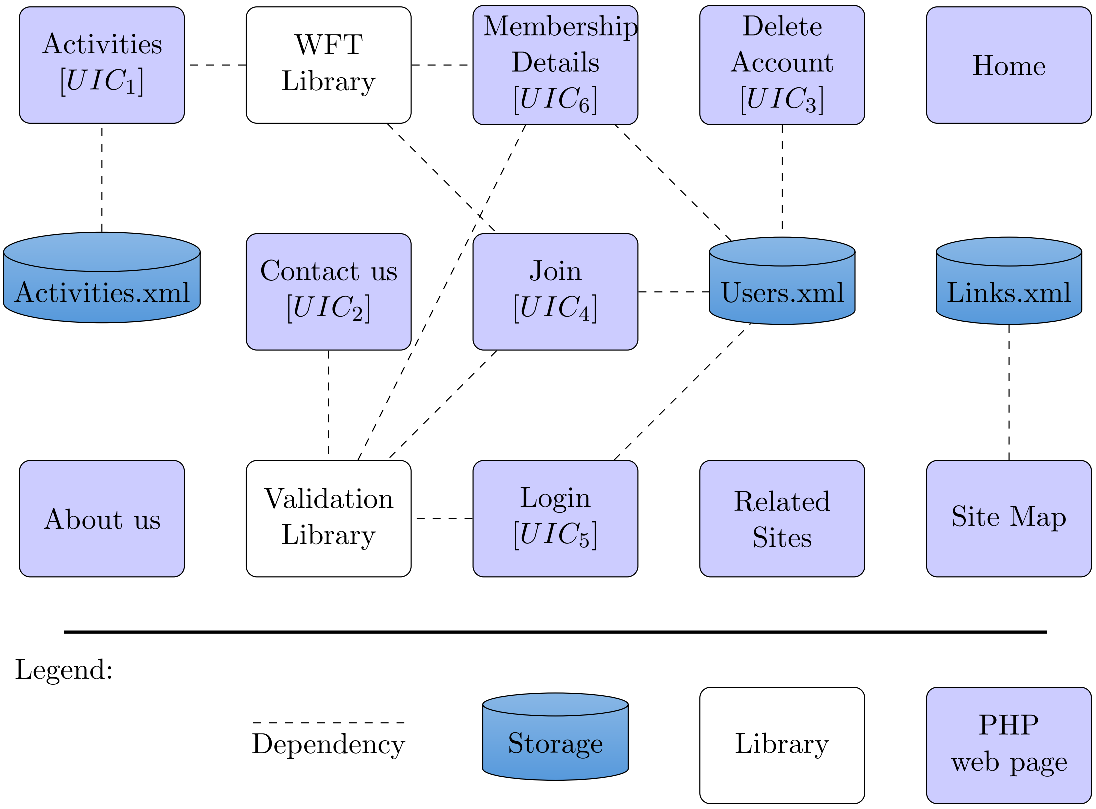


Figure 3.7: Example Website Architecture

qualified as UICs including *Home, Site Map, Related Sites* and *About us.* Reviewing the source code behind these web pages showed that they only contain static HTML contents with no dependency or connections to any other files. Changing these four web pages highly unlikely to affect other components.

For the six UICs in the website, we compared the dependencies in the source code and the domain-based coupling graphs. Table 3.5 presents a comparison between source code dependencies (Figure 3.7) and the symmetric domain-based coupling graph (Figure 3.6b). The third column indicates the architectural dependencies, if any, between two UICs and the forth column shows the coupling weight between them.

The weighted graph shows eleven CNC relationships between UICs, but only nine of them match the architecture connections, i.e., there were two false positives and two false negatives. In the false positive cases, web pages share domain variables which have the same name on the screen but are implemented as different data fields. This is a discrepancy between the system specification and the actual implementation in the source code. In both false negative cases, the web site pages are connected by referencing a utility library, although there is no common functionality or domain variable visible to the domain user. Both false positive results are located on the light edges ($w < 16\%$) which suggests a rough correlation between the weight of the graph edge and whether there is an architecture connection in the source code. In addition there are three edges with ($w > 80\%$) between UICs which are related in the source code as they read and write all fields of the same data source (User.XML).

Component 1	Component 2	Architecture dependency	Graph edge labels	False results
UIC_1 (Activities)	UIC_3 (DeleteAccount)		0.00	
UIC_1 (Activities)	UIC_4 (Join)	Yes [W]	0.00	FN
UIC_1 (Activities)	UIC_5 (Login)		0.00	
UIC_1 (Activities)	UIC_6 (Membership Details)	Yes [W]	0.00	FN
UIC_1 (Activities)	UIC_2 (Contact us)		0.08	FP
UIC_5 (Login)	UIC_6 (Membership Details)	Yes [X]	0.09	
UIC_2 (Contact us)	UIC_3 (DeleteAccount)		0.15	FP
UIC_2 (Contact us)	UIC_4 (Join)	Yes [V]	0.15	
UIC_2 (Contact us)	UIC_6 (Membership Details)	Yes [V]	0.16	
UIC_3 (DeleteAccount)	UIC_5 (Login)	Yes [U]	0.18	
UIC_4 (Join)	UIC_5 (Login)	Yes [X]	0.18	
UIC_2 (Contact us)	UIC_5 (Login)	Yes [V]	0.20	
UIC_3 (DeleteAccount)	UIC_6 (Membership Details)	Yes [U]	0.90	
UIC_4 (Join)	UIC_6 (Membership Details)	Yes [Z]	0.90	
UIC_3 (DeleteAccount)	UIC_4 (Join)	Yes [U]	1.00	

Legend: Pages are connected via [V]:Validation library, [W]: WFT library, [U]: User.xml, [X]: Validation library and User.xml, [Z]:Validation and WPF libraries and User.xml. False results: FN: False Negarive, FP: False Positive

Table 3.5: Weighted connection graph compared to architectural dependencies

3.4.3 Discussion

In this section, we demonstrated how the proposed methodology can be applied to a web application. Moreover, we describe how a domain expert can derive the domain-based coupling graphs, and how the source code dependencies can be compared to such graphs.

The presented results demonstrate that the highly coupled components with $\omega > 0.8$ are architecturally connected by reading and writing to the same data source, whilst components with weak coupling are less likely to be connected in the source code. However, this was a study on a simple website for the main purpose of demonstrating the details of the methodology. Thus, we can not draw a strong conclusion from these results.

In the next section, we perform a comprehensive study on a web-based enterprise system with a significant sized source code and multi-tier architecture. The aim of the next study is to evaluate if similar results can be achieved when we apply our methodology to a system with the complex architecture.

3.5 Case Study: An Enterprise Web Application

This section reports on a result of a case study on a web-based enterprise system. The purpose of this case study is to evaluate the efforts and challenges of domain analysis of a real life enterprise system. In addition, we examine to what extent the derived domain-based coupling graphs can assist to discover architectural dependencies in the source code.

The system under analysis is Building Condition Assessment (BCA), a web-based software application designed to manage a large volume[3] of building condition audit data. BCA is developed based on multi-tiered architecture. It has three distinctive layers including presentation layer, business and data layers which manifest the clear separation of concerns. These layers are implemented based on Microsoft .Net technology which takes advantage of AJAX for the presentation layer, Component-based Scalable Logical Architecture (CSLA) for the business layer and Microsoft Enterprise Library for the data layer. In general, it has an object oriented design, and consists of classes, organised in different namespaces.

In the presentation layer each UIC is composed of a webpage (ASPX file) and a class which contains the source code of the page. Later in Section 3.5.2, we analyse the dependencies between these classes and how they match the domain-based coupling value.

[3] Regression testing for BCA is passed by more than two million records in reports with response time less than four seconds.

3.5.1 Domain Analysis

We invited a domain user of BCA to use our proposed methodology to derive the dependency matrix and create the weighed graphs. He was not a software engineer, nor had access to system source code; however, he has worked with the system for more than three years, entering and maintaining the data pertaining to his business.

He completed the domain analysis with a manual procedure using only a generic Excel worksheet to record the domain information and create the dependency matrix. Then we compared the derived domain variables for each UIC with the defined fields in system functional specification[4]. The qualitative assessment of the results and the feedback of the domain user suggest the following challenges:

- There are dynamic behaviours in the system which hides some information or alter their presentation based on the user profile, or system settings. Such behaviours made some domain variables hidden for our domain user, causing false negative results. We used functional specification document to find and understand such dynamic behaviours.
- Some domain variables have very generic names on the screens such as *Status* and *Description.* These generic names for domain variables make them difficult to be uniquely identified in the weighted connection graph. We changed the name of these domain variables to a unique identifier by adding associative prefix such as *WorkOrderStatus* instead of *Status.*
- Some domain variables have different names in different parts of the system. This is a problem of inconsistency in naming conventions for different domains, for example *Job Number* and *Work Order Number* refer to the same entity although named differently in two separate sub domains of the system. We referred to the functional specification to clarify some of these ambiguities.
- Some domain variables are named inconsistently in screens. It is not uncommon to see synonyms or arbitrary prefixes/suffixes added to the name of entities. These inconsistencies can mislead the domain user to record false positive domain variables,

[4] The functional specification used in this case study describes the system behaviour with no information about software source code.

e.g., *Inspection Contact*, *Contact Name*, and *Inspector* refer to the same entity in BCA.

- Some data fields on UICs show calculated values derived from other domain variables, *e.g.*, *Total Cost* and *Summary Cost* are derived from the domain variable *Cost*. These calculative fields can hide underneath domain variables, and affect the domain analysis result of the SUA.

We observed that the domain user spent a noticeable part of the analysis time in consolidating the results, removing duplicated domain variables and leaving comments on ambiguous parts. The observed challenges, and the effort by the domain user suggest the need for tool support, and raise the question that how this process can be automated. We address this question in Chapter 6, where we discuss various possibilities for automating the domain-based coupling analysis, and present a design for tool support.

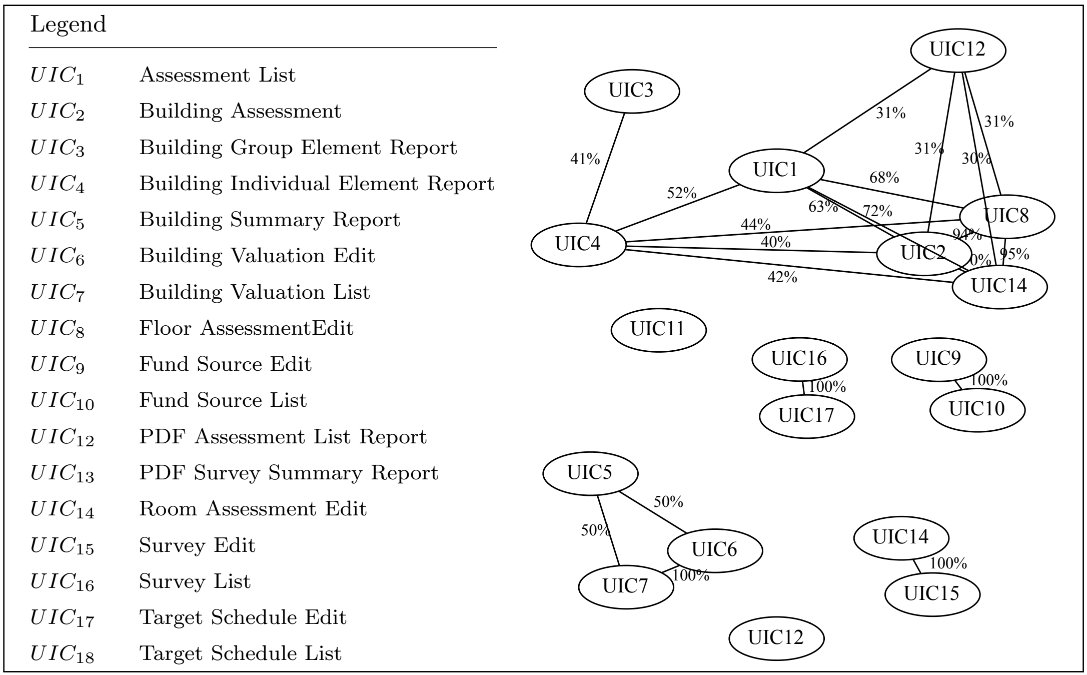
3.5.2 Comparison with Architectural Dependencies

In the next step, we use the derived domain relationships to approximate the dependencies in the source code. The aim is to perform a qualitative analysis on the source code dependencies, and demonstrate how they match the domain-based coupling between UICs.

We use the symmetric weighted graph to achieve this goal. The graph is generated using the derived dependency matrix by the domain user. However, the resulting graph is very dense (unreadable), and we changed the threshold to $w \geq 0.3$ where the graph is more readable. The outcome is illustrated in Figure 3.8. It turns out that the density of the graph is a potential problem for complex systems. Figure 3.9 is an alternative presentation for the weighted graph. It is a cross table which shows all domain-based coupling values between UICs.

We used .Net Code Model to extract the list of all classes and references between them. The result shows that the BCA architecture consists of 154 classes including 18 classes for UICs. Comparison between these results and the domain-analysis results shows that each of these 18 classes match one and only one UIC at the domain level.

When a member of a class calls another class member then there is a reference between them.


Legend
UIC1 Assessment List
UIC2 Building Assessment
UIC3 Building Group Element Report
UIC4 Building Individual Element Report
UIC5 Building Summary Report
UIC6 Building Valuation Edit
UIC7 Building Valuation List
UIC8 Floor AssessmentEdit
UIC9 Fund Source Edit
UIC10 Fund Source List
UIC12 PDF Assessment List Report
UIC13 PDF Survey Summary Report
UIC14 Room Assessment Edit
UIC15 Survey Edit
UIC16 Survey List
UIC17 Target Schedule Edit
UIC18 Target Schedule List
UIC12
UIC3
31%
31%
31%
41%
UIC1
30%
68%
52%
63%
72%
44%
UIC8
UIC4
40%
UIC2
94%
0%
95%
42%
UIC14
UIC11
UIC16
UIC9
100%
100%
UIC17
UIC10
UIC5
50%
50%
UIC6
UIC14
100%
UIC7
100%
UIC15
UIC12

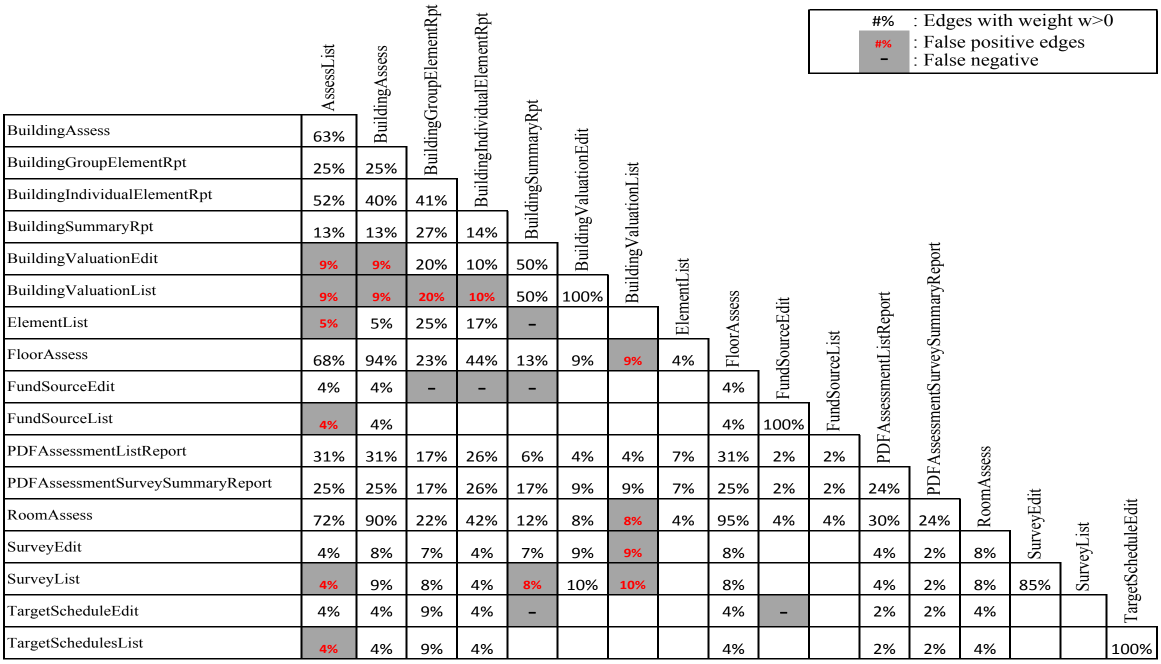
Threshold $w \geq 30\%$.

Figure 3.8: BCA Symmetric Weighted Graph

If there is at least one reference between members of two classes then they are architecturally dependent. In BCA, there are 1270 individual references between different classes yielding 403 dependencies between pair of classes.

We found 45 classes related to domain functionalities, covering majority of classes in the presentation layer and non-utility classes in the business and data layers. These classes have 169 (far less than 1270) individual source code references between them, yielding 121 pairs of dependent classes. In order to compare these dependencies with edges of the weighted graph, we searched for transitive dependencies between classes related to UICs (18 classes out of 45). For example two UICs *Room Assess* and *Floor Assess* are both architecturally dependent on *Assessment Edit* class in the business layer, so there is a transitive dependency between *Room Assess* and *Floor Assess*.

The result shows 107 pairs of connected UICs, each pair connected by at least one transitive architectural dependency. Comparing these pairs with the weighted graph shows only 101 pairs match edges in the graph (graph has 116 edges), i.e., 15 false positive results where the edges did not match any transitive dependency in the source code. Also we found 6 false



#% : Edges with weight w>0
#% : False positive edges
– : False negative

	AssessList	BuildingAssess	BuildingGroupElementRpt	BuildingIndividualElementRpt	BuildingSummaryRpt	BuildingValuationEdit	BuildingValuationList	ElementList	FloorAssess	FundSourceEdit	FundSourceList	PDFAssessmentListReport	PDFAssessmentSurveySummaryReport	RoomAssess	SurveyEdit	SurveyList	TargetScheduleEdit
BuildingAssess	63%																
BuildingGroupElementRpt	25%	25%															
BuildingIndividualElementRpt	52%	40%	41%														
BuildingSummaryRpt	13%	13%	27%	14%													
BuildingValuationEdit	**9%**	**9%**	20%	10%	50%												
BuildingValuationList	**9%**	**9%**	**20%**	**10%**	50%	100%											
ElementList	**5%**	5%	25%	17%	–												
FloorAssess	68%	94%	23%	44%	13%	9%	**9%**	4%									
FundSourceEdit	4%	4%	–	–	–				4%								
FundSourceList	**4%**	4%							4%	100%							
PDFAssessmentListReport	31%	31%	17%	26%	6%	4%	4%	7%	31%	2%	2%						
PDFAssessmentSurveySummaryReport	25%	25%	17%	26%	17%	9%	9%	7%	25%	2%	2%	24%					
RoomAssess	72%	90%	22%	42%	12%	8%	**8%**	4%	95%	4%	4%	30%	24%				
SurveyEdit	4%	8%	7%	4%	7%	9%	**9%**		8%			4%	2%	8%			
SurveyList	**4%**	9%	8%	4%	**8%**	10%	**10%**		8%			4%	2%	8%	85%		
TargetScheduleEdit	4%	4%	9%	4%	–				4%	–		2%	2%	4%			
TargetSchedulesList	**4%**	4%	9%	4%					4%			2%	2%	4%			100%

Figure 3.9: Cross Table Presentation of the BCA Symmetric Weighted Graph

negative results where there is a transitive dependency between two UICs in the source code but the graph does not include an edge between them. The reason for these false negatives is a specific functionality in BCA which is not visible to domain users. It is a function which protect the data integration, and as part of each transaction it checks the related data tables to avoid deleting a parent record while there is a child record in another table. All false negative results are related to this functionality. Figure 3.9 shows these false results in the weighted graph.

Figure 3.10 illustrates the distribution of graph edges based on their weight. The majority of edges in the graph have less than 0.33 weight and few edges have weight higher than 0.9. Also all the false positive results are in low strength edges, which suggests for stronger edges we can be more certain in an architectural dependency between the two UICs. Comparing the architectural dependencies with these edges shows all the UIC pairs connected by an strong weighted edge ($w > 0.8$) are reading and writing to the same data table.

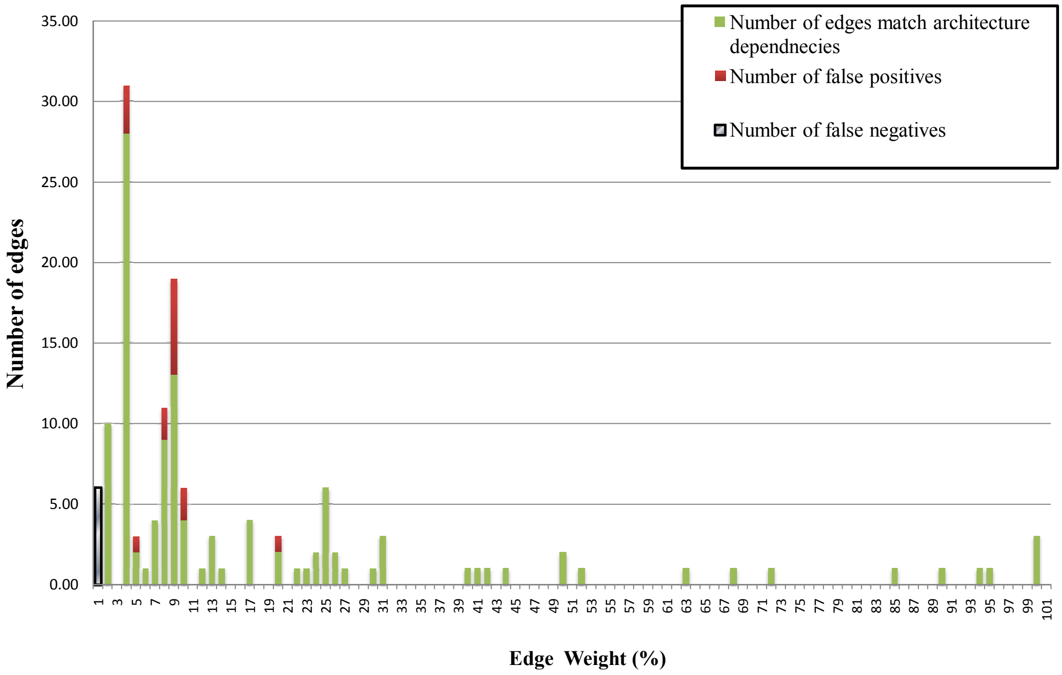


Figure 3.10: Distribution of edges in BCA Symmetric Weighted Graph

To avoid false positives in the graph we changed the threshold to $w > 0.3$ (see figure 3.8). Also it turns out to be practically useful to change the threshold where the UICs are grouped by strong edges in the weighted graph with a spring layout.

3.5.3 Discussion

In this section, we presented an application of domain analysis on an enterprise web-based system, where we reported on challenges and issues in the domain analysis of the system, and the process of creating the domain-based coupling graph. In addition, we provided a comparison between the derived graph and the source code dependencies between software components. The result shows that all UIC pairs with strong domain-based coupling are connected in the source code. In addition, a limited number of false results are identified which are mainly located between UICs with weak domain-based coupling. Thus, these false results can be filtered out by applying a threshold to the domain-based coupling graph.

The source code analysis which is performed in this study was limited, and mainly aimed at qualitatively examining how domain-based coupling between UICs can correspond to dependencies in the source code. Later in Chapter 4, we will provide a more formal and automated approach to the analysis of architectural dependencies.

3.6 Applicability

In this chapter we introduced a novel methodology for analysis of domain-based coupling between UICs, in order to answer the question of what kinds of software can take the most benefit from the proposed methodology.

In Section 2.3, we described the E-Type software as the group of programs that have been designed to mechanise human or societal activities. Most systems of this software type can take benefit from domain analysis. However, in a more detailed classification, Pressman organised computer software under seven categories: system, application, engineering/scientific, embedded, product-line, artificial intelligence, and web applications [152]. Our approach is applicable to subsets of application software, product-line software and web applications, which are data driven and provide their functionality through a number of user interface components.

Our approach is not applicable to software where the functionality of the system is not visible to the domain users, such as system software or embedded software. Also, domain analysis may not be suitable where systems are not data driven or have few user interface components, such as engineering/scientific or artificial intelligence software.

3.7 Open Issues

This chapter aimed to provide a pragmatic methodology for analysis of the software behaviour at the domain level; however, the case studies identify a number of outstanding issues in this area:

- The weighted graphs can be complex and dense for large scale systems. This will prevent us from reading the details in the graph, and it is a disadvantage in the analysis of systems where the clusters are not clearly separated. This issue might be answered using a better graph visualisation tool support.
- We addressed the issue of density of the graph by applying a threshold and filtering out edges with a coupling value less than the given threshold. In the case study, we have achieved the optimum threshold using a heuristic method and a manual process. However, finding the threshold automatically is still an outstanding issue. In the next chapter, we propose an alternative approach using an automated clustering technique.
- We examined the relationship between domain-based coupling and architectural connections in two case studies on web-based systems; however, given the limitations of these studies, extended experiments are required to evaluate the impact of domain-based coupling on dependencies in the source code and other architectural characteristics such as software modularisation.
- As we only examined web-based systems, similar case studies are needed to understand the application of domain-based coupling on different system types such as mobile applications, service-based systems, and embedded systems. Also such studies can investigate that how the process can be improved based on characteristics of different software types, for example, service-based systems which have XML-based documented interfaces that can be used for mining domain-variables and the $HAS.USE$ relationships.
- The case studies in this chapter were performed by a few domain users; hence, extended usability studies are needed to understand and address the challenges of domain analysis by non technical domain users.

3.8 Summary

In this chapter, we introduced domain-based coupling as a measure of semantic similarity between software components, and we described how to visualise such a coupling as symmetric and asymmetric weighted graphs.

We demonstrated the details of the domain analysis process for a web application, where we derived relationships between domain variables and components, and measured the domain-based coupling based on these relationships. In addition, we reported on a qualitative analysis on an enterprise web application where we compared the dependencies in the source code with the derived domain-based coupling. The results show that all component pairs with strong domain-based coupling are connected at the source code level. Although our results are positive, we cannot draw a strong conclusion from these limited studies.

In the next two chapters, we report on case studies on large-scale enterprise systems where we use domain-based coupling measure to predict architectural dependencies and change propagation.

Chapter 4

Predicting Architectural Dependencies

> Design and programming are human activities;
> forget that and all is lost.
>
> *The C++ Programming Language. pp. 693.*
> Bjarne Stroustrup

Software dependencies play a vital role in program understanding, reverse engineering, change impact analysis and other software maintenance activities. Traditionally, these activities are supported by source code analysis; however, source code analysis sometimes is difficult to achieve such as hybrid systems with heterogeneous source code. In addition, not all stakeholders have adequate knowledge about the source code. Non-technical domain experts and consultants raise most maintenance requests; however, they cannot predict the cost and impact of the requested changes without the support of developers.

Enterprise software systems are constructed to model business domains [114]. It is reasonable to expect that real-world dependencies are therefore reflected in the software itself. This chapter addresses the second research question of this thesis: how accurately can we identify architectural dependencies using domain-based coupling? In particular, we examine the following scenarios:

- **Searching for source code dependencies:** Suppose a software maintainer has no access to source code analysis tools. Using software domain information, how accurately can she predict existence of source code dependencies between UICs?
- **Searching for database relationships:** Some business constraints and relationships are defined and managed at the data layer. These relationships may or may not be visible at the source code level [179, 196], or can be difficult to analyse such as legacy databases. How accurately can a domain expert predict such relationships without analysing the database?
- **Searching for architectural dependencies:** When domain experts propose changes to UICs, how accurately can they identify other connected UICs which might be related to the propose changes?

In order to evaluate these scenarios, a case study is presented on a large-scale enterprise system, called ADempiere, where we demonstrated how the introduced domain-based coupling (Chapter 3) can be used to identify dependencies in the source code and database layers. The contribution of this chapter are:

- to demonstrate the process of capturing domain information, measuring domain-based coupling, and clustering the results for a large-scale enterprise system,
- to provide a formal approach for modelling architectural dependencies across the code and the database layers,
- to report on an empirical study of one of the largest open source enterprise systems, and demonstrate how domain-based coupling can be used to predict the source code and database dependencies.

The rest of this chapter is organised as follows: Section 4.1 introduces the system under analysis. Section 4.2 describes the formal model for architectural dependencies. Section 4.3 shows how to derive domain-based coupling. Section 4.4 evaluates how accurately domain-based coupling identifies dependencies at the source code and the database layers. Section 4.5 discusses the threats to the validity of our findings. Section 4.6 describes the further areas of investigation, and finally Section 4.7 summarises this chapter.

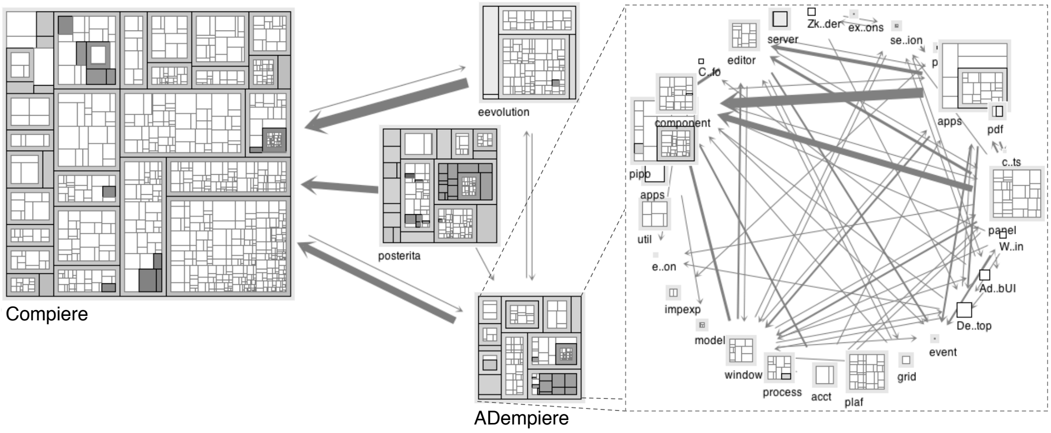


Figure 4.1: A high level view of ADEMPIERE*'s architecture*

4.1 Case Study: ADempiere

The system under analysis in this chapter is ADEMPIERE[1]; a large-scale Enterprise Resource Planning (ERP) software package. An ERP system integrates internal and external management information across an entire organisation, embracing accounting, manufacturing, sales and services, *etc.* The business rules and process in such domains are identifiable at the system level and independent from the software package *e.g.,* accounting terms and rules are defined beyond accounting software. This is the type of software which benefits mostly from domain-based coupling analysis.

The other qualities of ADEMPIERE are tiered architecture and complex design which manifests a commonly used enterprise software package. The system architecture composed of multiple tiers. It has a rich set of UI components and four distinct front-ends from which the user can choose including a Java GUI and three web interfaces. Also it heavily uses relational database management systems (*e.g.,* PostgreSQL and Oracle) for data storage as well as for storing business logic.

ADEMPIERE represents cutting edge open-source software technology. It is a multi-language system that includes more than two million lines of code. The core part is written in Java and contains more than 3,000 classes with more than half a million lines of code. The ADEMPIERE project traces its evolution back more than a decade. Created in September 2006 as a fork of the Compiere open-source ERP, itself founded in 1999. At the time of writing this thesis,

[1] http://www.adempiere.com

ADempiere is in the top five of the *SourceForge.net* enterprise software rankings. This is a measure of both the size of its community and its impact on the ERP software market.

Figure 4.1 presents a high-level architectural view of the Java core of ADempiere. The view is obtained by an architecture recovery tool called Softwarenaut[2]. It shows the result of aggregating direct relationships in the system along the package hierarchy [124]. The area of every visible module is proportional to its number of lines of code. Every visible dependency is directed and has its width proportional to the number of abstracted low-level dependencies. Every module is represented as a modified tree-map, with the sizes of the contained classes and modules proportional to their size in lines of code.

In addition, ADempiere has a very active community. The mailing list has more than 800 messages per month, and it is downloaded more than 15,000 times per month from *SourceForge.net*. This system is used by a large number of companies around the world.

For all these reasons, ADempiere is considered to be relevant, and represents a suitable case study for application of domain-based coupling in predicting architectural dependencies. The next section describes architecture dependencies in the scope of this study, and explains how they can be identified in ADempiere.

4.2 Dependency Analysis

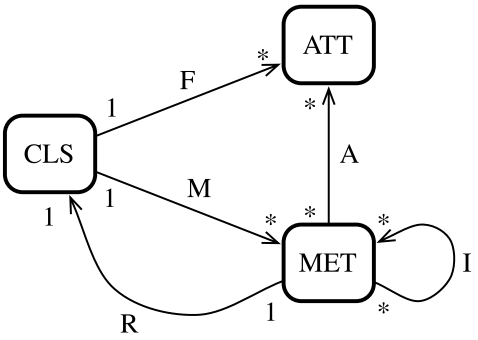
ADempiere has been designed in such a way that a developer can extend the system by touching as little code as possible. Whenever a new table is added to the database, the required Java code is automatically generated. Most business rules and domain-level relations are managed at the data layer. As a consequence, traditional code-based coupling metrics fail to capture all relationships between user interface components of ADempiere. Moreover, the database contains important information about the architectural dependencies in the system. Therefore for this study, a new model is required to express dependencies both at the source code and at the database layers.

[2] http://scg.unibe.ch/softwarenaut

4.2.1 Source Code Dependencies

Source code is the core of software architecture. At the source code level, there are three key entities. These entities are independent of the programming language, as long as it is object-oriented:

- *Classes* are represented by a finite set CLS.
- *Attributes* are represented by a finite set ATT. The binary relation $F \subseteq CLS \times ATT$ maps attributes to the containing classes.
- *Methods* are represented by the finite set MET. The binary relation $M \subseteq CLS \times MET$ maps methods to the classes that contain them.



CLS: classes, ATT: attributes, MET: methods

Figure 4.2: Source code elements and relations among them.

In addition, the relation $R \subseteq MET \times CLS$ expresses the return types of methods [3], $I \subseteq MET \times MET$ represents method invocations, and $A \subseteq MET \times ATT$ represents the accesses of methods to attributes. Two classes $cls, cls' \in CLS$ can have following relationships:

$$cls.M^{-1}.R^{-1}.cls' \tag{4.1}$$

$$cls.M.I.M^{-1}.cls' \tag{4.2}$$

$$cls.M.A.F^{-1}.cls' \tag{4.3}$$

[3] In order to model methods which return void, it has been considered that $Void \in CLS$

where Equation 4.1 shows cls is the return type of cls', Equation 4.2 shows a method of cls invokes a method of cls', and Equation 4.3 shows a method of cls accesses an attribute of cls'. These relationships are illustrated in Figure 4.2. The binary relationship $D \subseteq CLS \times CLS$ connects classes to classes based on the one or more of the relationships described by Equations 4.1, 4.2 and 4.3.

Definition 9 *For two classes $cls, cls' \in CLS$, cls is directly dependent on cls' if and only if $cls.D.cls'$*

Definition 10 *For two classes $cls, cls' \in CLS$, cls is indirectly dependent on cls' if and only if $cls.D.D^{-1}cls'$*

For example, for three classes $cls, cls', cls'' \in CLS$, cls is the return type of cls' (Equation 4.1) and a method of cls' invokes a method of cls'' (Equation 4.2); therefore, cls is directly dependent on cls' and indirectly dependent on cls''.

4.2.2 Database Relationships

A significant part of a system's business logic is incorporated in the database relationships, and these relationships complement the ones which are visible at the source code level.

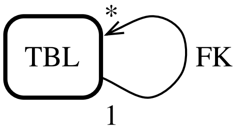


Figure 4.3: Database table with the foreign key relation

The main entity at the database level is the table, and in this analysis the set of all tables for a software system is denoted by TBL. The binary relation $FK \subseteq TBL \times TBL$ connects tables to tables based on the foreign keys. Figure 4.3 illustrates this relationship. As in the case of source code, for two tables $t, t' \in TBL$, the direct and indirect relationships in the database can be defined as follows:

Definition 11 *t has a direct relation to t' if and only if $t.FK.t'$.*

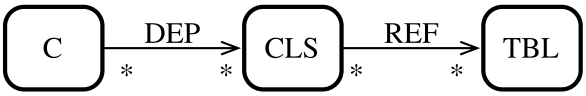
Definition 12 *t has indirect relation to t' if and only if $t.FK.FK^{-1}.t'$*

While foreign key relations among tables are there to model a specific aspect of the domain, indirect relations between tables should suggest how different concepts are bound together.

4.2.3 Architectural Dependencies

Two components are considered to be architecturally connected either by direct or indirect dependencies between the classes behind them, or by direct or indirect relationships between the tables accessed by these classes.

Figure 4.4 shows the relations between the components (C), classes (CLS) and tables (TBL) of ADempiere. These elements are related by $DEP \subseteq C \times CLS$ which represents classes that a component depends on, and $REF \subseteq CLS \times TBL$ which represents tables that a class reads or writes to.



C: components, CLS: classes, TBL:tables

Figure 4.4: Relationships between software elements

Definition 13 *For two components $c, c' \in C$, they are architecturally connected if and only if one or more of the following relationships exists between them:*

$$c.DEP.DEP^{-1}.c' \tag{4.4}$$

$$c.DEP.D.DEP^{-1}.c' \tag{4.5}$$

$$c.DEP.D.D^{-1}.DEP^{-1}.c' \tag{4.6}$$

$$c.DEP.REF.REF^{-1}.DEP^{-1}.c' \tag{4.7}$$

$$c.DEP.REF.FK.REF^{-1}.DEP^{-1}.c' \tag{4.8}$$

$$c.DEP.REF.FK.FK^{-1}.REF^{-1}.DEP^{-1}.c' \tag{4.9}$$

This definition describes all direct and indirect dependencies through classes or tables behind components. Equation 4.4 defines a connection between two components based on shared classes. Equation 4.5 connects two components considering direct dependencies between their shared classes. Equation 4.6 considers indirect dependencies between classes to connect two components. Equation 4.7 defines a connection between two components based on their shared database tables. Equation 4.8 and Equation 4.9 consider direct and indirect dependencies between database tables which connect two components.

4.2.4 Tracing Dependencies in ADempiere

ADempiere is large scale systems with more than two million lines of code and a multi-tire architecture. For a system at this scale the reverse engineering of its source code and database is too big to perform by manual analysis of the system, and necessitate a proper tool support.

At the time of writing this thesis, Moose [142] is one of the most equipped platforms for analysis of object oriented systems. Moose uses a language independent meta model called *FAMIX* [174]. For the purpose of this study, the meta-model is extended to describe the static structure of code entities with information about database relationships [5]. The extended entities are highlighted in bold in Figure 4.5 with the following relationships: The relation *maps* associate a table to a class where the class represents the table, for example, a table might be mapped to a class at the data layer. The same happens to the class attributes that *map* table columns. The relation *Access* represents class methods accessing database tables. The relation *Reference* represents connections among table columns achieved using a foreign key constraint.

For ADempiere, the entities and the relationships between them are derived in two data sources: Firstly, Classes, Methods, Attributes and relationships between them are derived from the source code. Secondly, tables and columns are mapped to classes and attributes by analysis of the *application dictionary* which is a meta-data collection for user interface elements such as windows, forms, fields and validation rules[4]. In ADempiere, the *application dictionary* is stored inside the database, and can be accessed by SQL queries.

In this study, a window of ADempiere is considered as a user interface component (UIC). Table 4.6 shows the number of dependencies between UICs at multiple layers of the code and the database. The results shows that there are 16,968 architectural dependencies between

[4] http://www.adempiere.com/Application_Dictionary

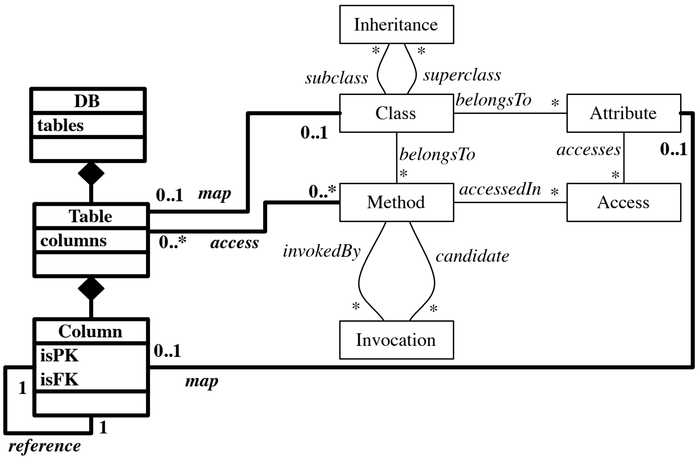


Figure 4.5: FAMIX meta-model including extended entities for relational databases.

	Number of Dependencies
Source Code Dependencies	14,898
Direct Database Relationships	8,132
Indirect Database Relationships	12,178
Architectural Dependencies	16,968

Figure 4.6: Number of dependencies between ADempiere windows at multiple layers of the source code, the database, and the aggregated results at the architecture level .

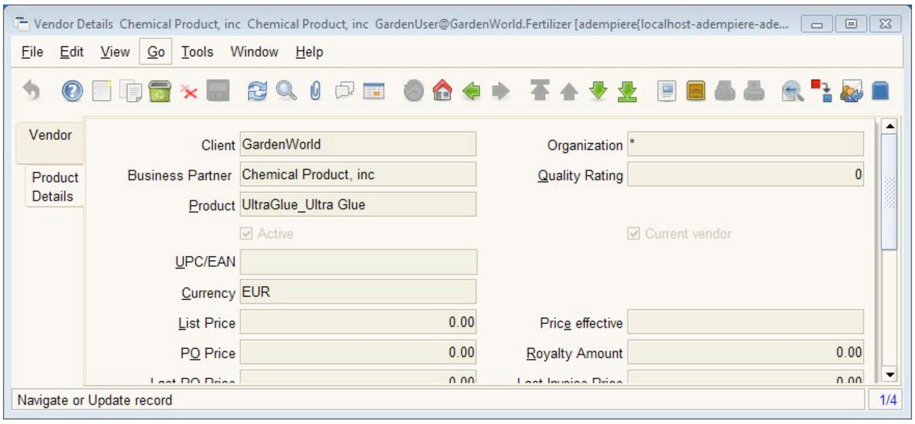
UICs derived based on Definition 13.

The next section describes the domain-level relationships in ADempiere, and Section 4.4 shows how well these relationships can reflect architectural dependencies.

4.3 Domain Analysis

In this section, we describe the domain model of ADempiere, how to process this model to identify domain-based coupling between UICs, and how to create a domain-based coupling graph from this model.

In the last chapter, we described three elements of the domain model: domain variable, data field and user interface component (UIC). At the presentation layer, ADempiere composed of number of windows, where each window includes one or more tabs, and each tab has multiple data fields. In ADempiere, data fields mostly represent domain variables, tabs



Vendor Details Chemical Product, inc Chemical Product, inc GardenUser@GardenWorld.Fertilizer [adempiere{localhost-adempiere-ade...

Client GardenWorld

Organization *

Business Partner Chemical Product, inc

Quality Rating 0

Product UltraGlue_Ultra Glue

☑ Active

☑ Current vendor

UPC/EAN

Currency EUR

List Price 0.00

Price effective

PO Price 0.00

Royalty Amount 0.00

Navigate or Update record

1/4

Figure 4.7: ADempiere : Vendor Details

represent domain functions, and windows represent UICs.

The following examples demonstrate how to measure the domain-based coupling between UICs, and how to predict the architectural dependencies using the domain-based coupling.

4.3.1 Example 1: Measuring Domain-Based Coupling

This example demonstrates how to measure the domain-based coupling between UICs. In ADempiere, *Vendor Details* (Figure 4.7) and *Import Product* are two UICs which we use in this example to demonstrate how derive domain relationships. *Vendor Details* (c_1) has 2 domain functions, and in total 25 domain variables, as follows:

$\mathrm{c}_1.HAS$ = { Edit Vendor, Edit ProductDetails }.

$\mathrm{c}_1.HAS.USE$ = { DeliveryTime, BusinessPartner, CostPerOrder, Currency, Vendor, Manufacturer, ListPrice,... }.

Import Product (c_2) contains one domain function and 42 domain variables as follows:

$\mathrm{c}_2.HAS$ = { Import Products }.

$\mathrm{c}_2.HAS.USE$ = { CostPerOrder, PriceEffective, Weight, BusinessPartner, SKU, UOM, Processed, Royalty,... }.

There are 18 common domain variables between these UICs as follows:

> $\mathrm{c}_1.\mathit{HAS.USE} \cap \mathrm{c}_2.\mathit{HAS.USE} = \{$ BusinessPartner, CostPerOrder, Currency, Discontinued, DiscontinuedAt, ListPrice, Manufacturer, MinOrderQty, OrderPackQty, PartnerCategory, PartnerProductKey, POPrice, PriceEffective, Product, PromisedDeliveryTime, Royalty, UOM, UPC/EAN $\}$.

and in total 49 (42+25-18) variables used by either of these UICs; thus:

- Common domain variables (Definition 5): $\vartheta(c_1, c_2) = 18$
- Symmetric coupling weight (Definition 7): $\omega(c_1, c_2) = 18/49 = 0.37$

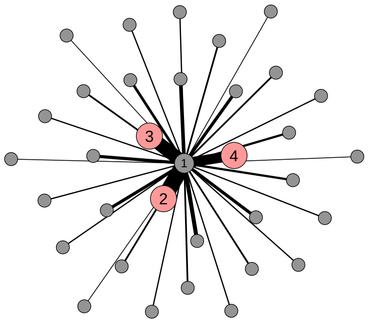
Note: Architectural dependency (Definition 13) is a symmetric relationship; hence, in this study we only focuses on symmetric domain-based coupling, and ignores asymmetric domain-based coupling.

4.3.2 Example 2: Predicting Dependencies

Now that we have explained the domain definitions, let's demonstrate how to use them for predicting dependencies. Imagine if a domain expert considers asking for an enhancement to *Vendor Details* (c_1), then given the domain information of ADEMPIERE, she can derive common domain variables (ϑ) among c_1 and other UICs similar to what was described in the previous example.

Figure 4.8 shows there are 33 UICs for which the coupling weight with c_1 is greater than a given threshold $\omega \geq 0.5$. The selected threshold is applied to avoid weak results which do not likely lead to any architectural dependencies. This also reduces the density of the resulting domain-based coupling graph and makes it more readable. The results are illustrated (Figure 4.8) as a weighted graph where the edge width is proportional to ω, and edge length is proportional to $1/\omega$, *i.e.,* the stronger the coupling weight, the thicker is the edge and the closer the node to the center (c_1).

The top 3 closest UICs are: *Import Products* (c_2), *Spare parts,* (c_3) and *Product Planning* (c_4), with the domain-based coupling values of 0.37, 0.32 and 0.25 respectively. Investigating



Nodes represent UICs and edges represent domain-based coupling. The tagged nodes are (1) Vendor Details, (2) Import Products, (3) Spare Parts and (4) Product Planning. Node size has no implication, but edge width is proportional to ω and edge length is proportional to $1/\omega$. For readability, the graph only contains $c_1.CNC$, excluding edges between other nodes.

Figure 4.8: Domain-Based Coupling Graph of Vendor Details

the source code shows that all three UICs are connected to *Vendor Details* by source code dependencies.

This two examples demonstrated how a domain expert can analyse ADempiere, and derive the domain-based coupling graph. In Section 4.4, we will present the comprehensive evaluation on how accurate such a graph identifies architectural dependencies in ADempiere.

4.3.3 Expectation Maximisation Clustering

In the last example, a threshold value for domain-based coupling was used to identify highly coupled components. The threshold value can be selected manually based on the system characteristics like distribution of the coupling values, or by graph visualisation [4]. However, the manual approach is subject to human errors and not scalable for large datasets. In order to address this limitation, in this study, we use a clustering technique to identify highly coupled components automatically.

The aim of clustering is to group a given set of objects so that similar objects are grouped together and dissimilar objects are kept apart. There are many different multi-dimensional clustering techniques [127]. We use a statistical clustering technique called Expectation Maximization (EM) since it can automatically find the optimum number of clusters [49].

The main idea behind EM is fitting the parameters of a distribution model by using training data. The EM algorithm assigns a probability distribution to each instance of the number of *common variables* (ϑ), which indicates the probability of the instance belonging to each of the generated clusters. In Section 4.4, we discuss how EM clustering improves the precision of identifying dependencies.

4.4 Evaluation

Now that both architectural dependencies and domain-based coupling between UICs in ADempiere have been discussed, let's evaluate how accurately domain-based coupling can approximate architectural dependencies.

4.4.1 Evaluation Setup

For a given UIC, $c \in C$, we test the query $AN = q(c, E)$ where the expected outcome $E \subseteq C$ is the set of UICs which have architectural dependencies to c, and the returned answer

$$AN = \{c_i | c_i \in C, \vartheta(c, c_i) > 0\}$$

is the set of UICs which are coupled with c at the domain level. We describe the outcome of such a query as follows:

TP $=|E \cap AN|$ shows the number of correctly identified dependent components.

TN $=|C \backslash \{AN \cup E\}|$ shows the number of correctly identified independent components.

FP $=|AN \backslash E|$ shows the number of incorrectly predicted dependent components.

FN $=|E \backslash AN|$, shows the number of incorrectly predicted independent components.

We use the well-known definitions of *precision* (P_q) and *recall*(R_q) to evaluate the outcomes of a given query:

$$P_q = \frac{TP_q}{TP_q + FP_q} \qquad R_q = \frac{TP_q}{TP_q + FN_q}$$

Precision and *recall* only evaluate TP. In order to describe both TP and TN, we measure *accuracy* (A_q) which is the degree of closeness of results to the preferable values where all dependent and independent components are correctly identified. *Accuracy* [128] is defined as follows:

$$A_q = \frac{TP + TN}{TP + FP + FN + TN}$$

The higher the *accuracy*, the closer the prediction outcomes to the perfect results where both FP and FN are equal to zero.

4.4.2 Macro Evaluation

In order to evaluate the results for all UICs in ADempiere, we take the mean value of measurements of all queries as

$$f_M = \frac{1}{n}\sum_{i=1}^{n} f_{q_i}$$

where f is one of these measurement functions: TP, TN, FP, FN, R, P or A.

4.4.3 Likelihood

One application of domain-based coupling might be notifying software maintainers of possible dependent components when they browse a list of UICs. To assess the usefulness of such notifications, we measure the *likelihood* (L) whether at least one of the top three, five or ten returned results have architectural dependencies. More formally if $AN_{c,n}$ shows the top n results for a component c, then

$$L_n = \frac{|\{c|c \in C, AN_{c,n} \cap E_c \neq \emptyset\}|}{|\{c|c \in C, E_c \neq \emptyset\}|}$$

The *likelihood* function distinguishes between the topmost results and the entire returned result set.

	#Dep.	TP_M	FN_M	TN_M	FP_M	R_M	P_M	A_M	L_3	L_5	L_{10}
COD	14,898	28	15	226	78	0.68	0.27	0.73	0.69	0.74	0.78
DDR	8,132	19	4	237	87	0.77	0.20	0.74	0.59	0.66	0.75
IDR	12,178	22	13	227	85	0.71	0.23	0.72	0.51	0.56	0.62
ARC	16,968	31	18	223	76	0.64	0.30	0.73	0.72	0.78	0.84

Legend: #Dep.: Number of dependencies, COD: Source code dependencies, DDR: Direct database relationships, IDR: Indirect database relationships, ARC: Architectural dependencies .

Table 4.1: Prediction Results

4.4.4 Results: Searching For Source Code Dependencies

ADempiere contains 347 UICs. The source code analysis revealed $14,898$ indirect dependencies and no direct dependencies among classes behind these UICs. We compared these dependencies with the domain-based coupling graph to evaluate how accurately source code dependencies can be derived from domain information.

The results are presented in Table 4.1. On average for a given UIC, 28 connected UICs by source code dependencies identified correctly whilst 15 UICs with source code dependencies are incorrectly described as independent components, and 78 independent UICs are falsely called to have source code dependencies. These results lead to average *recall* equal to 0.68 and average *precision* equal to 0.27.

Also the *accuracy* of the dependency prediction is equal to 0.73, implying that for more than 7 out of 10 UICs, our prediction method correctly identified if two UICs are dependent or independent at the source code level.

The likelihood of discovering source code dependencies in the top three coupled UICs is 69%, and it will increases to 78% for the top ten UICs.

Summary: *On average* 68% *of UICs connected by source code dependencies are discovered correctly, while for* 78% *of queries the top ten results contains one or more source code dependencies.*

4.4.5 Results: Searching For Database Relationships

The database analysis of ADempiere showed that there are 8,132 direct and 12,178 indirect relationships among data tables behind UICs.

We queried these relationships using the domain-based coupling, and the results are presented in Table 4.1. On average for a given UIC, 19 directly related UICs and 22 indirectly related UICs are identified correctly. The results show only 4 false negatives for direct relationships which is more than three times lower than 13 false negatives for indirect relationships. However, the number of false positives are similar: 87 and 85 for direct and indirect relationships respectively.

Comparing the results between direct and indirect relationships shows that for direct relationships the *recall* is slightly higher (0.77 vs 0.71) whilst the *precision* is slightly lower (0.2 vs 0.23). The *accuracy* values for both relationship types are more than 0.7, suggesting that for 7 in 10 UIC pairs, their relationship state is identified correctly.

In addition, validating the topmost results shows that the likelihood of database relationships in the top three results is 51% for direct and 59% for indirect relationships. Also the likelihood of indirect relationships increases to 75% for the top ten results.

Summary: *On average up to 77% of database relationships can be derived from domain information, and for 75% of queries, the top ten results contain at least one database relationship.*

4.4.6 Results: Searching For Architectural Dependencies

The analysis of the source code and the database of ADempiere shows $16,968$ architectural dependencies (Definition 13).

We evaluated how accurately a domain expert can predict if there is at least one architectural dependency between any given pair of UICs. The results are presented in Table 4.1. On average for a given UIC, 31 dependent UICs, and 223 independent UICs are identified correctly using domain information. However, 18 dependent and 76 independent UICs are incorrectly placed in the opposite dependency state. These results lead to an average *recall* of 0.64 and *precision* of 0.30. The mean *accuracy* of the predictions is 0.73, suggesting that for 7 in 10 UIC pairs, their dependency state is identified correctly.

In addition, the likelihood of discovering an architecturally dependent UIC pair in the top three results is 72%. This likelihood will increase to 84% for the top ten results.

Summary: *On average 64% of architecturally dependent UICs are discovered using domain information, and the likelihood of discovering a correct architectural dependency in the top ten predictions is 84%.*

4.4.7 Improving Precision

The prediction results for architectural dependencies (Table 4.1) show that the average precision is 0.30. In order to improve the *precision*, we utilised the expectation maximisation technique (Section 4.3.3) to filter out weakly coupled pairs, with the assumption that UICs with strong domain-based coupling are more likely to have architectural dependencies.

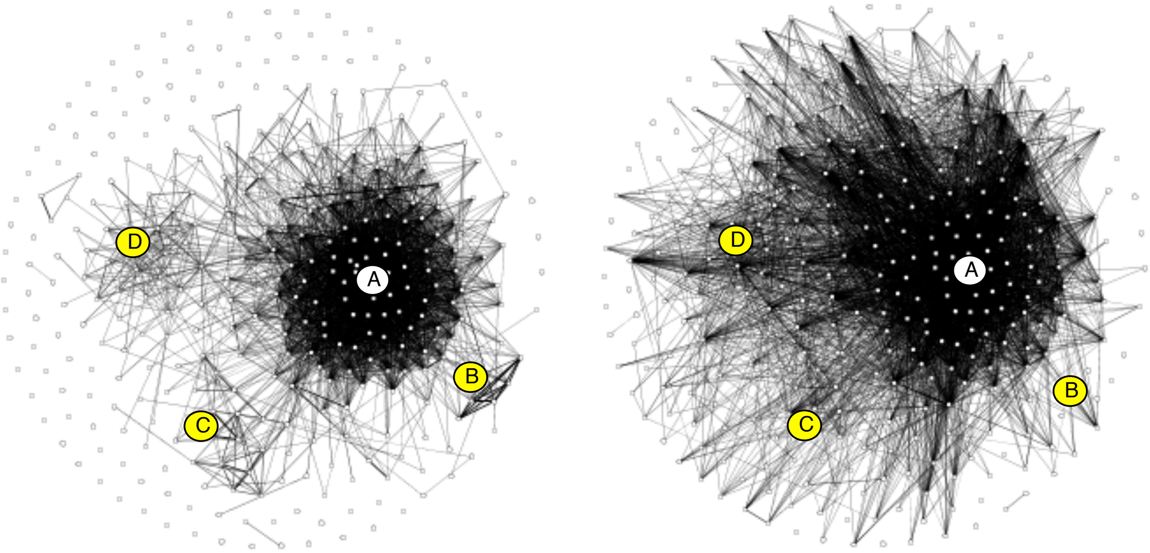
	R_M	P_M	A_M
Source Code Dependencies	0.29	0.68	0.88
Direct Database Relationships	0.40	0.57	0.89
Indirect Database Relationships	0.27	0.61	0.93
Architectural Dependencies	0.23	0.70	0.87

Table 4.2: Prediction Results Using EM Clustering

Table 4.2 shows the improved results. The mean *precision* for architectural dependencies is increase from 0.30 to 0.7, and the mean *accuracy* is increased from 0.73 to 0.87.

However, these improvements are achieved at the expense of the reduction in *recall*. While the value of *precision* is more than doubled, the value of *recall* decreased almost three times (from 0.64 to 0.23). This implies that there are a number of architectural dependencies between UICs which have no strong coupling at the domain level.

Summary: *By using expectation maximisation technique, precision can be improved up to 0.7. However, it is a trade-off between precision and recall.*



(a) Domain-based coupling graph

(b) Architectural dependency graph

Legend: Nodes are the UICs of ADempiere in both graphs. Left: Edges are domain-based coupling (Definition 7) which are selected by Expectation Maximisation (Section 4.3.3). Right: Edges are architectural dependencies (Section 4.2). Tags (A, B, C and D) are concentration areas.

Figure 4.9: Domain-based coupling vs architectural dependencies

4.4.8 Visual Comparison

The domain-based coupling graph (Figure 4.9a) is visualised using Fruchterman and Reingold's [54] force-based graph layout in three steps: first, the graph is created based on Definition 7; second, the exception maximisation (EM) technique (Section 4.3.3) is applied; third, the derived graph is visualised by the force-based layout algorithm.

In order to compare the domain-based coupling graph with the architectural dependencies, the edges from Figure 4.9a are replaced with the architectural dependencies without changing the location of nodes. The resulting graph (Figure 4.9b) illustrates the distribution of the architectural dependencies in compare to the domain-based coupling.

The comparison between Figure 4.9a and Figure 4.9b shows that the most populated cluster (tagged by A) in the domain-based coupling graph has the biggest number of architectural dependencies. However, the number of architectural dependencies decreases in the clusters with poor domain-based coupling (B, C and D). In addition, there are a number of architectural dependencies where there is no domain-based coupling, illustrating that not all dependencies can be derived from the domain-based coupling graph.

4.4.9 Discussion

In this evaluation, we reported that on average 64% of architectural dependencies could be derived from domain-based coupling graph. The accuracy of the prediction is on average 0.73 while the precision is 0.30. The precision can be increased up to 0.7 using expectation maximisation technique. Trading off precision for recall would be a good approach if one would build a tool that would be used by maintainers: having too many false positives might deter the users of such a tool.

In addition, we demonstrated how domain-based coupling could be used to inform software maintainers while they browse software UICs. The results show the likelihood of discovering architectural dependencies among the top ten coupled UICs is 84%. Given that these results are obtained without looking at the source code or the database, they are quite promising. On the other hand in the current form, domain-based coupling analysis cannot completely replace the source code analysis.

4.5 Threats to Validity

In this section, we discuss the threats to validity of our findings, and how we addressed them.

Threats to *external validity* are concerned with generalisation of our findings. Although we performed our evaluation on a large-scale enterprise system which is representative of the state of the art enterprise systems developed in Java, we are aware that more studies are required to be able to generalise our findings.

Threats to *construct validity* are concerned with the quality of the data we analysed, and the degree of manual analysis that was involved. The domain information typically is provided by the domain experts using a manual data collection process. To minimise the risk of human error, we extracted the relationship between domain variables and UICs from user manuals and help documents. In ADempiere, this information is stored in the database. We only used manual inputs from domain experts to confirm this information and kept the manual additions and alterations to a minimum.

One other factor that could affect the validity of the results is the granularity used to look at the selected UICs. We chose windows as UICs. Each window contains multiple tabs and each tab provides one or more functions. Different results could be achieved if the evaluation was performed at the tab level, or module level.

4.6 Open Issues

This chapter aimed to provide an empirical study on how domain-based coupling can be used to predict architectural dependencies. Given the limitations of the performed case study, we propose the following open questions to be addressed in the future studies:

- In this study, we have only examined the dependencies between application windows; however, there are finer-grained UICs (*e.g.,* tabs) in ADEMPIERE. The research questions to be answered are: What is the efficient granularity level? What properties of UICs affect the results (*e.g.,* size and complexity)?
- The other area of future investigation is the impact of different domains on the results. ADEMPIERE contains various modules which provide functions of different domains like ERP, CRM and Asset Management. Some of the research questions to be answered are: What are the factors in these domains (*e.g.,* complexity) that affect the prediction results? Does distinguishing between these domains make the predictions more accurate?

4.7 Summary

In this chapter, we demonstrated how domain information could be used to predict architectural dependencies, and assist software maintainers in searching for connected components at the source code or the database layers. Our proposed approach for predicting dependencies promises independence from software implementation and simplicity and usability for non-technical domain experts. Hence, it can assist managers and consultants to take decisions about software changes without the support of the developers.

The proposed dependency analysis method is based on relationships between software domain information and user interface components (UIC), modelled as a weighted graph. We demonstrated how such a model could assist predicting dependencies with a case study on a large-scale enterprise system, called ADempiere. We derived architectural dependencies as a set of source code and database dependencies, and compared them with the domain-based coupling between UICs. The results show that on average 68% of the source code and up to 77% of the database dependencies could be derived from the domain-based coupling. The accuracy of such predictions is on average more than 70%, implying that for 7 out of 10 component pairs their dependency state is identified correctly.

The results promise that domain information might be used to predict the existence of architectural dependencies, and the accuracy of these predictions could support maintenance activities such as change impact analysis. However, at the current stage, this approach cannot replace the source code analysis or the database analysis.

Chapter 5

Predicting Change Propagation

> The real world is of primary importance in any software system. It is, ultimately, the originating point of the system. The first attempts to introduce specific software systems are usually those systems that imitate what already exists in the real world. This imitation is the starting point from which the system evolves.
>
> *A Nontraditional View of the Dimensions of Software Evolution* [148]
> Dewayne E. Perry

Change propagation is the phenomenon whereby a change to one part of a system affects other parts and leads to subsequent changes (Section 2.6). Such propagation poses a major risk to software maintenance by leading to code decay, bugs and unforeseen extra development costs. In Chapter 2, we described the three major approaches to change impact analysis including document-based, source-based and history-based impact analysis methods. Although these methods can accurately identify change propagation based on tracking dependencies or co-change coupling among software elements, they are less efficient for hybrid systems which are composed of multiple programming languages or include subsystems based on different technologies. In Chapter 4, we demonstrated that domain information can be used to approximate software dependencies independent from software implementation and even without access to the software source code, design documents and maintenance history. In this chapter, we will explore the application of domain information in predicting change propagation. Specifically, we aim to answer the third and fourth research questions[1] of this thesis: (RQ3) How accurately can we predict change propagation using domain-based coupling? (RQ4) How does such a prediction compare with the well-established co-change coupling metric derived from maintenance history?

We report on a case study of a significant enterprise system where we perform both domain analysis and history-based analysis, and we compare the results. In addition, we demonstrate how domain-based coupling, like the history-based approach, can support maintenance tasks by avoiding bugs resulting from imperfect change propagation. We examine the following experimental questions in this study :

- **Correlation**. To what extent can the domain-based coupling between pairs of UICs be correlated with the history-based change coupling?
- **Error prevention**. Given a single transaction involving modification of multiple UICs, if a single UIC is missing from the transaction, how reliably can the domain-based coupling be used to find the missing component?
- **Estimating change scope**. Given a change to a single component, how reliably does domain-based coupling determine what other components will most likely be affected by the change?

The rest of this chapter is organised as follows: Section 5.1 introduces the system under analysis. Section 5.2 describes the evolutionary coupling approach and the analysis results. Section 5.3 shows the derived domain-based coupling for the system under analysis. Section 5.4 explains how we measure change propagation, and Section 5.5 evaluates how accurately the domain-based coupling approximates the change propagation. Section 5.6 discusses the threats to the validity of our findings. Section 5.7 describes the further areas of investigation, and finally Section 5.8 summarises this chapter.

[1] The complete list of research questions is provided in Section 1.2.

5.1 Case Study: BEIMS

The system under analysis in this case study is a facility management system called BEIMS[2]. At the time of this study (January 2009), BEIMS has more than a third of the market share for facility management systems used by Australian and New Zealand universities, hospitals and casinos. Mercury Computer Systems (Australia) Pty Ltd designed and developed the first version of BEIMS (Figure 5.1a) in 1989 using a 4GL. In 1998 the fifth generation of BEIMS was redeveloped for Windows platforms and extended with more than 100 individual subsystems and custom developed programs.

Subsystem name	ID	Source code lines
Asset Management System	AMS	15,700
Cost Control System	CCS	10,959
Information Setup System	ISS	29,330
Planned Maintenance System	PMS	13,857
Work Order System	WOS	34,164
		104,010

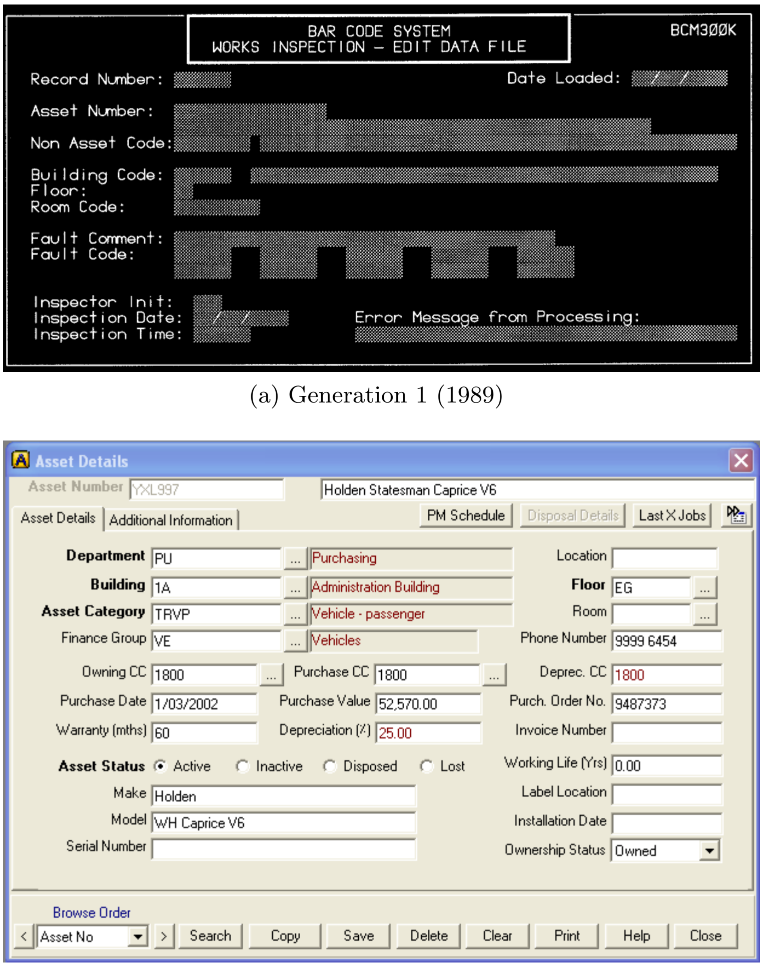
Table 5.1: BEIMS Core Subsystems

For our case study we chose the five core subsystems of BEIMS that are installed and used by all BEIMS clients. As demonstrated in Table 5.1, these together contain more than 100,000 lines of code. We looked at the 12 years maintenance history of these programs and studied how they have evolved over their life cycle. Given the rich maintenance history of BEIMS, we can evaluate to what extent the domain-based coupling between UICs correlates with what can be predicted from the change history of the software.

5.2 History-Based Analysis

The history of source code changes can reveal patterns in modifying software components as part of maintenance activities. In Section 2.7.3, we described that the evolutionary coupling [200] measures the co-change relationships between software elements based on their maintenance history. In this study, we will use the evolutionary coupling to identify change propagation in the recorded maintenance history of BEIMS.

[2] Building and Engineering Information Management System.



```
                BAR CODE SYSTEM                          BCM300K
         WORKS INSPECTION - EDIT DATA FILE

Record Number:                                  Date Loaded:   /  /

Asset Number:

Non Asset Code:

Building Code:
Floor:
Room Code:

Fault Comment:
Fault Code:

Inspector Init:
Inspection Date:   /  /          Error Message from Processing:
Inspection Time:
```

(a) Generation 1 (1989)

Asset Details

Asset Number YXL997 — Holden Statesman Caprice V6

Department	PU	Purchasing		Location	
Building	1A	Administration Building		**Floor**	EG
Asset Category	TRVP	Vehicle - passenger		Room	
Finance Group	VE	Vehicles		Phone Number	9999 6454
Owning CC	1800	Purchase CC	1800	Deprec. CC	1800
Purchase Date	1/03/2002	Purchase Value	52,570.00	Purch. Order No.	9487373
Warranty (mths)	60	Depreciation (%)	25.00	Invoice Number	

Asset Status ● Active ○ Inactive ○ Disposed ○ Lost — Working Life (Yrs) 0.00

Make	Holden	Label Location	
Model	WH Caprice V6	Installation Date	
Serial Number		Ownership Status	Owned

Browse Order: Asset No

(b) Generation 5 (2009)

Figure 5.1: BEIMS from 1989 to 2009

Let $T \subseteq 2^{UIC}$ be a set of *transactions* where each transaction is defined by a set (of *changed components*). Following standard data mining approaches [200, 6], define an *(association) rule* $x_1 \Rightarrow x_2$ for two disjoint sets x_1 and x_2 (interpretation: if a programmer changes x_1 then she has also changed x_2). The *frequency* of a (changed component) set x in T is defined as

$$freq(T, x) := |\{t | t \in T, x \subseteq t\}|$$

and the *confidence* of a rule $x_1 \Rightarrow x_2$ (interpreted as the strength of the rule) is defined as

$$\mathit{conf}(T, x_1 \Rightarrow x_2) := \frac{\mathit{freq}(T, x_1 \cup x_2)}{\mathit{freq}(T, x_1)}.$$

For BEIMS, Microsoft Source Safe is used as the source code version control. We use the sliding window technique proposed by Zimmermann and Weißgerber to recover transactions from Source Safe [198], since it does not track which files have been modified in a transaction (committed changes in conjunction). The history of changes to a given file can be derived from Source Safe as a set of change log records whereby each record represents a single *check in* (commit) command and contains user name, comment (message) and the differences between two subsequent revision of a file.

At the time of analysis there were 78,632 change log records for the five subsystems of BEIMS. Some of these records are the result of a labelling action in Source Safe. Labels have been used to tag all files with a given time (typically a released version of BEIMS) for the purpose of creating branches. Labelling records are not related to any modification to file contents, and the transactions derived from them do not imply any code change coupling between files. We are only interested in the coupling between source code files result of conjunction maintenance, therefore we removed the labelling records to avoid false positive results. The remainder is 10,912 records, yielding 4,456 transactions.

5.3 Domain Analysis

We analysed the behaviour of all UICs for the five BEIMS' subsystems, based on information provided in the software functional specification. The BEIMS' functional specification describes its subsystems from the perspective of a domain user including the behaviour of each screen at the domain level, that is, actions, interactions and provided information. The functional specification is derived from existing user manuals and expert user knowledge about system behaviour, as described earlier [4]. The analysis result is collected in the form of a dependency matrix (Definition 4) consisting of 68 UICs and 381 domain variables whereby for 731 elements $M_{c,v} = 1$. From the dependency matrix, we derived symmetric and asymmetric weights (Definitions 7 and 6) for pairs of UICs in each subsystem as summarised in Table 5.2 with the aggregated statistical information for each subsystem.

		Min	Max	Mean	Std. Dev.
AMS	ω_a	0.00	1.00	0.11	0.18
	ω	0.00	0.38	0.06	0.09
CCS	ω_a	0.05	0.45	0.23	0.14
	ω	0.04	0.27	0.11	0.09
ISS	ω_a	0.00	1.00	0.04	0.12
	ω	0.00	0.25	0.01	0.04
PMS	ω_a	0.00	1.00	0.44	0.30
	ω	0.00	0.88	0.23	0.20
WOS	ω_a	0.00	1.00	0.24	0.25
	ω	0.00	0.77	0.12	0.13

Table 5.2: BEIMS: Domain Analysis Results

5.4 Change Propagation

Given a change request for modifying (fixing a bug or an enhancement) a UIC, we query other UICs which most likely will be affected by the given change. When changing a given user interface component $c \in C$, we define a function

$$AFC : (C \times C \to \mathbb{R}) \times \mathbb{R} \to (C \times C)$$

which generates a relation between a component c and other components which will be *most likely affected* by a change to c. That is $c.AFC(f, \lambda)$ represents the set of most likely affected components by a change to component c, defined as

$$AFC(f, \lambda) = \{(c, c')|c, c' \in C \wedge c \neq c' \wedge f(c, c') > \lambda\}$$

where f is a function measuring the level of coupling between two components and the λ is a given threshold.

Using Definitions 6 and 7, $c.AFC(\omega_a, \lambda)$ and $c.AFC(\omega, \lambda)$ functions predict the set of components that most likely will be affected by a change to the component c.

5.5 Evaluation

In this section, we evaluate how accurately domain-based coupling can approximate evolutionary coupling that is derived from BEIMS maintenance history.

5.5.1 Evaluation Setup

For assessment of results we follow standard definitions of *precision* (P_q), the percentage of a returned answer which was expected, and *recall* (R_q), the percentage of an expected answer which was returned [77]:

$$P_q = \frac{|A_q \cap E_q|}{|A_q|} \qquad R_q = \frac{|A_q \cap E_q|}{|E_q|}$$

In order to calculate the precision and recall for all queries for a given subsystem, we take the mean value of the precision and recall of individual queries:

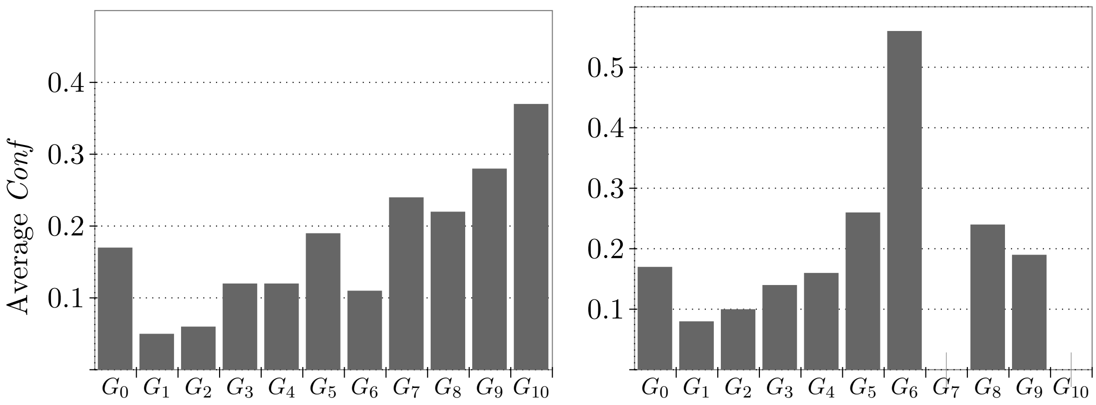
$$P_M = \frac{1}{n}\sum_{i=1}^{n} P_{q_i} \qquad R_M = \frac{1}{n}\sum_{i=1}^{n} R_{q_i}$$

5.5.2 Results: Correlation

The experimental hypothesis is that there is a correlation between evolutionary and domain-based coupling. We evaluate this hypothesis in two stages. Firstly we examine the trend of evolutionary coupling among UIC pairs with respect to the domain-based coupling. Secondly we measure the correlation coefficent between the asymmetric/symmetric weight functions and the *conf* function (Section 5.2).

Average Trend

In the first stage, we grouped the UIC pairs in all BEIMS subsystems by domain-based coupling— first asymmetric weight, then symmetric weight—and measured the average confidence (*conf*) for association rules between all pairs in each group.



(a) Grouped by *asymmetric* weight

(b) Grouped by *symmetric* weight

Legend: G_n = Group of pairs where (a) $n-1 \leq w_a * 10 < n$ and (b) $n-1 \leq w * 10 < n$.

Figure 5.2: Trend of evolutionary vs domain-based coupling

Figure 5.2a illustrates the relationship between *conf* and *asymmetric* weight. Each group G_n consist of pairs $\langle c, c' \rangle$ where $n-1 < w_a(c, c') * 10 \leq n$.

Figure 5.2b illustrates the relationship between *conf* and *symmetric* weight. Each group G_n consists of pairs $\langle c, c' \rangle$ where $n-1 < w(c, c') * 10 \leq n$.

In general average *conf* increases with respect to asymmetric or symmetric weight, i.e., pairs with stronger domain-based coupling have greater confidence levels for evolutionary coupling. There are exceptions to this trend that we will discuss later in this section.

Table 5.3 shows the number of pairs in each group. In general the number of pairs decreases as domain-based coupling increases, i.e., most pairs are weakly coupled at the domain-level, and only few pairs have strong domain-based coupling.

As illustrated in Figure 5.2, the first exception in the trend of average *conf* increasing with domain-based coupling is at G_0. In comparison to nearby groups (G_1, G_2) the expected average *conf* for G_0 is a value close to zero; however, the actual value is 0.17 (true in both Figures 5.2a and 5.2b). That there are 414 asymmetric pairs (Figure 5.2a) in this group, suggests that not all change couplings can be derived from domain-based coupling. Change logs show some changes to the code are motivated by refactoring, and initiated by programmers where there are no bug reports or enhancement requests.

G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}
414	124	103	80	33	15	36	15	8	5	13

(a) Group size (asymmetric weight)

G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}
207	123	54	26	7	2	2	0	1	1	0

(b) Group size (symmetric weight)

Legend: G_n = Group of pairs where (a) $n-1 \leq w_a * 10 < n$ and (b) $n-1 \leq w * 10 < n$.

Table 5.3: Size of groups of UIC pairs

The second exception is visible in Figure 5.2b, which shows a correlation between the symmetric weight and average *conf* maximised at $w \leq 0.6$ (G_6). The exceptions to this trend are in G_8 and G_9, each with a single pair (Table 5.3b) and lower confidence than G_6.

The first pair is ⟨WorkOrderCompletion (WO), BarcodeWorkOrderCompletion (BWO)⟩. BWO provides the same functionality as WO but instead of typing the work order information, a barcode scanner is used to read the work order and fetch the data. BWO and WO have a clear overlap in their functionality and a lot of duplicated source code. However, WO has been much used by BEIMS users, and subjected to more refinement and minor enhancements. Change logs show that BWO is more often ignored for minor BEIMS revisions, and more subject to changes in major revisions.

The second pair is ⟨MaintenancePlan, AssignTaskToAssets⟩. Both these UICs enable users to add and manage jobs related to assets. However, these UICs provide two different presentations of similar information: the first UIC allows the user to review and manage the jobs in bulk using a calendar view; the second UIC is more focused at the detailed level of individual tasks. The behavioural difference between these components is the main cause of disjoint sets of changes to their source code.

Correlation Coefficient

In the next stage, we examined each subsystem individually, and measured the correlation between *conf* and weights. We used Pearson's correlation coefficient $r_{x,y}$ as a measure of linear dependence between two variables x and y, giving a value between $+1$ and -1 inclusive [163].

Table 5.4 shows on average there is a positive correlation between weights and *conf*, however the correlation is not the same for all subsystems. The behavioural and architectural characteristics of these subsystems affect the pattern which in their source code is changed.

For example, Information Setup System (ISS) is in charge of defining primary data entities in BEIMS, leading to individual UICs containing detailed information unique to each UIC. Also there are number of cases where two UICs hold information about master-detail data entities. Such cases reduce the symmetric weight between UICs; however, the asymmetric weight function can reflect such relationship where one component holds the superset of another component's domain variables.

The other example is Planned Maintenance System (PMS), where there is a negative correlation of -0.04 between the symmetric weight and *conf*, suggesting that evolutionary couplings can not be derived from the symmetric weight function. However, the asymmetric weight function for the same UIC pairs has a correlation of 0.42.

Subsystem	$r(w, conf)$	$r(w_a, conf)$
AMS	0.341	0.392
CCS	0.703	0.45
ISS	0.199	0.534
PMS	-0.042	0.422
WOS	0.481	0.61
Average	0.3364	0.4816

Table 5.4: Correlation coefficient

Summary: *For all five subsystems, confidence level of evolutionary coupling and asymmetric weight are correlated with average Pearson's correlation coefficent of 0.48.* Notably, the level of correlation is not the same for all subsystems, and the variation is greater for symmetric weight.

5.5.3 Results: Error Prevention

In this section, we evaluate to what extent domain-based coupling can be used to prevent software bugs where a programmer changes multiple UICs but misses a single component.

λ	0.9		0.7		0.5		**0.4**		0.3		0.2		0.1		0	
	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M
CROSS	0.07	0.57	0.09	0.45	0.23	0.35	0.39	0.28	0.47	0.14	0.78	0.09	0.84	0.04	0.95	0.02
AMS	0.05	0.76	0.05	0.76	0.14	0.58	0.26	0.56	0.36	0.54	0.66	0.36	0.79	0.21	0.90	0.17
CCS	0	1.00	0	1.00	0	1.00	0.55	0.64	0.58	0.62	0.58	0.62	0.98	0.36	1.00	0.27
ISS	0.05	0.56	0.05	0.56	0.06	0.40	0.12	0.25	0.17	0.23	0.33	0.16	0.46	0.08	0.67	0.07
PMS	0.19	0.35	0.24	0.34	0.53	0.34	0.71	0.25	0.76	0.17	0.84	0.17	0.84	0.16	0.97	0.15
WOS	0.16	0.69	0.40	0.45	0.66	0.29	0.75	0.25	0.81	0.20	0.86	0.16	0.92	0.10	0.97	0.09
Average	0.09	0.66	0.14	0.59	0.27	0.49	**0.46**	**0.37**	0.52	0.32	0.67	0.26	0.80	0.16	0.91	0.13

Legend: λ=Threshold; *CROSS*= Cross program transactions; P_M= Precision; R_M= Recall

Table 5.5: Error prevention using asymmetric weight function

If $\Psi \subset UIC$ represents a set of UICs modified in a given transaction, for each component $c \in C$ we test a query as

$$Q = \Psi - \{c\}$$

For a given Q (and some suitable f and λ), the prediction derived from domain-based coupling is:

$$A = \bigcup_{x \in Q} x.AFC(f, \lambda)$$

Our experimental hypothesis is that for some suitable f and λ, $A = \{c\}$ always.

In order to evaluate this hypothesis, we tested queries for five subsystems of BEIMS using $f = w$ and $f = w_a$ (Definitions 7 and 6). As a benchmark for evaluation, we compared the effectiveness of domain-based coupling to evolutionary coupling with respect to error prevention, repeating the experiment above using $f = conf$.

In addition, we found cross-program transactions containing changes to multiple subsystems. Such transactions are the result of logical coupling [57] between different BEIMS subsystems. The logical coupling between these pairs is not visible at the source code level as there is no code dependency between these subsystems, but, more abstractly, these subsystems are connected at the domain level [57]. As all these subsystems are maintained by a single programming team, it is often the case that programmers are aware of such coupling.

In the first stage, we tested the queries using the asymmetric weight function ($f = w_a$), Table 5.5 shows the results for the five systems. To avoid many false positive results (false warning), we set $\lambda = 0.9$, yielding an average recall of 0.09 and precision of 0.66. This means that only one in 11 queries the $AFC(w_a, 0.9)$ warns the programmer about the missing UIC, and more than half of the results are valid warnings. However, the detailed results show

λ	0.7		0.5		0.3		**0.2**		0.1		0	
	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M
CROSS	0.02	0.62	0.04	0.58	0.14	0.43	0.40	0.20	0.76	0.06	0.95	0.02
AMS	0	1.00	0	1.00	0.18	0.63	0.28	0.57	0.53	0.23	0.90	0.17
CCS	0	1.00	0	1.00	0	1.00	0.55	0.64	0.98	0.36	1.00	0.27
ISS	0	1.00	0	1.00	0	1.00	0.05	0.33	0.19	0.12	0.67	0.07
PMS	0.16	0.52	0.19	0.48	0.35	0.22	0.80	0.17	0.80	0.16	0.97	0.15
WOS	0.12	0.66	0.38	0.60	0.53	0.40	0.70	0.20	0.85	0.11	0.97	0.09
Average	0.05	0.80	0.10	0.78	0.20	0.61	**0.46**	**0.35**	0.69	0.17	0.91	0.13

Legend: $\lambda = Threshold$; *CROSS*= Cross program transactions; P_M= Precision; R_M= Recall

Table 5.6: Error prevention using symmetric weight function

λ	0.9		0.7		0.5		0.4		**0.3**		0.2		0.1		0	
Subsystem	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M
CROSS	0.01	0.96	0.02	0.96	0.07	0.69	0.12	0.59	0.25	0.42	0.37	0.35	0.63	0.21	1.00	0.04
AMS	0	1.00	0	1.00	0	1.00	0	1.00	0.04	0.96	0.26	0.58	0.90	0.22	1.00	0.12
CCS	0	1.00	0	1.00	0.62	0.41	0.98	0.36	0.98	0.36	0.98	0.36	0.98	0.36	1.00	0.27
ISS	0.02	0.80	0.14	0.67	0.34	0.35	0.60	0.32	0.77	0.24	0.81	0.20	0.86	0.11	0.96	0.06
PMS	0	1.00	0	1.00	0.05	0.92	0.08	0.91	0.36	0.62	0.63	0.45	0.93	0.21	0.97	0.15
WOS	0.13	0.81	0.26	0.81	0.31	0.80	0.37	0.72	0.47	0.61	0.65	0.47	0.81	0.23	0.98	0.10
Average	0.03	0.93	0.07	0.91	0.23	0.70	0.36	0.65	**0.48**	**0.54**	0.62	0.40	0.85	0.22	0.99	0.12

Legend: λ=Threshold; *CROSS*= Cross program transactions; P_M= Precision; R_M= Recall

*Table 5.7: Error prevention using evolutionary coupling (*conf*)*

that for the threshold of 0.9, no results were returned for the CCS subsystem. The highest threshold that we can get to cover all subsystems is $\lambda = 0.3$, with the average recall of 0.52 and precision 0.32.

In the second stage, we tested the queries using $AFC(w, \lambda)$ (i.e. based on the symmetric weight function). Notably the maximum value for w in all subsystems is 0.88 (Table 5.6), so we selected the maximum threshold as $\lambda = 0.7$, yielding average recall of 0.05 and precision of 0.8. This means for only one in 20 queries $AFC(w, 0.7)$ returns at least one missing UIC, and 80% of the raised warnings to the programmer are true missing UICs. However, as represented in Table 5.6 there are no results for three subsystems (AMS, CCS, ISS). The highest threshold that can be achieved to cover all subsystems is $\lambda = 0.2$, with average recall of 0.46 and precision 0.35.

Finally, as a benchmark for evaluation, we tested the queries based on $AFC(conf, \lambda)$. The results are represented in Table 5.7. For a strong threshold of 0.9, $AFC(conf, 0.9)$ returns one out of 34 missing UICs with the precision of 0.93, however, the results only include the WOS, ISS and cross-program queries. To achieve a result for all subsystems the maximum threshold of 0.3 yields recall of 0.48 and precision 0.54.

	A_a: Asymmetric function									A: Symmetric function					
λ	**0.7**			0.4			**0.1**			0.4			**0.1**		
Subsystem	R_M	P_M	F_b	R_M	P_M	F_b	R_M	P_M	F_b	R_M	P_M	F_b	R_M	P_M	F_b
AMS							0.48	0.68	0.89				0.33	0.82	0.67
CCS				0.33	1.00	0.67	1.00	0.92	1.00	0.00	1.00	0.00	0.75	1.00	0.75
ISS	0.07	0.25	1.00	0.04	0.44	0.67	0.12	0.45	0.94	0.00	1.00	0.00	0.07	0.57	0.67
PMS				0.75	0.25	1.00	0.84	0.72	1.00	0.00	0.00	1.00	0.67	0.65	1.00
WOS	0.50	0.63	0.50	1.00	0.41	1.00	**0.92**	0.52	1.00	0.60	0.90	0.60	0.68	0.50	1.00
Average	**0.29**	**0.44**	0.75	0.53	0.53	0.83	**0.67**	**0.66**	0.97	0.15	0.73	0.40	**0.50**	**0.71**	0.82

Table 5.8: Result: Estimating the scope of change propagation

Summary: *For all five subsystems, on average 46% of errors arising from imperfect change propagation can be avoided using only domain-level information. This is a promising result in compare with the 48% average error prevention using evolutionary coupling.* In addition, comparison between results for asymmetric and symmetric weight functions suggests that the asymmetric weight function provides better recall; however, more precision can be achieved at the expense of recall using the symmetric weight function.

5.5.4 Results: Estimating Change Scope

In this section, we evaluate to what extent change propagation can be estimated using domain-based coupling. Based on evolutionary couplings a set of components can be derived which are coupled to a given UIC with the confidence greater that a given threshold. The hypothesis is that such a set can be derived from the domain-based coupling.

For $c \in C$, we define a query as a tuple $q = \langle c, \lambda \rangle$ where λ is the minimum required level of confidence. The expected set of affected components by a change to c defined as

$$E = c.AFC(conf, \lambda)$$

We used asymmetric and symmetric weight functions (Definitions 6 and 7) with the same threshold as *conf* to derive the following answers:

$$A = c.AFC(w, \lambda), \;\; A_a = c.AFC(w_a, \lambda)$$

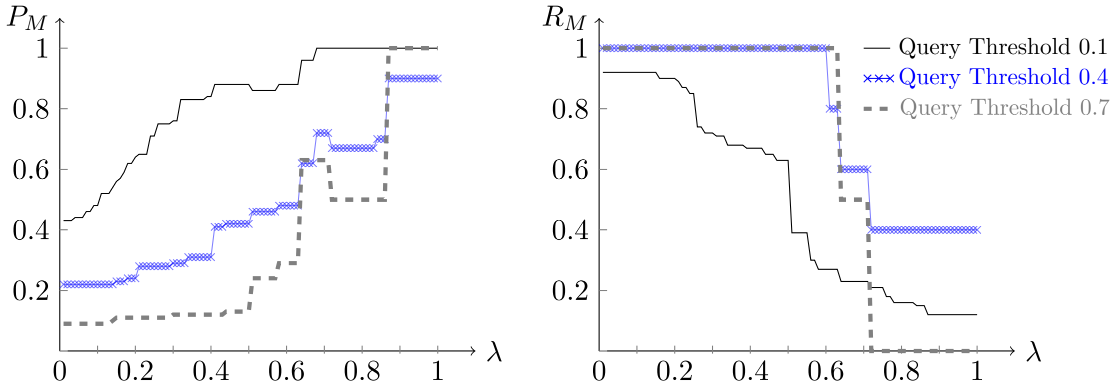
Where Q is the set of queries for a system, we measured the percentage of the queries where our approach can give at least one recommendation Q^* as *feedback* $= |Q^*|/|Q|$.

Table 5.8 shows the results for the five subsystems of BEIMS with three threshold levels 0.7, 0.4, and 0.1, each resulting in a different level of *conf* for change propagation.

A strong threshold of $\lambda = 0.7$ yields empty query sets for three subsystems AMS, CCS and PMS, where the maximum *conf* values are 0.33, 0.60, 0.66 respectively. For the two other subsystems $AFC(w_a, \lambda)$ returns at least one UIC for three out of four queries (*feedback*=0.75), and on average for each query 29% of answers were correct with precision of 44%.

For a threshold of $\lambda = 0.1$, there are queries derived from all subsystems. Using the asymmetric weight function, 97% of these queries have been answered with an average recall of 67% and precision of 66%, meaning more than half of the expected answers been returned.

Using symmetric weight function with $\lambda = 0.1$ precision improves to 71% at the cost of reducing both feedback and recall. The results suggest that the asymmetric weight function is more effective for high and midrange thresholds, and the symmetric function can be used with lower thresholds. This is a tradeoff between precision and recall in favor of precision.



(a) Precision (b) Recall

Figure 5.3: Prediction results for change propagation in WOS

The predictive power of domain-based coupling is affected by the confidence level and the given threshold to the weight functions. Figure 5.3 shows the comparison between recall and precision for confidence levels 0.1, 0.4 and 0.7. In order to find the impact of the different thresholds on the results, the queries have been answered using the asymmetric weight function with a range of answer thresholds from 0 to 1 exclusive (horizontal axis).

For all query thresholds, increasing the answer threshold reduces the recall and increases the precision. For queries with stronger confidence levels, the recall suddenly drops after some given threshold; however, for the queries created from low confidence levels this change is gradual.

Summary:*Using the asymmetric weight and a threshold of 0.1, we achieved up to 92% correct estimations of change propagation in WOS, and on average 67% correct estimation for all five subsystems.* Using the symmetric weight for estimating the change propagation on average improves the precision in cost of recall, making the symmetric weight a more preferable choice where a high level of accuracy is required.

5.5.5 Discussion

The results of this case study show how change propagation between software components can be predicted based on domain-based coupling. Also we demonstrated that such prediction could assist in avoiding bugs arising from imperfect software alteration.

We measured the correlation between domain-based and evolutionary coupling. The results suggest that there is a positive correlation between domain-based and evolutionary coupling for all five subsystems. The results in sections 5.5.3 and 5.5.4 show that the asymmetric weight function is more suitable for predicting change propagation at higher confidence levels, whereas in contrast the symmetric weight function is more suitable at lower confidence levels where it yields higher precision.

The effectiveness of this approach depends on the type of queries, and the behavioural characteristics of the system, and even though we used only domain information for change propagation analysis, the results are seemingly close to evolutionary coupling, suggesting that a domain-based approach is a plausible alternative.

5.6 Threats to Validity

This section discusses the main threats to the validity of our study.

Internal validity concerns factors that can influence our observations. We examined the evolutionary coupling at the coarse-grained level of individual source files. However, co-changes among files can be the result of modifying two unrelated functions. Hence there will be some false co-changes which leads to false increases in evolutionary coupling. The granularity level of UICs is the other factor which can affect our results. In this study a UIC is a screen that is accessible directly in the menu of the system. However, one can choose more fine-grained UICs such as individual panels and tabs in each screen. This might affect the domain-based coupling, and its correlation with evolutionary coupling. Also it might affect the result of the error prevention.

Construct validity concerns the relationship between theory and observations. The evolutionary coupling derived from the source code repository and the domain-based coupling derived from software functionalities. In the presented case study, we observed the correlation between these coupling metrics, but this observation does not provide any support to claim about any cause-effect relationship between the domain-based coupling and the co-changes which happen among files.

In our study, the examined system has been developed and maintained by a single company. The development culture and practices in the company might influence the way that individual developers manage and implement change propagation. This factor can affect the pattern with which developers commit the changes to the source code repository, and so derived evolutionary coupling between UICs. Hence, this factor can affect the correlation (Section 5.5.2) and error prevention (Section 5.5.3) results.

External validity concerns generalisation of our findings. In this study, we examined five subsystems of an enterprise application which is situated in the domain of facility management. These subsystems have been developed based on similar architecture and by the same company. This similarities limit the generalisation of our results to different domains, and other systems with different architectures.

5.7 Open Issues

The presented cases study in this chapter provides an insight into how domain-based coupling can be compared to evolutionary coupling. It also highlights the following future area of investigation:

- We envisage that the domain-based approach can be used to complement the history-based techniques and source code analysis methods, in a hybrid approach. Our case study shows that although overall evolutionary coupling and domain-based coupling are correlated, there are some exceptions. In such cases, each coupling measurement may reveal different aspects of a system's behavioural or architectural characteristics. In this work we did not investigate these cases qualitatively looking for complementarity, although we believe complementarity may well exist. A study evaluating complementarity is a clear candidate for future work.

- We applied asymmetric and symmetric weight functions to different tasks and compared the results, demonstrating that asymmetric weight has higher recall (providing more results) whereas symmetric weight has better precision. However, whether these functions can be used in conjunction to achieve even better efficiency is a subject for further work.
- We also propose to extend this work using information mined from bug reports and support records, leading to yet other forms of coupling with further potential benefits. Descriptive information recorded by users about application bugs, and required enhancements, may reveal complementary couplings between software components.
- We have shown that on average there is a positive correlation between predictions derived from domain-based and history-based analyses. However the variation in the quality of individual predictions is noticeable. A qualitative evaluation would yield insight into the underling causes of these variations.

The open issues which are discussed in this section are beyond the scope of this thesis. In Chapter 7, we will discuss the future direction of this research, and how these open questions form the possible road map of the future work.

5.8 Summary

In this chapter, we reported on a case study on a significant enterprise system, called BEIMS in order to address the third and fourth research questions of this thesis.

The third research question asks *how accurately can we predict change propagation using domain-based coupling?*, and the fourth research question asks *how does such a prediction compare with the well established co-change coupling derived from maintenance history?*

The results of the BEIMS case study show that the domain-based coupling can estimate up to 92% of change propagation derived from more than 12 years maintenance history of BEIMS. We applied both evolutionary coupling and domain-based coupling to detect missing components from change propagation. This exercise aimed to evaluate the efficiency of these metrics to avoid software bugs results of imperfect change propagation. The results shows the close performance of these methods with 46% to 48% recall and 37% to 54% precision for domain-based coupling and evolutionary coupling respectively.

Although domain-based coupling does not outperform evolutionary coupling, given its independence from software implementation and maintenance history, it can support maintenance of hybrid systems or legacy applications whose maintenance history is not easily traceable.

Chapter 6

Semi-Automated Approach

> Make everything as simple as possible, but not simpler.
>
> Albert Einstein

This chapter introduces a semi-automated approach to domain-based coupling analysis. The case studies in Chapter 3 demonstrated how domain-based coupling can be derived by observing the working software and manually recording the relations between domain variables and UICs. Although this approach can be implemented without any specific tool and support of developers, the required labour by the domain experts is a drawback to the manual approach. The semi-automated approach in this chapter addresses this issue by reducing the effort from the domain experts in collecting the domain information. This approach is based on the assumption that the system database stores the domain information, and one can exploit the database to derive the domain-level relationships. The semi-automated approach is applicable to information systems which use some form of relational database management system (RDBMS) to store domain information, and provide this information to the domain users via graphical interfaces (i.e., forms or screens).

Though this approach is based on automated steps, the domain expert's input is still required to verify the results. We examine the cost and benefit of the domain expert's input with a study on a subsystem of BEIMS[1].

[1] BEIMS is the enterprise system that we studied in Chapter 5 for change impact analysis.

This chapter is organised as follows: In Section 6.1, we explore the opportunities for automating the process of domain-based coupling analysis. Section 6.2 describes the semi-automated approach, and Section 6.3 presents the tool support. Section 6.4 shows the evaluation of the semi-automated approach. Section 6.5 discusses the future area of investigation, and finally, we summarise this chapter in Section 6.6.

6.1 Toward Automation

In Chapter 3, we described how the domain-based coupling between UICs can be modelled using the weighted graph. To construct such a model the domain experts will need to collect the following information: domain variables, UICs and their relationships. In a typical enterprise environment, domain elements can be derived from the following data sources:

- *Software artefacts*: Documents such as user manuals are valuable information sources about system features (domain functions), screens (UICs) and data fields (domain variables). This information can be derived using an automated tool. Although without formal documentation such a process cannot be fully automated, the domain experts can supervise the analysis process and improve the results.
- *Data schema*: In most information systems the domain variables are modelled as the table structure in a database. If the data schema of the database is accessible, then some or all of the domain variables can be mined from the schema.
- *Domain experts' knowledge*: Domain experts can provide valuable information about a system such as data fields (domain variables), relations between data fields and screens (UICs), and features (domain functions) of each screen. This information can be collected using questionnaires. However, filling in questionnaires is time consuming, instead, domain expert knowledge can correct or complete the data automatically derived from other sources like data schema. In Section 6.2, we will describe such a semi-automated process.
- *Working software*: The actual working software is a rich source of information about system elements. For most information systems a list of system screens (forms) is available through the software menu, or the site map for web applications. Also for these systems typically there is a security module which manages user access permissions to

system functions and screens. These resources can be used to derive a list of system UICs. Then an automated method can be used to observe user interaction with the system, record changes in UICs and discover related domain variables.

	V	C	$HAS.USE$
Software artefacts	✓	✓	✓
Data schema	✓		
Domain experts' knowledge	✓	✓	✓
Working software		✓	✓

Legend: V represents the domain variables, C represents the system UICs and $HAS.USE$ is the relationship between the UICs and the domain variables (Section 3.2).

Table 6.1: Sources of Domain Information

Table 6.1 summarises the available sources for domain information. As presented in this table, the system domain variables can be derived from software artefacts, data schema, or domain experts' knowledge. However, data schema might be the most cost effective data source. The list of system UICs and their relations with domain variables can be derived from software artefacts, domain experts' knowledge, and the working software.

For information systems the working software is a rich source of information about the UICs and their contents. In the next section, we describe a semi-automated process based on the analysis of the working software and the system data schema. This process reduces the effort required for domain experts to create the domain-based coupling graphs.

6.2 Semi-Automated Process

The semi-automated process refines the domain analysis process (Section 3.3) utilising the system data schema and analysis of the working software. The new approach derives the system domain variables from the data schema, and analyses the working software to derive the dependency matrix (Definition 4).

The domain experts supervise the various steps in the process and provide their input by verifying the output of automated steps. The aim of this verification is to correct false positive and false negative results arising from imperfectly automated activities. In Section 6.4, we will evaluate the effect of the effort of the domain experts on cost and quality of the process.

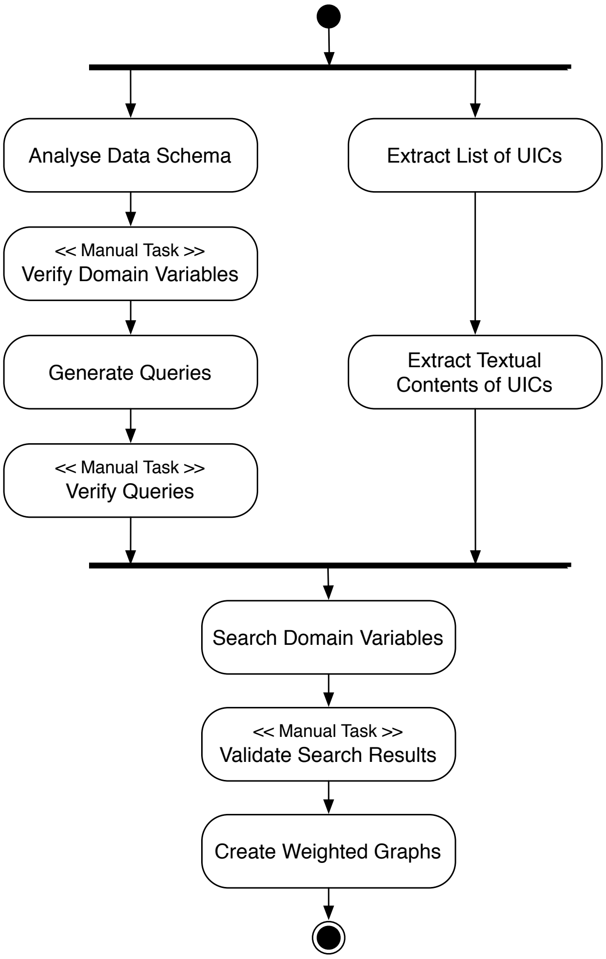


Figure 6.1: Semi-Automated Process

We assume that the information system incorporates some form of relational database (e.g Oracle or SQL server), and its data schema is accessible. The data schema can be automatically extracted from the database, or in some systems, it is published in a form of data dictionary. However, software artefacts tend to become outdated during the software evolution, thus, it is preferable to extract the schema from the live database.

Figure 6.1 illustrates the semi-automated process that includes the following activities:

- **Analyse Data Schema:** The aim of this activity is to discover domain variables based on the structure of the system data schema. An automated tool reads the system data schema and extracts a set of tuples of the form $\langle table, field\rangle$ where *table* is the name of a data table, and $field$ is the name of a data field in the *table*. For most relational database management systems (RDBMS) this can be achieved using an SQL script.

There may be redundancy between data fields for two primary reasons: (1) a foreign key in a table contains the same value as the primary key in the master table, (2) not all databases may be fully normalised, leading to duplicate fields in multiple tables.

To address these issues, the automated script ignores all fields which have foreign keys. Also it generates warnings when two data fields have the same name. These warnings will be reviewed by the domain expert (see below).

- **Verify Domain Variables:** The aim of this activity is to verify that all tuples present the domain variables. This is a manual process where domain experts audit the generated tuples and remove those which are not domain-related. For example, systems often record application settings in the database which contain screen dimensions, user preferences, fonts or colours. In addition, domain experts review the warnings for fields with the similar names (raised in the last activity) to avoid multiple tuples for the same domain variable.

  The tuples which are retained by the domain experts are interpreted as the system domain variables, and hereafter we refer to them simply as *domain variables*.

- **Generate Queries:** The aim of this activity is to create queries for searching the domain variables.

  We extend the derived domain variables to 4-tuples $\langle table, field, TableQuery, FieldQuery\rangle$ where $TableQuery$ and $FieldQuery$ are two sets of terms[2] which will be searched to identify if a domain variable is related to a UIC.

  We use the binary information retrieval method [128] in our work. The queries are generated via the following steps:

  - *Tokenisation:* To account for compound names we segment the *table* and $field$ string representations into *tokens*. Since meaningful names are usually made by compounding terms, and spaces are usually disallowed in identifiers, it is common to create compound names by delimiting multiple words with separators ('-','_','.',

[2] By *term* we mean an individual word which is considered atomic, i.e. indivisible and meaningful in the system's language locale. Some techniques described may work best for European languages.

etc.) or by using CamelCase[3]. We segment the *table* and $field$ from these delimiters into terms which can then be checked against the domain vocabulary[4]. The $TableQuery$ and $FieldQuery$ are two sets of terms which represent the tokens for the *table* and $field$ respectively.

- *Stemming and Case-folding:* For grammatical reasons different variants of a word might be used in UICs (e.g. code, codes, coding) which usually correspond to the same domain concept, but do not match using ordinary string comparisons. There are two strategies for resolving this issue; *stemming* and *lemmatization* [128].

  Stemming is a heuristic process which removes word endings to eliminate inflectional parts and derive a base form, and *lemmatization* is the method which achieves this by using a form of dictionary and morphological analysis of words. Stemming is simpler to implement than lemmatization, and given that the labels on UICs and the field names in the database often are very similar, stemming should provide sufficient accuracy.

  There are a number of well known stemming algorithms such as Lovins, Porter, and Paice [128]. As various stemming methods produce different outputs, for the optimum results, the same stemming method should be used on both the query and search areas (UICs contents). In a following activity (Extract Textual Content of UICs), we will discuss the tokenisation and stemming of UIC contents. We assume that most terms in the domain vocabulary are case-insensitive, thus, we convert all tokens to lowercase before searching the domain variables.

- **Verify Queries:** The aim of this activity is to verify the generated queries using domain knowledge. In this activity, domain experts review the generated queries and perform the following tasks:

  1. *Drop common tokens*: Extremely common tokens, or terms such as articles or prepositions do not help in finding domain variables, and might cause false positive results. There are two strategies to identify these tokens: (1) Sort the terms by collection frequency (total number of times a token is derived from a database),

[3] CamelCase is a practice of writing compound names in such a way that each element's initial letter is capitalised within the compound, the first letter is either lower or upper case, and the rest of the letters are lower case. E.g. HomeAddress, userName

[4] A domain vocabulary is a set of terms which are common in a domain and have domain-specific meaning.

and then remove the most frequent tokens. (2) Use a *stop list* which contains predefined terms. Such a list may be derived from the the system language dictionary.

2. *Add alternative queries*: The aim is to reduce false negative results. Domain experts review the tokens in the $TableQuery$ and $FieldQuery$ and add new tokens in two cases: First, not all possible terms (synonyms) for a domain variable can be derived from the *table* and $field$. Second, there are composite variables which are calculated from multiple domain variables (*e.g.,* key performance indicators (KPI), subtotals). The presence of these on a UIC indicates a relationship between the finer domain variables and the UIC. For example, a KPI called *ConditionIndex (C.I.)* is derived from *maintenance cost* and *asset value.* For every UIC that contains *C.I.*, we conclude that it is related to *maintenance cost* and *asset value.*

- **Extract list of UICs:** The aim of this activity is to derive a list of system UICs. For multi-user systems, it is common to have a central permission management module which controls user-access to different UICs. These modules typically store and read user permissions from a data source such as a database table, and an automated method can be used to read a UIC list from this table.

  Another data source for UICs is the site-map for many web applications and software menus for desktop clients. It is reasonable to assume that an automated tool will be able to read this information from the working software and generate a list of the system's UICs.

- **Extract Textual Content of UICs:** The aim of this activity is to extract the text from the UICs, and prepare the text to be searched for domain variables. This activity includes the following tasks:

  1. Extract the textual content of UICs. This can be done using optical character recognition techniques for desktop applications or an HTML parser for web-based applications. It is more likely that this task needs to be customised for individual applications.
  2. Tokenise the text derived from UICs using the same method which was used to search for the domain variables.
  3. Apply the same stemming method as used on the queries.
  4. Convert the tokens to lowercase.

The outcome is a set of tuples $\langle c, UicContent\rangle$ where c is a UIC and $UicContent$ is a set of tokens.

- **Search Domain Variables:** In this activity we use a boolean information retrieval function $SEARCH : A \times A \rightarrow \{false, true\}$ where A denotes the set of tokens. If Q and T are two sets of tokens representing a query and the search target respectively, then $SEARCH(Q, T)$ returns whether Q and T have common elements, that is:

$$SEARCH(Q, T) \equiv (Q \cap T \neq \emptyset)$$

For a domain variable $\langle table, field, TableQuery, FieldQuery\rangle$ to exist in a UIC $\langle c, T\rangle$, both $SEARCH(TableQuery, T)$ and $SEARCH(FieldQuery, T)$ should be *true*.

Example 1 *Imagine a report screen represented by the tuple* $\langle c, T\rangle$ *where* T *is the set of tokens from the screen contents. For the domain variable* Job-Number *represented by* $\langle$ *"Work_Orders", "No",* $TableQuery$ *,* $FieldQuery\rangle$ *where "Work_Orders" is the table name, "No" is the field name,* $TableQuery = \{$*"work", "job", "wo"*$\}$*, and* $FieldQuery = \{$*"no", "number", "id"*$\}$*, the search results shows that "job"*$\in T$ *and "number"*$\in T$ *therefore we can conclude that* Job Number *exists in the report screen.*

- **Validate Search Results:** In this activity, domain experts review the results, and resolve the following issues:
  - *False positives*: The automated search may return some false positive results where domain variables do not exist in the content of a UIC, but they have been returned because of similarity in labels and an imperfect search method. The domain experts compare the search outcomes with the UIC screen-shots, and remove these false positive variables from the set of domain variables for the UIC.
  - *False negatives*: Not all domain variables are recorded directly in the database; moreover, queries for domain variables may be imperfect. Domain experts review the search results for individual UICs and add any missing domain variables.

- **Create Weighted Graphs:** This is the last activity in the process where the domain information is combined to give the domain-based coupling graphs. This step can be achieved using an automated script that transforms the domain variables and their relationship with UICs to the weighted graphs based on Definition 6 and Definition 7.

In Section 6.4, we will evaluate the semi-automated approach, the required effort for the manual activities and their impact on the accuracy of the derived domain-based coupling.

6.3 Tool Support

In order to evaluate the semi-automated approach, we designed and developed a Java-based tool that supports collecting the domain information and performing the activities of the semi-automated process. The tool called Camros [33] is composed of a number of Java libraries that supports the semi-automated activities. We released the software package under the (BSD) license [48]. This allows for free use of the package by users and researchers. In addition, this license allows developers to incorporate their own services into the software and redistribute the applications.

The tool has three principle stakeholders: Subject Matter Expert, Requirements Analyst, Software Engineering Researcher.

It is important to make clear the difference between these stakeholders and the domain experts. While the label of domain expert can be applied to a broad range of expertise in a domain, the subject matter expert is more specifically experienced in the software system being analysed. In addition, the role of requirements analyst is not concerned with the domain but with the process of eliciting and assessing the impact of changes to the system. However, these stakeholder roles are often subsets of the responsibilities of a domain expert, and are commonly found as part of the following job titles associated with the software development life-cycle; Business Analyst, Expert User, Product Manager, System Analyst, and System Consultant. In contrast, the software engineering researcher role is not commonly associated with the domain expert, but with researchers in the specific areas of software evolution and maintenance.

6.4 Evaluation

The semi-automated approach aims to reduce the effort required by domain experts to predict change propagation based on the software domain information. In this section, we evaluate this approach by a case study where we use the semi-automated approach to predict the change propagation in the maintenance history of an enterprise system. In particular, we aim to answer these questions:

Q1: How much effort is required on the part of the domain experts to perform the manual activities? The answer to this question indicates the cost of the semi-automated process for predicting the change propagation.

Q2: To what extent will the recall and precision be affected if the domain-experts do not verify queries and search results? The answer of this question identifies the value of the domain experts' input to the process.

The system under analysis (SUA) is the Work Order System; one of the core subsystems of BEIMS[5] that has been introduced in Chapter 5. Two domain experts[6] performed the semi-automated process on SUA. Their process outcomes were cross checked to reduce the mistakes introduced through imperfect domain knowledge or human error.

6.4.1 Evaluation Setup

We perform the domain-based coupling analysis on SUA and use the derived domain-based coupling graph to predict the change propagation in the maintenance history of SUA.

In Section 5.4, we introduced $c.AFC(f, \lambda)$ which represents the relation between $c \in C$ and other components based on the function f and the given threshold λ.

We measure the likelihood of change propagation based on the frequency of the co-changes in the software maintenance history as presented in Section 5.2. Given a change request for a component $c \in C$, we query $q = (c, \lambda)$ other components which might be affected by the change with the *confidence* greater than λ. The expected answer to the query is the set of components derived from

$$E_q = c.AFC(conf, \lambda)$$

where $conf$ is the confidence function that returns the likelihood of change propagation based on the maintenance history and λ is the minimum acceptable value for $conf$.

[5] Building and Engineering Information Management System (BEIMS) is the enterprise systems that we used in Chapter 5 for change impact analysis.

[6] One of the domain experts had computer science background, and worked for Mercury Computer Systems, the software vendor for BEIMS.

We answer these queries using the asymmetric[7] domain-based coupling (Definition 6). We predict the set of affected components as

$$A_q = c.AFC(w_a, \gamma)$$

where γ is the threshold for the asymmetric domain-based coupling between the components. The queries are answered with range γ between $[0.1, .., 0.9]$.

The results are assessed using definitions of precision and recall as follows:

$$P_q = \frac{|A_q \cap E_q|}{|A_q|} \qquad R_q = \frac{|A_q \cap E_q|}{|E_q|}$$

We measure the mean value of the precision and recall for the SUA as

$$P_M = \frac{1}{n}\sum_{i=1}^{n} P_{q_i} \qquad R_M = \frac{1}{n}\sum_{i=1}^{n} R_{q_i}$$

which is derived from the individual queries for all components in the system.

6.4.2 Results: Required Effort by Domain Experts

In order to answer the first question (Q1) in this study *"How much effort is required on the part of the domain experts to perform the manual activities?"* we report on the results derived from the manual activities as part of the semi-automated process:

Verifying Domain Variables: The analysis output of the data schema shows 182 data tables that yields 1,790 tuples $\langle table, field \rangle$. Since the SUA is a subsystem of a large scale enterprise system, only a subset of these data tables contains information related to the SUA. The domain experts individually reviewed the analysis output and both agreed to remove 77 tables because of non related domain information. In addition, domain experts reviewed the warnings issued by the tool support for similar field names and removed duplicated fields. They left 701 tuples as related domain variables for the

[7] The asymmetric coupling provides better recall and lower precision, while symmetric coupling provides higher precision and lower recall (Chapter 5). In this study, we report only on the results from asymmetric domain-based coupling, aiming to improve recall.

SUA.

In order to reduce the time spent on this activity. The domain experts verified the domain variables only based on their knowledge about the system without consulting any system document or access the working software. This is a tradeoff between accuracy and productivity in favour of productivity, as the extended analysis time can discourage domain experts to use this method. For this study the domain experts completed this activity in less than an hour.

Verifying Queries: The domain experts reviewed the frequency list of tokens, and derived a set of 10 common tokens to drop from the queries. The removal of tokens from the queries was performed automatically using the tool support. In the next step they reviewed the derived queries and added alternative queries to 79 domain variables. The domain experts performed both these stages in less than forty minutes.

Verifying Search Results: The domain experts reviewed the generated search results for each UIC, added the missing domain variables (false negatives), and removed unrelated domain variables (false positive). On average for each UIC, 142±74 false positive domain variables were removed, and 2±3 false negatives domain variables were added. The time spent by domain experts on this activity was, on average, less than 10 minutes per UIC.

In summary, all the manual activities were performed in less than 3 hours.

6.4.3 Results: Precision and Recall

In this section, we answer the second question (Q2) of this study *"To what extent will the recall and precision be affected if the domain-experts do not verify queries and search results?"*

We evaluated the precision and recall of the change propagation prediction by the domain-based coupling using two different thresholds. The first threshold γ identifies the strength of the domain-based coupling. Table 6.2 shows that increasing γ provides more precise results at the expense of recall. This threshold is useful if someone wants to build a tool that provides limited and precise information about change propagation to the domain experts.

The second threshold λ identifies the minimum confidence for the change propagation. The

γ	0.1		0.2		0.3		0.4		0.5		0.7		0.9	
	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M	P_M	R_M
λ=0.1	0.59	0.82	0.73	0.8	0.85	0.61	0.92	0.57	0.99	0.39	1.00	0.15	1.00	0.04
λ=0.2	0.47	0.75	0.54	0.75	0.66	0.56	0.71	0.56	0.78	0.54	0.91	0.39	1.00	0.21
λ=0.3	0.37	0.71	0.4	0.71	0.42	0.57	0.46	0.57	0.52	0.57	0.7	0.43	1.00	0.29

Legend: γ=Threshold for domain-based coupling; λ=Threshold for co-change coupling

Table 6.2: Semi-Automated Process Results

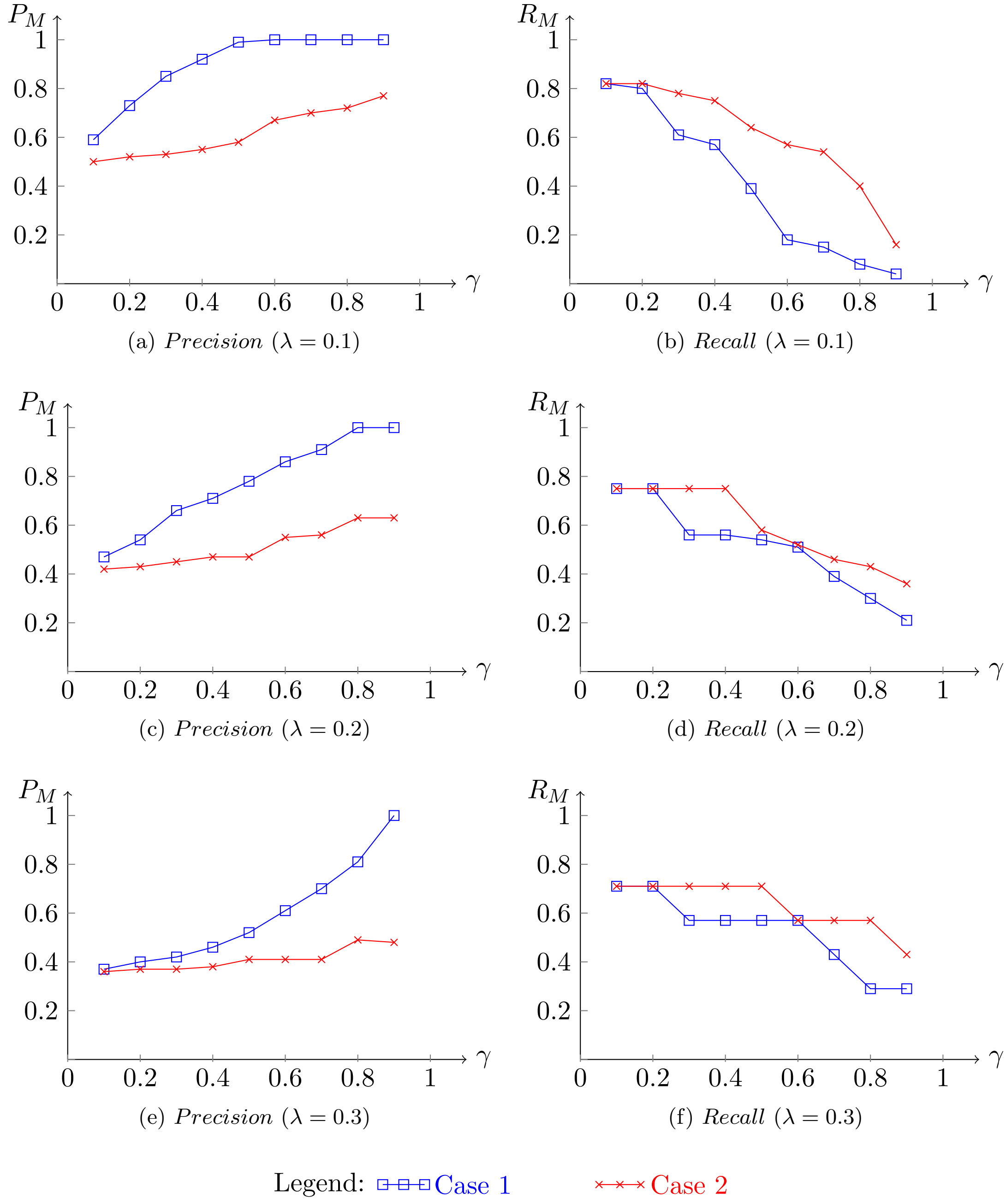
co-changes derived from the maintenance history represent the confidence of change propagation. Thus, the stronger λ the more significant is the likelihood of change propagation. This threshold can be used to focus the evaluation on components with strong evidence of change propagation in their maintenance history.

Table 6.2 shows the results of the change propagation analysis on the SUA using the semi-automated process. The domain-based coupling with the minimum threshold γ=0.1 returns 82% of components with likelihood of change propagation greater than 0.1, and the precision of the result is 0.59 implying that more than one in two returned components have are correct results. The precision will improve by increasing the threshold for the domain-based coupling. For γ=0.7 the precision increases to 0.99 while the recall decreases to 0.39.

Both precision and recall decrease for the queries which require stronger confidence levels. For λ=0.3, the domain-based coupling returns 37% of the expected results with the precision of 0.71 and we can increase the precision to 1.00 with γ=0.9 at the expense of reducing recall to 0.29.

Figure 6.2 shows the comparison between two different test cases. Case 1 is the result derived from the complete manual process that is the same as Table 6.2. In Case 2 we omit two activities in the process including Verify Queries and Validate Search Results. Hence, domain experts only provide input of verifying domain variables, and this significantly reduces their effort required to perform the analysis. The comparison between these two cases shows that the verification of queries and search results by domain experts has a positive impact on precision especially for stronger thresholds. For λ=0.1 and γ=0.1 the precision has been increased by almost 20% while for λ=0.3 and γ=0.9 the precision has been improved by 100% as a result of the domain expert input.

The other effect of the manual validation process is reducing the number of components in



(a) *Precision* ($\lambda = 0.1$)

(b) *Recall* ($\lambda = 0.1$)

(c) *Precision* ($\lambda = 0.2$)

(d) *Recall* ($\lambda = 0.2$)

(e) *Precision* ($\lambda = 0.3$)

(f) *Recall* ($\lambda = 0.3$)

Case 1: The complete process includes all manual and automated activities , Case 2: This case omits two manual activities: Verify Queries and Validate Search Results.

Figure 6.2: Impact of Manual Activities

the results. This has a negative impact on recall, but the comparison between the changes in the precision and recall for these cases shows that the improvement in precision is more significant than the reduction in recall.

6.4.4 Discussion

The evaluation result shows that the semi-automated process derived up to 82% of change propagation with the precision of 0.59. The precision can be improved to 1.00 at the expense of reduction in recall by increasing the threshold for the domain-based coupling. We evaluated our approach against three confidence levels which represent the strength of likelihood for change propagation. The result shows that an increase in the required confidence level for the change propagation reduces our precision and recall. At the strongest level with the minimum threshold for the domain-based coupling the recall is 0.29 and 100% precision.

The results derived from this case study suggest that the manual activities performed by the domain experts improve the precision of predicting change propagation. For the SUA, the domain experts performed these activities in less than 3 hours. The first two activities were each performed once only for the entire database, and the verified queries that they generated can be reused for the analysis of other subsystems of BEIMS as all subsystems of BEIMS use the central RDBMS, and the data schema used for this case study includes all data tables.

6.4.5 Threats to Validity

This section discusses the main threats to the validity of the case study:

Internal validity threats concern factors that may influence our observation. The time recorded for manual activities is affected by external variables including the time spent on conversation between domain experts, taking notes, and reading domain documents. In addition, the domain experts had previous experience with the domain-based coupling analysis, and so could be considered to be method experts.

The variation in domain knowledge and expertise of the domain experts who performed the study was not measured. This could affect the time taken for the analysis and the accuracy of the verification results.

The effect of prior learning on the accuracy and the effort spent on the later activities has also not been measured. This may have impacted the results because the domain experts, who performed all three manual activities, would have learned about the data schema and domain variables in the early activities.

External validity threats are concerned with the generalisation of the results. The system under study is a subsystem of an enterprise system. Given that the system is focused on a single domain (facility management), the case study results might not be applicable to other domains.

In addition, the system under study is a legacy system developed in a two-tier architecture. The location of the domain related source code can affect the change propagation results, thus more study is required to examine this method on different architecture types.

This is a limited case study with only two domain experts performing the process. We believe that challenges and patterns observed in this case study can assist in the application of the semi-automated process to other enterprise systems; however, we can not draw a strong conclusion from a limited case study.

6.5 Open Issues

This section presents the open issues and outstanding questions related to the presented process and the tool support.

In this chapter, we introduced four data sources for domain information. In extending the semi-automated approach, further studies can reveal new ways of driving domain concepts, and might assist in further reduction in required effort by domain experts. Specifically, we proposed to investigate the following questions:

- We derived the domain variables from data schema. What are the other sources that can be used to find domain variables, or confirm the ones which are derived from the data schema?
- We used the working software, and generated queries to match domain variables and

UICs. What are the other sources that can be used to investigate these relationships? For example, data mining from system design artefacts can be used to discover such relationships?

In the semi-automated method, we had a basic approach to enquiry domain experts by asking them directly about individual verifications. More work is required to study how domain experts' input in the process can be collected with their minimum effort. For example, for open source systems, there are social networks and public forums available which might be used to collect such information.

As it has been discussed in Section 6.2, there are two different methods for addressing the issue of words' various forms: stemming and lemmatization. In our approach we took the assumption that there is a similarity between UIC design and field names in the database so we chose the less expensive approach, stemming. A further study is required to evaluate if such an assumption is valid in most enterprise systems, or if there is any improvement in results by using a lemmatization method.

We evaluated the semi-automated approach against one individual system, and no strong conclusion can be drawn from a single case study. Extended studies are required to evaluate the efficiency of the proposed approach and the tool support against various software architectures and different domains.

In Chapter 8 we discuss the future direction of this research, and how the open issues presented in this section form the road map of the future work.

6.6 Summary

In this chapter, we answered the fifth research question by examining the various information sources which can be used for automating the domain-based coupling analysis. We introduced a semi-automated method which derives domain variables from system data schema, and reduces the effort for searching the domain variables in content of the system's UICs, and creating the domain-based coupling graphs.

We evaluated the proposed method with an enterprise case study, and the results show that the semi-automated method notably reduces the effort required to analyse and create the list of domain variables, analyse UICs and create the domain-based coupling graphs. However, the results suggest that the domain expert verification and input in two stages of this process are required to improve both precision and recall.

Chapter 7

Evaluation

> Science can only ascertain what is,
> but not what should be.
>
> Albert Einstein

Managing the cost and impact of software changes is a constant challenge in software maintenance. These days, it is common to find large scale enterprise systems that consist of subsystems developed in different languages; in addition, more than ever the software maintenance community is meeting the challenge of maintaining legacy systems with missing source code and outdated design documents.

In Chapter 2, we have identified three main approaches to change impact analysis, including document-based, code-based and history-based methods. The problem is that while domain experts have an important role in software evolution, they often find the existing change impact analysis methods difficult to use without the support of developers; as such, the existing methods limit the contribution of the domain experts in the process of evaluating and making decisions about software changes.

This thesis has addressed this problem by providing a methodology for change impact analysis that conforms to the following criteria:

- *Simplicity and usability*: This thesis aimed to provide a pragmatic methodology for change impact analysis that is simple and usable by non-technical domain experts.

- *Practicality*: This thesis aimed to provide a methodology that is applicable to typical enterprise systems with outdated design artefacts, heterogeneous source code and an inaccessible maintenance history.
- *Generality*: This thesis aimed to provide a methodology that works for general enterprise systems without any requirements to specific tools which are dependent on a particular programming language, software architecture, framework or implementation technology.
- *Efficiency*: This thesis aimed to enable domain experts to reliably estimate the change propagation in an acceptable timeframe with respect to the scale and complexity of the system.

A large group of software systems are constructed to model business domains (Section 2.3), and it is reasonable to expect that real-world dependencies should be reflected in their source code. In this thesis, we hypothesised that software dependencies can be predicted by exploiting domain information, and we investigated the following research questions:

- **RQ1:** What kind of model can we derive from domain experts' knowledge about relationships between software elements?
- **RQ2:** How accurately can we identify architectural dependencies using such a model?
- **RQ3:** How accurately can we predict change propagation using such a model?
- **RQ4:** How does such a prediction compare to the well-established co-change coupling derived from maintenance history?
- **RQ5:** What is the required effort and the cost of making the prediction?

In this chapter, we evaluate how the proposed methodology addresses the research questions and how it satisfies the described criteria. Moreover, we investigate the questions about how scalable is this methodology? How transferable are our results to different software types? What are the open issues and the threats to the validity to our findings?

The rest of this chapter is organised as follows: Section 7.1 shows the research contributions and findings. Section 7.2 discusses how the proposed research questions in this research are answered. Section 7.3 evaluates the proposed methodology against the criteria. Section 7.4 describes the transferability and scalability of the proposed method. Section 7.5 discusses the validity of the performed case studies in this thesis and we finally summarise this chapter in Section 7.7.

7.1 Research Contributions and Findings

This thesis has made the following contributions to the field of software evolution and maintenance:

- **Methodology:** This thesis introduced a novel methodology for change impact analysis based on only the software domain information. The benefits of the introduced methodology are: it is independent of the software implementation; therefore, it is applicable to software environments where source code analysis is not easily achievable, such as systems with heterogeneous source code. It is independent of the software maintenance history; therefore, it is applicable to systems with inaccessible maintenance logs, such as systems in their initial development stage. In addition, it is usable by non-technical domain experts who do not have access to software source code. The introduced methodology addresses RQ1.

- **Domain-based coupling:** In Chapter 3, this thesis introduced the domain-based coupling as a novel metric for measuring the semantic similarity between software domain level components. The domain-based coupling metric is the key element for predicting the change propagation based on software domain information and, as such, it is the core of this thesis.

- **ADempiere case study:** In Chapter 4, this thesis presented a case study of a large-scale open source ERP[1] system, called ADempiere. The result of this study demonstrates that the domain-based coupling can approximate the architectural dependencies among software components with an accuracy of more than 70%.

- **BEIMS case study:** In Chapter 5, this thesis presented on a case study of a significant-sized enterprise system, called BEIMS[2]. The result of the this study shows our proposed

[1] Enterprise resource planning

[2] Building and Engineering Information Management System

method can predict an average of more than 67% of the change propagation derived from more than 12 years maintenance history of BEIMS.

- **Semi-automated process:** In Chapter 6, this thesis provided a semi-automated process that reduces the efforts of the domain experts to measure the domain-based coupling and predict change propagation.

The next section describes how these contributions and findings answer the five research questions in this thesis.

7.2 Answers to Research Questions

This thesis investigates the five research questions as follows:

- **RQ 1: What kind of model can we derive from domain experts' knowledge about the relationships between the software elements?** In Chapter 3, we described the three main software domain-level elements: user interface components (UICs), domain functions and domain variables. These elements and their relationships are visible to the domain experts, and we showed how these experts can measure the level of the coupling between the UICs using the proposed domain-based coupling metric. As discussed in the background (Section 2.8), the proposed model in this thesis is not the only software domain model; however, the existing research in this area mainly aims to incorporate the domain information into the development environments that are mostly usable by programmers. In comparison the proposed model in this thesis is intended for use by domain experts.

- **RQ2: How accurately can we identify architectural dependencies using such a model?** In Chapter 3, we reported on a qualitative analysis of an enterprise web-based system where we described how domain-based coupling between the UICs corresponded to the dependencies in the source code. Furthermore, in the ADEMPIERE case study (Chapter 4), we analysed the dependencies across the multiple tiers of the software architecture, and demonstrated that domain-based coupling can be used to predict the probability of finding architectural dependencies between the UICs. In this study, the accuracy of such a prediction is, on average more than 70%.

- **RQ3: How accurately can we predict change propagation using such a model?** The results of the case study on the subsystems of BEIMS (Chapter 5) show that an average of 67% of the change propagation derived from the 12 years of the BEIMS maintenance history can be predicted by the domain-based coupling between the UICs. Although such a prediction is not sufficiently accurate to replace the existing code-based impact analysis methods, it can support the software maintainers where traditional code analysis tools do not work, such as hybrid systems or a legacy application with missing source code and design documents.

- **RQ4: How does such a prediction compare with the well-established co-change coupling derived from maintenance history?** In the BEIMS case study (Chapter 5) we compared the domain-based coupling with the well-established evolutionary coupling derived from the maintenance history; the result showed a positive correlation between these metrics. In addition, we applied both these metrics to predict software bugs resulting from imperfect maintenance activities. The results show the close performance of these methods, 46% to 48% recall and 37% to 54% precision for the domain-based coupling and evolutionary coupling, respectively.

- **RQ5: What is the required effort and cost of making the prediction?** In the presented case studies in Chapters 3 and 5, we described how the relationships between the domain elements can be derived from observing the working software and manually recording the related domain variables to the UICs. This approach requires only a functional knowledge of the system and the domain users can perform it without the support of developers . Therefore, the cost of the analysis is the time spent by the domain users. Although this is a usable method for small software packages, it is not scalable to large enterprise systems.

  To improve the scalability of this approach, we examined the data sources that can be used to automate the process of measuring the domain-based coupling (Chapter 6). We identified the system data schema as a source of information that can be used to derive the domain variables, and we developed an open source tool that reduced the effort by the domain experts by providing information about the potentially-related domain variables to the UICs. The evaluation results show that, when using this tool, the time spent by the domain expert is an average of eight minutes per UIC.

7.3 Evaluating the Methodology Against the Criteria

The domain-based approach to change impact analysis conforms to the pragmatic methodology criteria as follows:

7.3.1 Simplicity and usability

In Chapter 1, we described how domain experts can collect and transform domain information into a dependency matrix. From the dependency matrix, an automated tool can measure the domain-based coupling between the UICs and generate the weighted graphs. As demonstrated in the ADempiere and BEIMS case studies (Chapter 4 and Chapter 5), domain experts can identify highly-coupled components based on the domain-based coupling graphs and predict the dependencies and change propagations in the system.

Neither the domain-based coupling analysis nor the domain-based change impact analysis require any knowledge of software engineering or the software source code. However, the required labour by the domain experts can be a drawback in this approach. In Chapter 6, we addressed this issue by providing a semi-automated approach that reduces the efforts of the domain experts to measure the domain-based coupling.

In summary, our results show that domain-based change impact analysis is adequately simple and usable by domain experts without any need for technical knowledge of software engineering or access to the source code.

7.3.2 Practicality

The proposed approach in this thesis for change impact analysis is based on the information derived from the software behaviour at the domain-level. This information is independent of the software implementation and software source code. In the presented case studies (Chapter 4 and Chapter 5), the only documents that have been used for the domain analysis were the functional specifications of the system, user manuals and help documents, without a need to access any design artefacts or the technical specifications of the system.

In Chapter 5, we compared our method with evolutionary coupling derived from the maintenance history. However, our approach has no dependency on the maintenance history. Therefore, it is applicable to software systems with inaccessible version controls or brand new systems with few maintenance records. We will further discuss the scope of the applicability of this approach in Section 7.4.

In summary, our approach is practical for typical business applications and information systems with no requirement for source code analysis, access to design artefacts or data mining from the source code version control. Hence, it can be applied to hybrid systems (*e.g.,* C++ and Python) or recently-developed systems with little or no maintenance history. This approach is also applicable to legacy systems or systems where their design documents or source code are not available.

7.3.3 Generality

The introduced domain analysis method is based on software functionality, and independent of the software non-functional properties. Hence, the proposed method is not dependent on any specific programming language, architecture type (*e.g.,* SOA) or design specification (*e.g.,* UML diagrams).

The case studies in Chapter 3 demonstrate that the domain analysis can be performed using generic tools such as spreadsheets (*e.g.,* Excel) and a generic script for creating the weighted graphs from the collected data by the domain experts. The proposed tool support in Chapter 6 is introduced to facilitate the process; yet, it is not a requirement for the domain-based change impact analysis. The assumptions for the tool are access to the system data schema, access to the system UICs list, and the ability to take screen shots from the UICs. These assumptions are acceptable for the majority of business applications and enterprise systems.

The requirement for the introduced method is access to the system domain information, including the relationship between the UICs, domain functions and domain variables. This requirement limits the applicability of this method to systems where the majority of their functionality is related to storing and analysing data, and include a number of UICs. Hence, this approach is not applicable to systems where most of their functions are hidden to the end-users, such as an operating system.

In summary, the domain-based change impact analysis is applicable to general business applications and typical enterprise systems, regardless of their implementation.

7.3.4 Efficiency

In Chapter 5, we demonstrated the application of the domain-based coupling for predicting software bugs, and we compared its performance with the evolutionary coupling. The prediction result derived from domain-based coupling is close and, in some cases, even better than the evolutionary coupling.

The domain-based change impact analysis requires domain expert effort for domain-based coupling analysis and for creating the weighted graphs. The provided tool support in Chapter 6 reduces the effort by the domain expert; however, a once-off data preparation effort is still required by the domain experts. There is a linear correlation between the number of the UICs and the required effort by the domain experts, *i.e.,* the more screens and fields the enterprise system has, the more costly it is to create the weighted graphs.

However, after creating the weighted graphs, the process of the change impact analysis requires very little effort by the domain experts and, in most cases, it can be automated.

In summary, the domain-based change impact analysis is sufficiently efficient to predict the impact of the maintenance requirements for typical enterprise systems. This method requires neither expensive tools nor special technical knowledge; hence, it can be used by domain experts for typical enterprise systems to guide software maintenance and to facilitate planning for software enhancements.

7.4 Transferability and Scalability

In this section, we discuss the transferability and scalability of the domain-based change impact analysis. We describe its applicability to various software categories and different kinds of software changes.

First, *to what categories of software systems do these methods apply?* Pressman [152] organised computer software under seven categories: system, application, engineering/scientific, embedded, product-line, artificial intelligence and web applications. Our approach is applicable to subsets of the application software, product-line software and web applications that are data driven and provide their functionality through a number of user-interface components. Our approach is not applicable to software where its functionality is not visible to the domain users, such as the system software or embedded software. Also, domain-based change impact analysis may not be suitable where systems are not data driven or have few user-interface components, such as engineering/scientific or artificial intelligence software.

Second, *for what kinds of software changes can we use these methods?* Lientz and Swanson [118] classified software changes as perfective, adaptive, corrective and preventative. Preventative changes are typically initiated by programmers/developers or software engineers who are concerned with the non-functional properties of the system, such as the maintainability of the source code. Such changes might be difficult to map to domain functions; therefore, domain-based change impact analysis would not be a suitable approach for this kind of change. However, perfective, adaptive and corrective changes are typically performed in response to a request from the system users or in response to changes in the software environment. Such software changes, if related to changes in the software domain functions, can be assessed using domain-based change impact analysis.

Pressman [152] suggests four fundamental sources of software changes in relation to the business environment: (1) changes of business conditions, (2) changes in customer demands, (3) growth/downsizing of the business, and (4) budgetary or scheduling constraints. All of these changes might be defined as changes to the domain functions or user interface components or both; hence, their impact can be estimated using domain-based change impact analysis.

Finally, *how scalable are these methods?* Domain-based change impact analysis requires the domain experts' knowledge about user interface components. In Chapter 6, we described how domain information can be derived automatically from various sources, such as the actual working software, but we still rely on the domain experts' feedback in the final stage of the process. There is a linear relationship between the required effort for the analysis process by the domain experts and the number of user-interface components and domain variables.

7.5 Threats to Validity

We believe that the presented case studies in this thesis can be helpful for other researchers and practitioners. They demonstrate how the domain-based coupling can be derived from information systems, and how it can be used to predict the architectural dependencies and the impact of the software changes. Nevertheless, our results should not be generalised too hastily without first considering the following possible threats to the validity.

Internal validity concerns uncontrolled factors that can be responsible for the results. In ADempiere and BEIMS case studies (Chapters 4 and 5), we identified the following threats to the internal validity:

- The domain information is collected by the domain experts and human error is a factor that can affect the results. To minimise the risk of human error, we extracted the relationship between the domain variables and the UICs from user manuals and help documents. In ADempiere , this information is stored in the database. We used only manual inputs from the domain experts to confirm this information and kept the manual additions and alterations to a minimum.

- One other factor that could affect the results is the granularity of the UICs. In both ADempiere and BEIMS studies, we chose windows as the UICs. Each window contains multiple tabs and each tab provides one or more functions. Different results could be achieved if the evaluation were performed on the fine-grained tabs or the coarse-grained modules.

- In the BEIMS case study, we derived evolutionary coupling from the co-changes at the file level. However, a developer can apply unrelated changes to two files in a close time frame. For example, a developer can work on two unrelated bugs in the same time frame and send the changes to the repository as part of the same transaction. Such co-changes can lead to false positive evolutionary coupling and reduce the recall of the prediction results by the domain-based coupling.

- BEIMS is a proprietary software system developed by a single company. The company standard practices and development cultures might influence both the software architecture and the maintenance activities, including the way the developers fix bugs and enhance the system. To reduce this impact, we examined more than 12 years of the maintenance history of the system. The longevity of this maintenance history reduces the influence of the individual developers by including more developers and different software versions.

External validity concerns the generalisation of our findings. In ADempiere and BEIMS case studies, we evaluated our approach against the large-scale enterprise systems. Although the maturity of the data about these systems provided an insight into the relationships among the architectural dependencies, change propagation and the domain-based coupling, the following limitations in our studies should be considered before generalising our findings:

- ADempiere is developed in JAVA and based on the multi-tier architecture. The architecture of this system is designed to enhance the maintainability and extendibility of the system; it reduces the code coupling and code clones, as such ADempiere manifests the state of the art open source enterprise systems. However, one might get a different result for a system with much code coupling or a legacy system with a flat architecture.
- In BEIMS case study, we examined the five subsystems that operate in the domain of facility management. Although these systems have separate functionalities, they have been developed based on a similar architecture and by the same company. This similarities limit the generalisation of our results to different domains and other systems with different architectures.

Construct validity concerns the relationship between the theory and the observations. In BEIMS case study, we reported on a case study that compared the domain-based coupling with the evolutionary coupling. In these studies, we demonstrated the correlation between the two coupling metrics; domain-based coupling from the system behaviour and evolutionary coupling from the co-changes in the source code repository. However, our observation did not provide support to claim a cause-and-effect relationship between these coupling metrics. The correlation only suggests that one coupling metric can be used as a proxy for the other.

7.6 Roadmap for Future Work

In this section, we summarise the open issues in our work and describe a roadmap for the future areas of investigation. We have identified the following open issues in the prior chapters of this thesis:

- In Chapter 3, we reported on two issues concerning the density of the domain-based coupling graph: first, the graph can be too complex and not readable for the large-scale systems with many components. Second, although applying a threshold to edges' weight improves the readability of the graph, finding an optimum threshold value is a challenge. We proposed a solution to these issues in Chapter 4, where we used the Expectation Maximisation (EM) clustering technique to reduce the number of edges in the graph. However, we have evaluated only this clustering method with a single case study, and more studies are required for examining the efficiency of the clustering method for identifying highly-coupled components.

  In addition, we have identified the following limitations of the performed case studies in Chapter 3: the performed case studies reported only the overall architectural dependencies and did not distinguish the relationships between the dependencies in the source code and the other application layers. Moreover, both these studies report on web-based systems; thus, the observations are difficult to relate to other application types. We addressed these issues in Chapter 4 where we compared the domain-based coupling with the dependencies in the source code and database layers of a heterogeneous open source system. In addition, the case studies in both Chapter 4 and Chapter 5 include the analysis of the desktop applications that typify the majority of the enterprise systems and legacy applications.

- In Chapter 4, we reported two open issues with the ADEMPIERE case study: first, the observations provide only limited information about the impact of the granularity of the UICs on the results. In this study, we reported only on the course-grained UICs; however, one can get different results for the analysis of the dependencies between the fine-grained UICs. Second, the evaluation does not include the impact of the domain characteristics (*e.g.,* complexity) on the prediction accuracy. For example, extended case studies might reveal the prediction is more accurate for systems composed of more independent components.

- In Chapter 5, we have envisaged that the domain-based coupling and the evolutionary coupling can be combined to provide a more accurate prediction about the change propagation. A similar hybrid approach can be implemented based on both the symmetric and asymmetric domain-based coupling. In addition, we have examined only the recorded change propagation in the source code version control, but other sources such as bug reports and support records can be explored for identifying the change propagations. Finally, we have demonstrated that there is a positive correlation between the domain-based coupling and the evolutionary coupling. However, the variation in the quality of the individual predictions is noticeable. Hence, a qualitative evaluation might yield insights into the underling causes of these variations.

- In Chapter 6, we have identified these future areas of investigations: the proposed tool support derives the domain variables from the system data schema, but one can investigate other sources that can be used to find domain variables or confirm those that are derived from the data schema. In addition, the proposed tool uses the working software and optical character recognition to identify the relationships between the domain variables and the UICs. One can extend this method to other dynamic analysis methods and identify other sources that can be used to investigate these relationships.

- The case studies in this thesis have involved only a few domain users because these were the only ones who knew the systems and had time to evaluate our research. However, future work could involve other domains and provide a more thorough evaluation of the challenges in domain analysis faced by non-technical domain users.

Given the described open issues, contributions and findings of this thesis, we envisage the following road map for the future work:

- *Extended case studies*: we envisage that further case studies can extend the findings of this thesis by:
  - providing information about the impact of the granularity of the UICs on the prediction of the change propagation and the dependencies between the UICs
  - identifying the properties of the UICs that can affect the prediction results (*e.g.,* size and complexity)

  - assessing the potential impact of the domain characteristics (*e.g.,* complexity) on the prediction result
  - qualitatively comparing the change propagation and the domain-based coupling to identify the other factors that can affect the prediction result
  - increasing the information about the usability of the domain-based coupling by non-technical domain users and introducing new methods for collecting domain information.

- *Exploring other sources of domain information*: we envisage that other aspects of software systems can be exploited to:
  - identify the domain variables, *e.g.,* domain ontology and system documents
  - identify the relationships between the domain variables and the UICs, *e.g.,* dynamic analysis.

- *Exploring hybrid approaches*: the observations in ADEMPIERE and BEIMS case studies suggest that the following hybrid approaches might lead to a better prediction of the software dependencies and change propagations:
  - combining the domain-based coupling and the evolutionary coupling derived from the software maintenance history
  - combining the domain-based coupling and a coupling metric derived from the source code analysis
  - combining the symmetric and asymmetric domain-based couplings.

We envisage that the domain analysis can complement many of the existing code analyses and reverse engineering methods and we hope this work will encourage researchers and practitioners to explore other applications of the domain-based coupling.

7.7 Summary

In summary, this thesis has described how to model the domain-level coupling between the software user-interface components and how such a model can be used to discover the architectural dependencies and change propagation.

In this chapter, we discussed how the derived model from domain information and the proposed methodology conforms to the criteria for a pragmatic change impact analysis methodology. In addition, we described the contributions of this thesis, the outcomes and research findings of the presented case studies and the threats to the validity of our findings.

Chapter 8

Conclusion

In this thesis, we introduced a novel methodology for change impact analysis based on the domain information. The introduced approach is independent of the software implementation, inexpensive to implement and is usable by domain experts without the requirement to access and analyse the source code. As part of this methodology, we introduced the domain-based coupling as a novel metric for measuring the semantic similarity between the software domain level components. This metric enables domain experts to assess the likelihood of the dependencies and the change propagation between the software components. This approach is based on the assumption that domain-level relationships are reflected in the software source code, and one can predict software dependencies and change propagation by exploiting the software domain-level information.

This thesis evaluated the proposed methodology with two large-scale enterprise case studies. In the first case study, we compared the domain-based coupling with the architectural dependencies in a large-scale open-source enterprise system, called ADempiere. The results show that the domain-based coupling can approximate the architectural dependencies among the software components with an accuracy of more than 70%. In the second case study, we compared the domain-based coupling with more than 12 years of maintenance history for a significant-sized proprietary enterprise system, called BEIMS. The analysis of the maintenance history of the five core subsystems of BEIMS shows that the domain-based coupling predicts more than 67% of the change propagations.

The results of these studies support our hypothesis that the domain-based coupling can approximate the software dependencies and change propagation. Although the accuracy of such predictions is not sufficiently strong to replace the existing code analysis and reverse engineering techniques, it can be used where conventional code analysis methods are not easily applicable. For example, the domain-based coupling can predict dependencies and change propagation in hybrid systems with heterogeneous source code or legacy systems with missing source code and outdated design documents. In addition, the proposed method can be used by domain experts to evaluate the impact of prospective software changes without the support of the developers. Such an evaluation by domain experts can reduce the cost of managing software maintenance and assist software development teams to more efficiently plan for minor or major software changes.

We envisage that the contributions and findings of this thesis can be extended in the following areas[1]: (1) extended case studies that investigate the functional and non-functional properties of the system that affect the accuracy of the prediction results, (2) exploring other sources of domain information, and (3) exploring the hybrid approaches that combine the domain-based coupling with the existing coupling metrics.

We hope this work will encourage researchers to explore further applications of domain analysis in software maintenance and software engineering, and entice practitioners to take advantage of domain-based analysis as part of their software development practices and discover software properties that are difficult to derive from the source code.

[1] The roadmap for future work is described in detail in Section 7.6.

Glossary

ADempiere An Enterprise Resource Planning or ERP software package released under an open source software license. 130

BEIMS Building and Engineering Information Management System, an enterprise system designed and developed by Mercury Computer Systems (Australia) Pty Ltd. 87

Change Propagation The phenomenon whereby a change to a software component lead to changes to other components. 23

Domain Analysis The activity of identifying the objects and operations of similar systems in a particular domain. 33

Domain Expert The expert who provides information about the domain model of a system, and supports domain analysis. 34

Domain Function Proactive or reactive domain-level behaviour of the system which includes at least one domain variable as an input or output. 38

Domain Model The definition of the functions, objects, data, and relationships in a domain. 33

Domain Variable A variable unit of data with a clear identity at the domain level. 38

Domain-Based Coupling A coupling measurement between components based on software domain-level behaviour. 44

Expectation Maximization A clustering technique that groups a given set of objects so that similar objects are grouped together and dissimilar objects are kept apart. 75

Impact Analysis The activity of identifying what to modify to accomplish a change, or of identifying the potential change propagation. 24

Logical Coupling The semantic relationships between classes based on source code version history. 31

MSR Mining Software Repository. 31

Precision The percentage of a returned answer which was expected. 76

RDBMS Relational Database Management System abbreviation. 103

Recall The percentage of an expected answer which was returned. 76

SLC Software Life Cycle. 12, 14, 15, 19–23

Software Evolution The dynamic behaviour of software systems as they are maintained and enhanced over time. 14

Software Maintenance The correction of errors and modifications needed to allow an existing system to perform new tasks and to perform the old ones under the new conditions. 14

SUA System Under Analysis abbreviation. 40

UIC User Interface Components abbreviation. 38

Bibliography

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

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

[3] R.S. Arnold and S.A. Bohner. Impact analysis - towards a framework for comparison. In *Conference on Software Maintenance (CSM)*, pages 292–301, September 1993.

[4] A. Aryani, I.D. Peake, M. Hamilton, H. Schmidt, and M. Winikoff. Change propagation analysis using domain information. In *20th Australian Software Engineering Conference (ASWEC)*, pages 34–43, Australia, April 2009. IEEE.

[5] A. Aryani, F. Perin, M. Lungu, A.N. Mahmood, and O. Nierstrasz. Can we predict dependencies using domain information? In *18th Working Conference on Reverse Engineering (WCRE)*, Limerick, Ireland, October 2011. IEEE.

[6] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval.* Addison-Wesley Reading, MA, 1999.

[7] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering*, pages 86 –95, July 1995.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, pages 368–377. IEEE, 1998.

[9] K. Beck. *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

[10] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

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

[12] K. H. Bennett and V.T. Rajlich. Software maintenance and evolution: a roadmap. In *International Conference on the Future of Software Engineering*, pages 73–87, 2000.

[13] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *International conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[14] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.

[15] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings. International Conference on Software Maintenance (ICSM)*, pages 44 – 53, September 2003.

[16] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105 – 178, 2004.

[17] D. Binkley, M. Harman, and J. Krinke. Empirical study of optimization techniques for massive slicing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):3, 2007.

[18] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(4):263–279, 2001.

[19] S. Black. *Software evolution and feedback*, chapter The role of ripple effect in software evolution, pages 249–268. John Wiley & Sons, 2006.

[20] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *30th International Conference on Software Engineering*, pages 511 –520. IEEE, May 2008.

[21] S.A. Bohner. Extending software change impact analysis into cots components. In *Software Engineering Workshop, 27th Annual NASA Goddard/IEEE*, pages 175–182, 2002.

[22] S.A. Bohner and R.S. Arnold. An introduction to software change impact analysis. In Jon Butler and Lisa O'Conner, editors, *Software Change Impact Analysis*, pages 1–26. IEEE Computer Society Press, 1996.

[23] S.A. Bohner and R.S. Arnold. *Software change impact analysis*. Wiley-IEEE Computer Society Pr, 1996.

[24] G. Booch. *Object-oriented design*, volume I. ACM, New York, NY, USA, March 1982.

[25] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. In *20th IEEE International Conference on Software Maintenance*, pages 453–457. IEEE, 2004.

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

[27] L.C. Briand, Y. Labiche, L. O'Sullivan, and M.M. Sówka. Automated impact analysis of uml models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[28] L.C. Briand, Y. Labiche, L. OSullivan, and M.M. Sówka. Automated impact analysis of uml models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[29] L.C. Briand, J. Wust, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 475 –482, 1999.

[30] R. Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[31] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September 2005.

[32] M. Burch, S. Diehl, and P. Weißgerber. Visual data mining in software archives. In *Proceedings of the 2005 ACM symposium on software visualization*, SoftVis '05, pages 37–46. ACM, 2005.

[33] Camros. http://sourceforge.net/p/camros/, November 2012.

[34] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Mechanical Design of software maintenance and evolution*, 13:3–30, 2001.

[35] M.A. Chaumun, H. Kabaili, R.K. Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. In *Third European Conference on Software Maintenance and Reengineering*, pages 155–174, 1999.

[36] A. Chen, E. Chou, J. Wong, A.Y. Yao, Qing Zhang, Shao Zhang, and A. Michail. Cvssearch: searching through source code using cvs comments. In *IEEE International Conference on Software Maintenance*, pages 364 –373, 2001.

[37] K. Chen and V.T. Rajlich. Ripples: tool for change in legacy software. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 230–239, 2001.

[38] X. Chen. Extraction and visualization of traceability relationships between documents and source code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 505–510, New York, NY, USA, 2010. ACM.

[39] S.R. Chidamber and C.F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 197–211, November 1991.

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

[41] S. Cook, R. Harrison, M.M. Lehman, and P. Wernick. Evolution in software systems: foundations of the spe classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):1–35, 2006.

[42] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Transactions on Software Engineering*, 35(5):684 –702, September 2009.

[43] D. Cubranic and G.C. Murphy. Hipikat: recommending pertinent software development artifacts. In *25th International Conference on Software Engineering*, pages 408–418, May 2003.

[44] H.K. Dam and A. Ghose. Automated change impact analysis for agent systems. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 33–42, September 2011.

[45] K.H Dam and M. Winikoff. Generation of repair plans for change propagation. In *Eighth International Workshop on Agent Oriented Software Engineering (AOSE)*, pages 30–44, 2007.

[46] K.H Dam and M. Winikoff. Supporting change propagation in uml models. In *International Conference on Software Maintenance (ICSM)*. IEEE, September 2010.

[47] K.H Dam, M. Winikoff, and L. Padgham. An agent-oriented approach to change propagation in software evolution. In *Australian Software Engineering Conference (ASWEC)*, pages 309–318, 2006.

[48] BSD License Definition. http://www.linfo.org/bsd_license.html, November 2010.

[49] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[50] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance(ICSM)*, pages 109–118. IEEE, 1999.

[51] M. El-Ramly and E. Stroulia. Mining software usage data. In *Proceedings of 1st International Workshop on Mining Software Repositories (MSR'04)*, pages 64–8, 2004.

[52] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[53] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *IEEE International Conference on Software Maintenance (ICSM)*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[54] T.M.J Fruchterman and E.M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, November 1991.

[55] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *30th International Conference on Software Engineering(ICSE)*, pages 321–330. IEEE, 2008.

[56] H. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, January 2009.

[57] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance*, pages 190–198. IEEE Computer Society Washington, DC, USA, 1998.

[58] H. Gall, M. Jazayeri, R.R. Klosch, and G. Trausmuth. Software evolution observations based on product release history. In *International Conference on Software Maintenance (ICSM)*, pages 160–166, September 1997.

[59] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: the use of color and third dimension. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 99–108, 1999.

[60] K. Gallagher and D. Binkley. Program slicing. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 58 –67, October 2008.

[61] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[62] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751 –761, August 1991.

[63] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*, December 2009.

[64] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[65] M.M. Geipel and F. Schweitzer. Software change dynamics: evidence from 35 java projects. In *7th joint meeting of the European software engineering conference and the*

ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pages 269–272, USA, 2009. ACM.

[66] Daniel M. German, Ahmed Hassan, and Gregorio Robles. Change impact graphs: Determining the impact of prior code changes. *Information and Software Technology*, 51(10):1394–1408, 2009.

[67] D.M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[68] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *IEEE International Conference on Software Maintenance*, pages 1–10, September 2010.

[69] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*, chapter 2, page 25. Prentice Hall PTR, 2002.

[70] T. Gırba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[71] M.W. Godfrey and D.M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 129–138, October 2008.

[72] M.W. Godfrey and Qiang Tu. Evolution in open source software: a case study. In *International Conference on Software Maintenance*, pages 131–142. IEEE, 2000.

[73] A. Goknil, I. Kurtev, and K.V.D Berg. Change impact analysis based on formalization of trace relations for requirements. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 59–75, June 2008.

[74] O.C.Z. Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994.

[75] Graphviz. Graph visualization software. http://www.graphviz.org, January 2011.

[76] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *8th International Workshop on Software Technology and Engineering Practice*, pages 172–182, London, UK, 1997.

[77] A.E. Hassan and R.C. Holt. Predicting change propagation in software systems. In *International Conference on Software Maintenance*, pages 284–293, 2004.

[78] A.E. Hassan and R.C. Holt. Using development history sticky notes to understand software architecture. In *12th IEEE International Workshop on Program Comprehension*, pages 183 – 192, June 2004.

[79] A.E. Hassan and R.C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11:335–367, 2006.

[80] J. Hassine, J. Rilling, J. Hewitt, and R. Dssouli. Change impact analysis for requirement evolution using use case maps. In *International Workshop on Principles of Software Evolution*, pages 81–90, September 2005.

[81] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *International Working Conference on Mining Software Repositories*, pages 141–150. IEEE, 2009.

[82] L. Hattori, M. Lanza, and R. Robbes. Refining code ownership with synchronous changes. *Empirical Software Engineering (EMSE)*, 17(4-5):467–499, 2012.

[83] L. Huang and Y.-T. Song. Dynamic impact analysis using execution profile tracing. In *Fourth International Conference on Software Engineering Research, Management and Applications*, pages 237–244, August 2006.

[84] L. Huang and Y.-T. Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *5th ACIS International Conference on Software Engineering Research, Management Applications*, pages 374 –384, August 2007.

[85] L. Huang and Y.-T. Song. A dynamic impact analysis approach for object-oriented programs. In *Advanced Software Engineering and Its Applications*, pages 217 –220, December 2008.

[86] IEEE. Standard for software maintenance, December 1992.

[87] ISO. *ISO/IEC 9126 - Software engineering - Product Quality - Part 1: Quality model*, 2001.

[88] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE)*, pages 96–105. IEEE Computer Society, 2007.

[89] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

[90] J.H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, 1994.

[91] C. Jones. *Applied Software Measurements: Global Analysis of Productivity and Quality*, chapter 4, page 414. McGraw-Hill, 2008.

[92] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[93] H. Kagdi, M.L. Collard, and J.I. Maletic. 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 (JSME)*, 19(2):77–131, March 2007.

[94] H. Kagdi, M. Gethers, D. Poshyvanyk, and M.L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 119 –128, October 2010.

[95] H. Kagdi, J.I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *International Conference on Program Comprehension*, volume 0, pages 145–154, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[96] H. Kagdi, S. Yusuf, and J.I. Maletic. Mining sequences of changed-files from version histories. In *International Workshop on Mining Software Repositories*, pages 47–53, New York, NY, USA, 2006. ACM.

[97] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[98] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[99] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report cmu/sei-90-tr-021, Software Engineering Institute, Carnegie Mellon University, 1990.

[100] C.F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July 1999.

[101] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97Object-Oriented Programming*, pages 220–242, 1997.

[102] B.A. Kitchenham, G.H. Travassos, A. von Mayrhauser, F. Niessink, N.F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance: Research and Practice*, 11(6):365–389, 1999.

[103] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis*, pages 40–56, 2001.

[104] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[105] Thomas Kuhn. *The Structure of Scientific Revolutions.* University of Chicago Press, 2nd edn edition, 1970.

[106] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *International Conference on Software Maintenance (ICSM)*, pages 202 –211, September 1994.

[107] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 430–441. IEEE, 2003.

[108] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *25th International Conference on Software Engineering*, pages 308–318. IEEE, 2003.

[109] M.L. Lee. *Change impact analysis of object-oriented software.* PhD thesis, George Mason University, 1998.

[110] Wen-Tin Lee, Whan-Yo Deng, Jonathan Lee, and Shin-Jie Lee. Change impact analysis with a goal-driven traceability-based approach. *International Journal of Intelligent Systems*, 25(8):878–908, 2010.

[111] M. M. Lehman and Juan F. Ramil. *Software evolution and feedback*, chapter 1, pages 7–40. John Wiley & Sons, 2006.

[112] Meir M. Lehman. Laws of software evolution revisited. *Software process technology*, 1149:108–124, 1996.

[113] Meir M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[114] M.M. Lehman. Programs life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[115] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution-the nineties view. In *Fourth International Software Metrics Symposium*, pages 20–32, November 1997.

[116] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *Software, IEEE*, 3(3):41–49, May 1986.

[117] B.P. Lientz and E.B. Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[118] B.P. Lientz and E.B. Swanson. *Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley Publishing Company, Reading, MA, USA, 1980.

[119] B.P. Lientz, E.B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21:466–471, June 1978.

[120] M. Lindvall and K. Sandahl. Practical implications of traceability. *SoftwarePractice & Experience*, 26(10):1161–1180, 1996.

[121] M. Lindvall and K. Sandahl. Traceability aspects of impact analysis in object-oriented systems. *Journal of Software Maintenance: Research and Practice*, 10(1):37–57, 1998.

[122] W.Q. Liu, S. Easterbrook, and J. Mylopoulos. Rule-based detection of inconsistency in uml models. In *Workshop on Consistency Problems in UML-Based Software Development*, pages 106–123, Dresden, Germany, 2002.

[123] A.D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):13, 2007.

[124] M. Lungu and M. Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press.

[125] Luqi. A graph model for software evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, August 1990.

[126] N.H. Madhavji. The prism model of changes. In *13th International Conference on Software Engineering*, pages 166 –177, May 1991.

[127] A.N. Mahmood, C. Leckie, and P. Udaya. An efficient clustering scheme to exploit hierarchical data in network traffic analysis. *IEEE Transactions on Knowledge and Data Engineering*, 20:752–767, 2008.

[128] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[129] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *16th Annual International Conference on Automated Software Engineering*, pages 107–114. IEEE, 2001.

[130] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135. IEEE Computer Society, 2003.

[131] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *International Workshop on Program Comprehension*, pages 33 – 42, May 2005.

[132] Margaret Masterman. The nature of a paradigm. In *Proceedings of the International colloqium in the philosophy of science: Criticism and the growth of knowledge*, pages 59–89. Cambridge University Press, 1970.

[133] A.V Mayrhauser and A. MarieVans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44 –55, August 1995.

[134] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution*, pages 13–22. IEEE, September 2005.

[135] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[136] A. Michail. Browsing and searching source code of applications written using a gui framework. In *International Conference on Software Engineering*, pages 327 –337, May 2002.

[137] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 177–188, 2007.

[138] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 120 –130, 2000.

[139] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPCí02)*, pages 219–228, 2002.

[140] J. M. Neighbors. *Software Construction Using Components*. PhD thesis, University of California, Irvine, 1980.

[141] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *25th International Conference on Software Engineering*, pages 455 – 464. IEEE, May 2003.

[142] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press.

[143] S.C. North. Drawing graphs with neato. http://www.graphviz.org/pdf/neatoguide.pdf, April 2004.

[144] A. Orso, T. Apiwattanapong, and M.J. Harrold. Leveraging field data for impact analysis and regression testing. *ACM SIGSOFT Software Engineering Notes*, 28(5):128–137, 2003.

[145] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M.J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 491–500. IEEE Computer Society, 2004.

[146] D.L. Parnas. Software aging. In *International Conference on Software Engineering*, pages 279–287, May 1994.

[147] I. Pashov and M. Riebisch. Using feature modeling for program comprehension and software architecture recovery. In *International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 406 – 417, May 2004.

[148] D.E. Perry. *Software evolution and feedback*, chapter A Nontraditional View of the Dimensions of Software Evolution, pages 41–51. John Wiley & Sons, 2006.

[149] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *17th International Conference on Program Comprehension (ICPC)*, pages 10–19. IEEE, May 2009.

[150] M. Petrenko, V. Rajlich, and R. Vanciu. Partial domain comprehension in software evolution and maintenance. In *International Conference on Program Comprehension*, pages 13 –22, June 2008.

[151] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, February 2009.

[152] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, seventh edition edition, 2010.

[153] J.-P Queille, J.-F. Voidrot, N. Wilde, and M. Munro. The impact analysis task in software maintenance: a model and a case study. In *International Conference on Software Maintenance*, pages 234 –242, September 1994.

[154] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *International Workshop on Program Comprehension*, pages 271 – 278, 2002.

[155] V.T. Rajlich. A model for change propagation based on graph rewriting. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 84–91, 1997.

[156] V.T. Rajlich. Modeling software evolution by evolving interoperation graphs. *Annals of Software Engineering*, 9:235–248, 2000.

[157] V.T. Rajlich. Changing the paradigm of software engineering. *Communications of the ACM*, 49(8):67–70, 2006.

[158] V.T. Rajlich and K.H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, July 2000.

[159] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don't). In *Working Conference on Reverse Engineering*, pages 83 –92, October 2006.

[160] D. Ratiu and F. Deissenboeck. Programs are knowledge bases. In *14th IEEE International Conference on Program Comprehension*, pages 79 –83, 2006.

[161] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *International Conference on Program Comprehension*, pages 91 –102, June 2007.

[162] M. Riebisch. Supporting evolutionary development by feature models and traceability links. In *International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 370 – 377, May 2004.

[163] J.L. Rodgers and W.A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[164] C.K. Roy and J.R. Cordy. An empirical study of function clones in open source software. In *15th Working Conference on Reverse Engineering (WCRE)*, pages 81–90, October 2008.

[165] C.K. Roy, J.R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[166] S. Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9:143–192, 2000.

[167] R. Settimi, J. Cleland-Huang, O. Ben Khadra, J. Mody, W. Lukasik, and C. DePalma. Supporting software evolution through dynamically retrieving traces to uml artifacts. In *International Workshop on Principles of Software Evolution*, pages 49 – 54, September 2004.

[168] J.S. Shirabad, T.C. Lethbridge, and S. Matwin. Supporting software maintenance by mining software update records. In *International Conference on Software Maintenance.*, pages 22–31. IEEE, 2001.

[169] Josep Silva. A vocabulary of program-slicing based techniques. *ACM Computing Surveys (To appear)*, 2011.

[170] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 174–188, USA, 2010. ACM.

[171] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *ACM SIGPLAN Notices*, volume 35, pages 264–280. ACM, 2000.

[172] E. Burton Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[173] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[174] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000.

[175] S.R. Tilley. Domain-retargetable reverse engineering. ii. personalized user interfaces. In *International Conference on Software Maintenance*, pages 336 –342, September 1994.

[176] S.R. Tilley. Domain-retargetable reverse engineering. iii. layered modeling. In *International Conference on Software Maintenance*, pages 52 –61, October 1995.

[177] S.R. Tilley, H. Muller, M.J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Conference on Software Maintenance*, pages 142 –151, September 1993.

[178] R.J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1):35–52, 1994.

[179] R. Vanciu and V.T. Rajlich. Hidden dependencies in software systems. In *International Conference on Software Maintenance (ICSM)*, 2010.

[180] B.L. Vinz and L.H. Etzkorn. Improving program comprehension by combining code understanding with comment understanding. *Knowledge-Based Systems*, 21(8):813 – 825, 2008.

[181] Y. Wang, W.-T. Tsai, X. Chen, and S. Rayadurgam. The role of program slicing in ripple effect analysis. In *SEKE*, pages 369–376, 1996.

[182] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[183] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[184] P. Weißgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, page 9, May 2007.

[185] N. Wilde, R. Huittand, and S. Huitt. Dependency analysis tools: Reusable components for software maintenance. In *Conference on Software Maintenance*, pages 126 – 131, October 1989.

[186] F.G. Wilkie and B.A. Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2):157–164, 2000.

[187] D. Willmor, S.M. Embury, and Jianhua Shao. Program slicing in the presence of database state. In *International Conference on Software Maintenance (ICSM 2004)*, pages 448–452, September 2004.

[188] K. Wong. On inserting program understanding technology into the software change process. In *Fourth Workshop on Program Comprehension*, pages 90 –99, March 1996.

[189] T. Xie and J. Pei. Mapo: mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 54–57. ACM, 2006.

[190] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30:1–36, March 2005.

[191] S.M. Yacoub, H.H. Ammar, and T. Robinson. Dynamic metrics for object oriented designs. In *Sixth International Software Metrics Symposium*, pages 50–61, 1999.

[192] S.S. Yau, J.S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International*, pages 60–65. IEEE Computer Society, 1978.

[193] S.S. Yau, R.A. Nicholi, J.J.P TSAI, and S.S. Liu. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, 14(8):1128–1144, 1988.

[194] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Using version information for concern inference and code-assist. In *Tool Support for Aspect-Oriented Software Development Workshop*, November 2002.

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

[196] Z. Yu and V.T. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 293–299, Canada, 2001.

[197] Y. Zhou, M. Wursch, E. Giger, H. Gall, and J. Lu. A bayesian network based approach for change coupling prediction. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 27–36, October 2008.

[198] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In *International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

[199] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, May 2004.

[200] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31:429–445, 2005.