Supporting Architecture Evolution by Mining Software Repositories

Adam Vanya

2012

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems. SIKS Dissertation Series No. 2012-03

This work has been carried out as part of the DARWIN project at Philips Healthcare under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

Reading Commitee:

dr. P. America (Philips Research)
prof. dr. S. Brinkkemper (Utrecht University)
prof. dr. H. Gall (University of Zurich)
dr. R. Krikhaar (VU University Amsterdam)
dr. A. Zaidman (Delft University of Technology)

ISBN 978-94-6108-261-9
NUR 982

Printed by Gildeprint Drukkerijen - The Netherlands
Printed on 100gsm G-print

Typeset in LaTeX by the author

VRIJE UNIVERSITEIT

Supporting Architecture Evolution by Mining Software Repositories

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op vrijdag 2 maart 2012 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Adam Vanya

geboren te Esztergom, Hongarije

promotor:	prof.dr. J.C. van Vliet
copromotoren:	dr. A.S. Klusener
	dr. R. Premraj

Contents

List of Figures

List of Tables

1 Introduction

The way a software system is decomposed into a set of components greatly affects the amount of effort spent on the development and maintenance of that system. Changes that are not limited to a single component are likely to require more communication between developers, delays in development and increased amount of test cases. Therefore, one of the most challenging tasks of software architects is to decompose the software system such that the resulting components can evolve as independent as possible. In most cases, software systems are already decomposed in some way and the task of the architect then is to assess the state of the decomposition, find the unwanted couplings and improve the decomposition if necessary. How to help architects identify and investigate those unwanted couplings which hinder the sound evolution of the software system is described in this thesis.

1.1 Software Evolution

Everyone uses software in one way or another. Simple, every day activities, like watching the television, calling someone or buying goods with your bank card, all require software to be used. Aeroplanes, MRI scanners and printers are only a few examples for those complex software intensive systems which are used and trusted by millions of us every day. Even some of the greatest achievements of humanity, like visiting far away planets with spaceships, would not have been possible without developing complex but reliable software systems.

In spite of the fact that software is commonly used, software has more than a single definition [IEEE, 1990, Linberg, 1999, Wikipedia, 2010]. Those definitions mainly differ in the level of abstraction and the type of software considered when the definition was created. In this thesis the definition from the Concise Oxford English Dictionary is used: software consists of "*Programs and other operating information used by a computer*". More concrete definitions refer to operating information as "documentation" and "data".

Most of the software developed today has to operate in and interact with its real world environment. The software built to operate mobile phones, for instance, has to perform predefined actions when the user pushes on a button. Also, mobile phones have to make use of the few

Parts of this chapter has been published as:

Vanya, A., Klusener, S., Premraj, R., van Rooijen, N., and van Vliet, H. *Identifying and Investigating Evolution Type Decomposition Weaknesses* In Van der Laar, P. and Punter, T. (Eds.) *Views on Evolvability of Embedded Systems*. Springer-Verlag, 2011. ISBN 978-90-481-9848-1

standard communication networks. In general, the environment of the software puts additional constraints on how software should operate.

The environment in which software needs to operate changes over time. Not a long time ago, for instance, it was enough if mobile phones could make use of the communication networks either in Europe or in America. However, our world is getting more globalized and people are traveling much more than ever before. As people do not want to change their mobile phones when entering another continent, mobile phones are now required to operate in a more diverse environment than before.

Desktop computers are another example for the changing environment of software. Early desktop computers had a few megabytes of RAM and hard drive storage space. Furthermore, their processors were much slower than the ones produced today. With the advent of hardware technologies used today, terabytes of data can be stored in desktop memories and hard drives. Also, the processors produced nowadays are amazingly fast thanks to the multi core technology, besides others. Therefore, the operating systems on the desktop computers have a much different environment than even a decade ago.

Continuous changes in the environment of software lead to continuous changes in the software itself. In other words, software needs to evolve over time, otherwise the quality of the software decreases. [Lehman, 1980] The capability of software to evolve in a cost effective way is known as evolvability [Cook et al., 2000]. Changes made to the software can range from smaller ones, like modifying the implementation of a function, to bigger ones affecting the architecture of the software. Software evolution is a software engineering concept [Cook et al., 2000, Mens et al., 2010] and it refers to the process of changes made to the software to keep it up to date.

It is challenging to maintain the smooth evolution of software systems [Mens et al., 2005]. One has to consider many different aspects, also referred to as quality characteristics and quality attributes [Brcina et al., 2009]. For instance, reducing complexity [Suh and Neamtiu, 2010], managing variability [Babar et al., 2010] and creating traceability [Rochimah et al., 2007] are some of the major concerns of software architects when they design for and maintain evolvability.

When evolving software, one needs to make sure, for instance, that the modifications introduced do not have unwanted side effects and that modifications are complete. This is especially difficult in case of big sized industrial software systems having a long time of development history. The difficulty of evolving software is further complicated by what Lehman described in [Lehman, 1980]: while software evolves, its quality to evolve degrades unless some actions are taken to keep that software evolvable. These actions include the identification and resolution of those issues which hinder software to evolve easily.

Depending on the source of information used, there are different approaches to identify evolution related issues in the software. Some of them observe how well different type of software artifacts, like requirements, design and test cases are linked together to evaluate traceability [Rochimah et al., 2007]. Others compare how the structure of the development organization differs from the structure of the software developed [Del Rosso, 2009]. Yet another group of approaches take a single version of the software and identify couplings between different com-

ponents of the software which should evolve independently from each other. Based on the type of coupling identified, approaches addressing the maintenance of software evolution can further be differentiated. Such couplings can be, for instance, dynamic (between run-time entities) [Arias et al., 2008], static (include and call relations between files and methods) [Breu, 2005], and semantic (which software entities implement the same concept) [Kuhn et al., 2005].

The approach described in this thesis addresses software evolvability by making use of the change information stored in version management systems. Version management systems, like CVS or Clearcase, are one kind of software repositories which are analyzed, for instance to understand how changes were introduced to the software. The research field analyzing software repositories is known as mining software repositories. In that sense, this thesis contributes to the field of mining software repositories.

There are good reasons that the research presented addresses software evolvability by analyzing software repositories. Since the presented research was carried out in an industrial environment, the software architects in that environment had an influence on what to research and how. They were supporting research which could improve the independent evolution of the software system's components. Based on this interest of the architects, the possible research directions were determined and divided between the PhD students being involved in the same research project. The author of this thesis, based on his preference, was assigned to research how to assess software evolvability by mining software repositories. The next section provides a short introduction to the field of mining software repositories.

1.2 Mining Software Repositories

During software development and maintenance activities a huge number of artifacts is created. For instance, implementing a new functionality requires developers to create new or new versions of files for integration purposes. Also, it is the nature of software development that developers are introducing bugs to the software implemented. Based on the bugs found during testing activities, problem reports or change requests are created and archived to manage, for instance, bug fixing. Overall, the artifacts created during the evolution of the software (like file versions and problem reports) are stored in so called software repositories.

Mining software repositories (MSR) is a research field where techniques are investigated to analyze software repositories. The main purpose of MSR is to understand the process of software evolution and changes made to the software system in order to improve the evolution of the that system. Although software repositories are in use for a while now, MSR is a relatively young research field; its main conference (also called MSR) started in 2004. Kagdi et al. [2007a] suggests charactering studies contributing to MSR along the following dimensions: (1) the type of software repository utilized, (2) the purpose why software repositories are analyzed, (3) the methodology applied, and (4) the evaluation of the approach used.

MSR related research investigates mainly three types of software repositories: version management systems [Zimmermann et al., 2005], defect-tracking systems [Ostrand and Weyuker, 2004] and repositories archiving communication between project personnel [Ohira et al., 2005]. Version management systems have evolved quite a bit since they first appeared. Early version

management systems, like CVS, store every file version created during the evolution of a software system. Furthermore, they store change meta-data related to every individual file change. Such meta-data include which file was changed, who changed that file, when and (optionally) for what reasons. More advanced version management systems, like Subversion also archive which of the file changes were submitted (committed) together. The submission of files to the version management system, however, does not necessarily mean the submission of interrelated files; it depends on the software process applied. Therefore, the most advanced version management systems, like ClearCase Unified Changed Management and Perforce, support relating file changes based on which of them were made due to the same development task. Such tasks can be problem report resolutions of feature additions. In spite of such an advance of version management systems, still many of the currently used ones are of the early type, like CVS. Therefore, one of major challenges of MSR is to relate modifications of files if it is not supported by the version management system itself.

Defect-tracking systems are mainly used to track defects or bugs in the software system from their identification till they are resolved. Defects or bugs are typically found when the software system is tested and they are pointing to improper behavior of the software system. Tests may reveal, for instance, that a required feature is not properly implemented. Similar to the version management systems, defect-tracking systems can also store some meta-data. In this case, the meta-data can consist of, for instance, the description of the bug or defect, the timestamp of report, which maintainer was assigned, state and priority.

Software repositories archiving communications between project personnel are populated when project members are exchanging knowledge with one-another. Most commonly, it is done by writing a e-mail, using a chat program or making a phone call. These sources of information are known to be important to understand how the system evolved, since communicating project members share, besides others, the reasons for making changes.

Software repositories are analyzed for multiple reasons. In general, software repositories are studied to learn about the strong points and weaknesses of software development processes. The lessons learned may then help to make adjustments to the software development process followed and/or to the software developed. When learning about the software process, software repositories can be of great help. For instance, to understand the potential consequences of a change in the software system. Such a consequence can be that a modification to a given file requires modifications to three other files. Another reason is to point out unwanted couplings in the decomposition of the software system. If software entities from different components of the software system have changed frequently together, than it may point to an unwanted coupling. Yet another class of studies analyzed software repositories to add semantics to the changes that occurred. Those studies answer questions like, which function was changed and in what way, did a change add new text or it is copied from somewhere else.

The methodologies which are used to analyze software repositories can be divided into two complementary classes. During the evolution of the software, changes take that software from one version to another. The first class methodologies derive high-level system properties for each version of the software, like complexity, and study the changes of those properties. The second class of methodologies, on the other hand, analyze the actual changes between two

consecutive versions of the software. Another way to classify methodologies used by MSR research is the abstraction level of the software entities studied. Some observe changes at the level of files, while others take a more abstract (sub-system level) or more detailed (method or class level) approach.

When it comes to the evaluation of methodologies mining software repositories, the repositories of open source systems are used most of the time. The advantages of using the software repositories of open source systems is that there are are many of them and widely available for everyone who wants to analyze them. It also means that the research done on such repositories is repeatable. Open source development, however, is relatively different from closed source development. Therefore, some of the observations with open source systems may not be as valid with closed source systems. Since many of the software systems developed today are still closed source, evaluating MSR methodologies on those software systems is important. However, one of the major downsides of MSR research conducted on closed source software systems is that it is difficult to repeat. Independent from the type of software system studied (open or closed source), the historical data sorted by software repositories is typically divided into two parts: one to build the methodology and another one to evaluate that methodology.

Mining software repositories is a demanding process. One of the reasons is that software repositories contain a huge amount of information which needs to be extracted and processed. To mitigate that problem, one should know which portion of the sorted information is actually useful to be considered. Since the data in software repositories got captured as a result of executing a software process, better knowing the data requires one to know the software process as well. Furthermore, in order to analyze and interpret the data extracted, the data has to be presented in some way, for instance using visualizations.

Using the classification of Kagdi et al. [2007a], the research presented in this thesis can be characterized as follows. The research carried out analyzes data from the version management system of an industrial software system. From that software repository change related data is mined to help the software architect find and resolve unwanted couplings in the decomposition of the software system. To do so, the methodology applied investigates the actual changes made to files between different versions of the software system. The evaluation of the work presented is done on the industrial software system analyzed. During the evaluation, the feedback of architects and developers are used in the first place. The reason that MSR was applied this way comes partly from the characteristics of the software system studied and partly from the actual need of the architect supported. The research presented here was conducted within the framework of the Darwin project. The next section elaborates further on that project.

1.3 The Darwin Project

The Darwin project has been a research project under the supervision of the Embedded Systems Institute. The project started in September 2005 and after five years it finished in September 2010. Within the project five Dutch universities have cooperated with one-another and with the industrial partner Philips Healthcare MRI to study the challenges related to system evolvability.

(a) An MRI machine

(b) An MRI image

Figure 1.1: Magnetic Resonance Imaging

The objective of the Darwin project was to create tools, methods and architectures for optimizing system evolvability. Researchers from the universities involved in the Darwin project addressed both hardware and software evolvability. Some of them worked on how software and the accommodating hardware can evolve together. Van der Laar et al. [2007] describe the objective of the Darwin project more into details.

The Darwin researchers addressed different research problems of system evolvability. These research problems were defined in collaboration and agreement with Philips Healthcare MRI. This was done to make sure that each of the researchers address concrete research problems that troubled the organization. As the research problems addressed were interrelated, researchers worked time-to-time together to benefit from each others knowledge. Researching system evolvability in Darwin not only led to a number of publications. It also resulted in concrete modifications made to the system's design and development processes.

1.4 The Software System Analyzed

This thesis is based on a research which analyzed the software system of Philips Healthcare MRI. The software system analyzed is designed to operate magnetic resonance imaging or MRI machines and their peripherals. MRI machines are primarily used in hospitals to visualize detailed internal structures of the human body. Visualization of that matter is helping doctors to diagnose illnesses of patients or to control the effects of a medical interventions. With the help of an MRI machine a doctor can, for instance, check if the patient has a tumor or not. Another usage scenario is to investigate which part of the brain gets active as a response to a given stimulus. Figure 1.1a shows an MRI machine, and Figure 1.1b provides an example for

an image acquired with such a machine. In that example the internal structures of a patient's head are visible.

That system has approximately 34 000 files and the total number of lines of code is more than nine million. Hundreds of developers are working on the software system at three development sites, each of them on a different continent. The complexity of the software system has increased over time and it became a challenge to further evolve it.

The programming languages used to develop the software system are mainly C#, C++ and C. When developing and maintaining the software system, developers are working on *development tasks*, such as feature modification, problem report resolution and the like. Which software entities were changed because of the same development task is, however, not captured. In this thesis historical information, namely file check-in related meta-data is used to approximate which modifications belong to the same development task. Check-in related meta-data is stored from the last ten years in a version management system called ClearCase. For every file modified, the check-in related meta-data provides information on who introduced the modification to that file, and when. Furthermore, ClearCase also captures the reason of modifications given that the developers provide it.

The analyzed software system has been developed using *building blocks*. Building blocks are the directories representing the next level of abstraction above individual files in the file hierarchy. A building block is the unit of reuse. The internal structure of a building block is invisible to the other building blocks in the system. Usage of building block functionality is through import and export interfaces only; see also [van der Linden and Müller, 1995]. One abstraction level above building blocks we find *subsystems*. A subsystem is a set of building blocks mainly related to the same major functionality of the software system. In the directory structure of the software system, each of the subsystems is represented by a directory. The system comprises around 600 building blocks organized into 16 subsystems.

Based on their common properties subsystems can further be grouped together. A grouping of all the subsystems form a decomposition of the whole software system. The following decompositions were of interest to the architects of the software system studied:

1. subsystem decomposition
2. development group decomposition
3. release group decomposition
4. deployment group decomposition
5. development site decomposition

The first decomposition groups building blocks into subsystems, mainly based on functionality. The last four decompositions listed are different abstractions above the subsystem level. Development groups are only allowed to modify the subsystems they are responsible for. At the moment of writing, the development group decomposition is not yet implemented in the organization. It is a decomposition which will be used in the future. In the present study, a

development group decomposition is a fictitious one, used to assess a possible development group decomposition. Release groups contain collections of subsystems which should be released independently. Deployment groups comprise collections of subsystems which should be deployed to different pieces of hardware. Finally, subsystems form groups based on the development site they are developed at.

While conducting the research we interacted with software architects and software engineers. Our interaction was especially frequent with a lead architect. With him we had formal meetings nearly every week. Both the types of decomposition and the actual decompositions of the system we studied were given by the architect we worked with.

1.5 Problem Statement

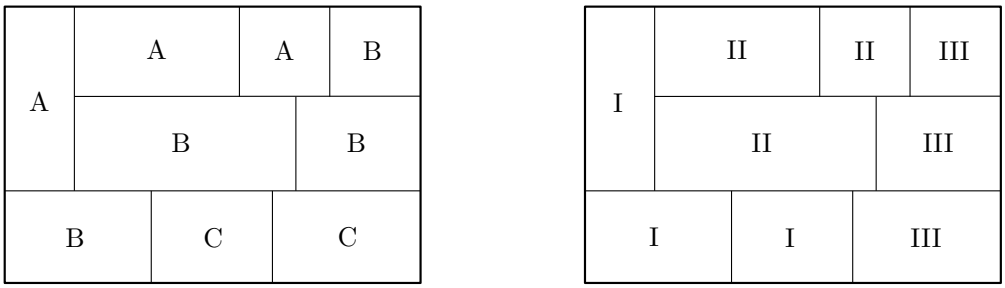
As mentioned before, evolving software systems, like the one studied, is a difficult task to achieve. One of the reasons is that software systems are typically huge and complex. It is not uncommon that industrial software systems contain thousands of files and an order of magnitude more functions. When a developer modifies one of those files, he has to make sure, for instance, that he does a consistent change to the software system. This requires the developer to think about which other files to modify related to his initial change. Also, modifications to a software system have to respect additional constraints. Modifications to the software system have to be ready on time, the cost of development has to stay below a predefined threshold and the resulting software system has to be of good quality.

A typical way to develop and maintain complex software systems is to use the principle of divide and conquer also known as separation of concerns [Parnas, 1972]. Applying that principle results, among others, in a decomposition of the software system into a set of *decomposition elements*. A decomposition element is a set of software entities, e.g. set of files, classes or methods, which are used to build the software system from. The decomposition elements are disjoint and their union contains all software entities of the software system. Decomposition elements can be modules but not necessarily. In our study environment, a decomposition element can be, for instance, a single subsystem or the collection of subsystems of some given development group.

There is more than one way in which a software system can be decomposed. For instance, one may consider two software entities to belong to the same decomposition element if they

1. belong to the same subsystem (subsystem decomposition)
2. are developed by the same group of developers (development group decomposition)
3. are deployed to the same piece of hardware (deployment group decomposition).

In case of the software system developed at Philips Healthcare MRI, unions of subsystems form the decomposition elements of development group and deployment group decompositions. Figure 1.2 illustrates the three described decompositions of the software system of Philips Healthcare MRI. There, subsystems are indicated by the smaller rectangles. The letters (A, B



(a) Development Group Decomposition

(b) Deployment Group Decomposition

Figure 1.2: Two decompositions of the same software system

and C) and numbers (I, II and III) used in Figure 1.2a and Figure 1.2b show which subsystems belong to the same decomposition elements regarding the development group and deployment group decompositions, respectively.

A good decomposition needs to fulfill several requirements. One of these requirements is that the decomposition elements have to evolve as independent as possible. For instance, a change to files maintained by one development group should not require a change to files maintained by another development group. In other words, changes should ideally by limited to a single decomposition element. By fulfilling this requirement it is possible to reduce the number of files to be modified due to a change, reduce testing time, and the like [Brcina et al., 2009]. Therefore, having a decomposition of the software system where the elements can evolve as independent as possible is important to help overcome the complexity of the software system and, as a consequence, help respect the defined time, cost and quality constraints.

A decomposition of the software system may not completely fulfill the requirement of independent evolution. Even if the decomposition was initially created such that decomposition elements could evolve independently, the structure of the software system may degrade over time and that effects the independent evolution of the decomposition elements. If software entities from one decomposition element are likely to be changed as a consequence of a change to software entities in another decomposition element, then we may face an *unwanted coupling*.

Removing the unwanted couplings is one of the tasks of the software architect. This way, the architect can keep the decomposition in such a shape that the development and maintenance activities can keep benefiting from the fact that the decomposition elements can evolve relatively independently. Software architects have, however, many tasks to perform, see [Hofmeister et al., 2000, Bass et al., 2003, Clements et al., 2007]. These tasks include, amongst others, the communication with stakeholders, the translation of requirements to design decisions, and the documentation and assessment of the software architectures developed. Furthermore, architects typically are pressed for time and have limited time available to resolve unwanted couplings. During this limited time, architects seek to address the most severe ones.

In this thesis a method is described which was developed to help software architects iden-

tify and investigate the unwanted couplings related to the software system they maintain. The method described in this thesis was applied to the software system of Philips Healthcare MRI. With the help of the method the architect could improve the decomposition of the software system in several cases. While dealing with unwanted couplings, a retrospective approach is utilized with the yesterday's weather assumption [Gîrba et al., 2004]; if software entities have been changed frequently in the past then we assume that they will also change together in the near future. Knowing which entities are likely to change together in the future we can identify potential unwanted couplings.

1.6 Research Questions

From the problem statement follows that the main research question of this thesis is:

MRQ *How to help software architects improve the decomposition of a software system?*

Research question MRQ is rather complex. Therefore, this thesis addresses MRQ by defining and answering three related sub-questions (RQ-1, RQ-2 and RQ-3). The premise is that by addressing the three sub-questions, the main research question MRQ is also addressed. Next, those sub-questions are motivated and described.

One can improve the decomposition of a software system only if the unwanted couplings are known. When independent evolution is of concern, potential unwanted couplings are indicated by frequently co-changed software entities from different decomposition elements. Therefore, our first research question is:

RQ-1 *How to identify groups of frequently co-changing software entities?*

Not all groups of frequently co-changing software entities indicate unwanted couplings. For instance, if all the co-changing software entities are coming from the same decomposition element. Furthermore, in practice only those potential unwanted couplings are of interest to the software architect which severely impact the evolution of the software system. These concerns lead to the following research question:

RQ-2 *How to select those groups of co-changing software entities which may indicate unwanted couplings?*

Resolving unwanted couplings is what eventually leads to a better system decomposition. Prerequisites to do so include understanding the reasons why software entities co-changed and finding alternative solutions to eliminate the unwanted couplings. Knowing that interactive visualizations are powerful means to present and analyze huge amount of information, the following research question emerges:

RQ-3 *How can interactive visualizations help facilitate a detailed analysis of potential unwanted couplings?*

1.7 Research Methods and Methodology

This research was conducted at Philips Healthcare MRI between September 2006 and September 2010. As mentioned, the main problem investigated was how to help software architects identify and investigate the unwanted couplings related to the software system they worked on. This problem was addressed by executing a number of studies while working closely together with the industrial partner of the Darwin project.

As part of the co-operation with Philips Healthcare MRI, weekly interviews were scheduled with the software architect. During the interviews research progress was discussed and feedback was collected. Next to that, developers were contacted in different ways to get insights from them. Typically face-to-face meetings were organized with them, but from time to time they were also asked to fill out a questionnaire or an on-line survey. Furthermore, experts of the company were contacted who could tell more about the software development process and the way to retrieve historical information from the version management system applied.

The research presented by this thesis is based on action research cycles. As described by Baskerville [1999], action research solves immediate problem situation by taking action. This is what the Darwin project aimed for. As described by Susman and Evered [1978], an action research cycle has the following five main stages:

Diagnosing Diagnosing corresponds to the identification of the primary problems that are the underlying causes of the organizations desire for change. Diagnosing involves self-interpretation of the complex organizational problem, not through reduction and simplification, but rather in a holistic fashion. This diagnosis will develop certain theoretical assumptions (i.e., a working hypothesis) about the nature of the organization and its problem domain.

Action Planning Researchers and practitioners then collaborate in the next activity, action planning. This activity specifies organizational actions that should relieve or improve these primary problems. The discovery of the planned actions is guided by the theoretical framework, which indicates both some desired future state for the organization, and the changes that would achieve such a state. The plan establishes the target for change and the approach to change.

Action Taking Action taking then implements the planned action. The researchers and practitioners collaborate in the active intervention into the client organization, causing certain changes to be made. Several forms of intervention strategy can be adopted. For example, the intervention might be directive, in which the research ”directs” the change, or non-directive, in which the change is sought indirectly. Intervention tactics can also be adopted, such as recruiting intelligent laypersons as change catalysts and pacemakers. The process can draw its steps from social psychology, e.g., engagement, unfreezing, learning and re-framing.

Evaluating After the actions are completed, the collaborative researchers and practitioners evaluate the outcomes. Evaluation includes determining whether the theoretical effects of the action were realized, and whether these effects relieved the problems. Where the change was successful, the evaluation must critically question whether the action undertaken, among the myriad routine and non-routine organizational actions, was the sole cause of success. Where the change was unsuccessful, some framework for the next iteration of the action research cycle (including adjusting the hypotheses) should be established.

Specifying Learning While the activity of specifying learning is formally undertaken last, it is usually an ongoing process. The action research cycle can continue, whether the action proved successful or not, to develop further knowledge about the organization and the validity of relevant theoretical frameworks. As a result of the studies, the organization thus learns more about its nature and environment, and the constellation of theoretical elements of the scientific community continues to benefit and evolve.

In order to address research questions RQ-1, RQ-2 and RQ-3 three action research cycles were executed one after another. In the first one (1) the problem of identifying groups of co-changing software entities was identified, (2) an approach to group software entities was designed, (3) the approach was applied on the data extracted from the version management system of Philips Healthcare MRI, (4) the groups of software entities were evaluated and (5) the lessons learned were collected and described. One of the lessons learned was that there are too many of the groups identified and a selection method is needed. This observation motivated to execute the second research cycle.

In the second research cycle (1) the problem of selecting groups of co-changing software entities was further investigated, (2) an approach to characterize and select those groups were designed, (3) based on the input from the software architect we supported, a selection was defined and executed, (4) the selected groups were analyzed and (5) the lessons learned were summarized. It was between the observations that the analysis of the groups need to be better supported, which required to execute the third research cycle.

In the third research cycle (1) the problem of analyzing groups of co-changing software entities was further investigated, (2) an interactive visualization were designed to support the analysis, (3) the selected groups of co-changing software entities were analyzed together with the architect and developers, (4) this resulted in an evaluation of the visualization itself, (5) the lessons learned were formulated.

The first research cycle is addressed in Chapter 2 and Chapter 3. The second research cycle is addressed in Chapter 4. The third research cycle is addressed in Chapter 5. During the execution of the three research cycles a process to help a software architect improve the decomposition of a software system gradually emerged. This process together with its steps gives an answer to the main research question of this thesis: MRQ.

The process has the following steps:

Step 1 Extraction of the raw data from the version management system.

Step 2 Identification of groups of software entities changed because of the same development task, like a problem report resolution.

Step 3 Identification of the groups of frequently co-changing software entities.

Step 4 Selection of the identified groups of frequently co-changing software entities.

Step 5 Analysis of the selected groups of frequently co-changing software entities.

Step 6 Based on the outcome of the analysis, deciding how to improve the system's decomposition and takes actions.

Step 2 and Step 3 of the process are addressing RQ-1. Step 4 is addressing RQ-2 and Step 5 gives an answer to RQ-3. The process is further described in Chapter 6. While executing the process to help the software architect several lessons were learned, see also Chapter 6 for more details. These lessons can be organized into three major, but interrelated, categories:

- *frequent feedback loops* with stakeholders such as architects and developers are necessary so that they understand the approach used.
- *knowing the process* of how the software is being modified helps in the proper interpretation of the data retrieved from the software archive.
- *knowing the data* we use is needed for identifying the unwanted couplings that really matter.

1.8 Outline of the Thesis

The following chapters are describing the studies done to realize the three research cycles discussed in Section 1.7. These studies are answering the research questions RQ-1, RQ-2, RQ-3 and all together they answer the main research question MRQ.

- In Chapter 2 it is investigated how accurately different techniques can approximate sets of software entities which changed because of the same development task (prerequisite to answer RQ-1).
- In Chapter 3 the application of a clustering algorithm is described to identify which groups of software entities have change together frequently (RQ-1).
- In Chapter 4 the frequently co-changing groups of software entities are characterized and filtered to identify those which are pointing to unwanted couplings (RQ-2).
- In Chapter 5 it is investigated how interactive visualization of the selected groups of software entities can be of help to reason about and resolve unwanted couplings (RQ-3).

Chapter 6 concludes this thesis by describing the (1) process which emerged during helping the software architect improve the decomposition of the software archive, (2) the lessons learned along the way and (3) the future challenges to be addressed.

1.9 List of Publications

Most of the research presented in this thesis has been published. The chapters of this thesis are based on the following publications:

Parts of Chapter 1 has been published as:

- Vanya, A., Klusener, S., Premraj, R., van Rooijen, N., and van Vliet, H.
  Identifying and Investigating Evolution Type Decomposition Weaknesses
  In Van der Laar, P. and Punter, T. (Eds.) *Views on Evolvability of Embedded Systems*. Springer-Verlag, 2011. ISBN 978-90-481-9848-1

Parts of Chapter 2 have been published as:

- Vanya, A., Premraj, R., and van Vliet, H.
  Approximating Change Sets at Philips Healthcare: A Case Study.
  In *15^{th} European Conference on Software Maintenance* (CSMR '11), pages 121–130, IEEE Computer Society, 2011.

Parts of Chapter 3 have been published as:

- Vanya, A., Hofland, L., Klusener, S., van der Laar, P., and van Vliet, H.
  Assessing Software Archives with Evolutionary Clusters.
  In *16^{th} International Conference on Program Comprehension* (ICPC '08), pages 192–201, IEEE Computer Society, 2008.

Parts of Chapter 4 have been published as:

- Vanya, A., Klusener, S., van Rooijen, N., and van Vliet, H.
  Characterizing Evolutionary Clusters.
  In *16^{th} Working Conference on Reverse Engineering* (WCRE '09), pages 227–236, IEEE Computer Society, 2009.

Parts of Chapter 5 have been published as:

- Vanya, A., Premraj, R., and van Vliet, H.
  Interactive Exploration of Co-evolving Software Entities.
  In *14^{th} European Conference on Software Maintenance and Reengineering* (CSMR '10), pages 269–272, IEEE Computer Society, 2010.

- Vanya, A., Premraj, R., and van Vliet, H.
  Resolving Unwanted Couplings Through Interactive Exploration of Co-evolving Software Entities – An Experience Report. Journal of Information and Software Technology (2011) – to be published

Parts of Chapter 6 have been published as:

- Vanya, A., Klusener, S., Premraj, R. and van Vliet, H.,
  Supporting software architects to improve their software system's decomposition - lessons learned. Journal of Software Maintenance and Evolution: Research and Practice. (2011) doi: 10.1002/smr.574

Approximating Change Sets at Philips Healthcare

2

A single development task such as solving a bug or implementing a new feature often involves changing a number of entities, also known together as a change set. Change sets can be approximated from the version control system. They are then used by the architects and developers to take important decisions. So change sets need to be approximated carefully. It is common to assume that two entities checked-in less than a small time interval from each other, and having the same meta-data associated with them, belong to the same transaction. Transactions may be good approximations of change sets if developers commit change sets in one go and if the required meta-data is available. This is however not the case in the industrial environment (Philips Healthcare) we study. This chapter presents a case study in which we investigated how change sets can be approximated in an environment with a complex workflow and limited meta-data in the version repositories. By doing so, we execute Step 1 and Step 2 of the process described in Section 1.7 to help the software architect. We found that, dependent on the commit practices used, a suitable time intervals between check-in timestamps of files has to be determined and leveraged to reliably approximate change sets.

2.1 Introduction

A single development task such as resolving a bug or implementing a new feature often involves changing a number of files in a series of changes in the software system. Together, these series of changes associated with a single task are referred to as a *change set*. Knowledge about such change sets help architects and developers in evolving a software system. For instance, it may help architects assess the decomposition of a software system and analyze the dependencies between different parts of that decomposition to initiate any necessary restructuring activities. But such decisions can be costly and have far-reaching consequences, so it is crucial that they are taken with care and are based on reliable change set data. Actions based on grossly inaccurate change sets may result in undesirable consequences that may be costly to fix.

Changes to software systems are captured in version control systems like CVS, SVN, or IBM ClearCase. Previous research such as that by Zimmerman and Weißgerber [2004] and Fis-

Parts of this chapter has been published as:

Vanya, A., Premraj, R., and van Vliet, H. Approximating Change Sets at Philips Healthcare: A Case Study. In *15^{th} European Conference on Software Maintenance* (CSMR '11), pages 121–130, IEEE Computer Society, 2011.

cher et al. [2003b] proposed methods to process meta-data from such version control systems to extract sets of changes made to the software system. However, their research has focussed on leveraging available meta-data (developers' names, commit messages, and timestamps) to extract individual *transactions*. Transactions, also commonly known as *commits* [Alali et al., 2008], are different from change sets in that they only capture the set of new or modified files that were *checked-in* together into the version control system. Change sets, on the other hand, refer to a set of transactions that together represent all changes to the software system to complete a specific development task. Indeed, it is quite possible that a single transaction may also be the change set for a task. The availability of the set of transactions inferred from version control systems has spawned several new research initiatives such guiding further changes [Zimmermann et al., 2005], improving programmer productivity [Kerstin and Murphy, 2006], studying team development culture [Weissgerber et al., 2007], and the like.

A similar area of interest at Philips Healthcare in the Netherlands (our case-study environment) is improving the control over the evolution of their software systems by studying past changes. At Philips Healthcare, software architects are transitioning their systems such that development tasks can be completed with as few changes as possible to files across different parts of the system decomposition (e.g. across different sub-systems). Such organisation is expected to help them manage their development processes better, increase efficiency, and cut costs. A starting point to initiate the planned transition is to study and audit the composition of change sets to date and identify the causes behind cross sub-system changes. Architects can then consider maintenance activities (such as moving files from one subsystem to another) to better facilitate the evolution of the system [Vanya et al., 2010].

The nature of development processes at Philips Healthcare, and the task at hand (analyse past changes to their system) requires that change sets, and not individual transactions, be studied to get a reliable picture of the past changes (more details in Section 2.3). While state-of-the-art version management systems (such as IBM Synergy) can be used to recover change sets directly, which in turn can be queried and analysed, costs associated with switching to them have daunted Philips Healthcare, like many other organisations, to upgrade. Thus, change sets have to be approximated from the version control system in use.

In this chapter, we present a case study conducted at Phillips Healthcare to support them in recovering approximated change sets from their version control systems. Our work differs fundamentally from that of Zimmermann and Weißgerber in the following two ways:

- While Zimmermann and Weißgerber focussed on recovering transactions, our goal is to recover change sets. These change sets most often comprise several transactions that are separated by varying gaps of time (Section 2.4).
- Furthermore, developers at Philips Healthcare rarely explain the rationale behind changes by adding commit messages to their transactions. So, in our case, the change sets had to be approximated without the use of commit messages as compared to Zimmermann and Weißgerber's approach that heavily relied on these messages.

As mentioned before, it is crucial that the approximated change sets are as correct as possible in order to reliably use them in decision making. Hence, in addition to presenting details

on inferring change sets with limited meta-data from our study environment, we also present the methods we used to evaluate the approximated change sets and gauge their accuracy (Section 2.5). Our evaluation setup is fairly general in that it can be adapted with little effort to other study environments to measure the accuracy of approximated change sets.

Thus, through this work, we make the following contributions:

1. Present a method used in our case-study environment to recover approximated change sets (which are different from transactions) from version control systems, that too with limited availability of meta-data.

2. Present an evaluation setup to assess the accuracy of approximated change sets in the environment that can be adapted and reused in other environments for the same purpose.

The results of our investigations show that it is possible to approximate change sets with reasonable accuracy even in environments with limited meta-data. We discuss our results in detail (Section 2.6), then present the threats to validity (Section 2.7), and bring this chapter to a close with conclusions and consequences of our work (Section 2.8).

2.2 Related Work

2.2.1 Change Set Definitions

Reviewing earlier work makes it clear that the phrase "change set" is rather overloaded. The definitions provided have in common that they describe a set of changes, but in which way the changes are related and what the changed entities actually are makes quite a difference in those definitions.

A considerable portion of the related work defines change sets as all the files that have been checked-in together into the version repository [Moser et al., 2008, Kagdi et al., 2007a,b]. Rastkar and Murphy [2009] extend this definition by requiring that all the files checked-in together should belong to the same development task. We have experienced that in practice, unless a strict development process is applied, developers often check-in file modifications together which relate to more than a single development task.

Hassan and Holt [2004] leave the technicality of commits aside and define change sets as all the entities which have to be modified because of a single development task. Such a change set may come from a single commit but it does not necessarily do so. The authors define a development task to be a new feature addition, feature enhancement or a bug fix.

According to McNair et al. [2007] a change set comprises logical commits. A logical commit is understood to be all the files modified because of a single modification request (MR). Change sets are then defined as any set of logical commits developers or architects consider to be logically related.

Estublier et al. [2005a] describe change sets as the first class entities in advanced versioning. It is emphasized that although a change set may contain any set of changes, a change set is found to be practically applicable when it is associated with a feature or task.

In this chapter we will refer to change sets as defined by Hassan and Holt. The reason to do so is that we cannot safely assume the commits of developers coincide with the submission of task implementations. Also, we believe that change sets as defined by Hassan and Holt are the ideal input for many evolution-related work. Our decision is further supported by the observations of Estublier et al. [2005a].

2.2.2 Change Set Approximation

When change sets are not directly available from the version control system applied then they have to be approximated. According to our best knowledge, so far only commits have been approximated based on the meta-data stored in version management systems. The seminal work on such approximations include the work of Zimmermann and Weißgerber [2004] and Gall et al. [2003]. They refer to such approximated commits as *transactions*.

In case developers check-in all the modifications together which relate to a single development task, transactions may suffice to approximate change sets. However this is not always the case (see Section 2.3).

2.2.3 Using Transactions

As stated, given a strict development process transactions can be considered as approximated change sets. Predicting future changes is one of the primary applications of such transactions [Zimmermann et al., 2005, Hassan and Holt, 2004]. Kagdi and Maletic [2007a] combine the evolutionary dependencies derived from transactions with static dependency couplings to further improve the prediction of future changes. Shiraban et al. [2003] predict changes by using artificial intelligence techniques on the transactions, whereas in [Zimmermann et al., 2005] and in [Ying et al., 2004] rule mining is applied on the same type of input.

Further, transactions have been used to:

- identify sub-systems that fulfill the requirement of independent evolution [Beyer and Noack, 2005]
- find strong inter-module dependencies [Ratzinger et al., 2005a, Gall et al., 2003]
- identify sets of co-changed files [Vanya et al., 2008, Antoniol et al., 2005]
- relate software entities and non-program artifacts and find traceability links [Zimmermann et al., 2003, Zimmermann and Weißgerber, 2004, Kagdi et al., 2007c]
- identify cross cutting concern candidates [Breu and Zimmermann, 2006]

Transactions, as we see, are used for several reasons by many earlier works. In general, these studies are using transactions to derive which entities changed together and leverage this information to support the decisions of architects and developers. In contrast, we focus on recovering change sets in our case study environment.

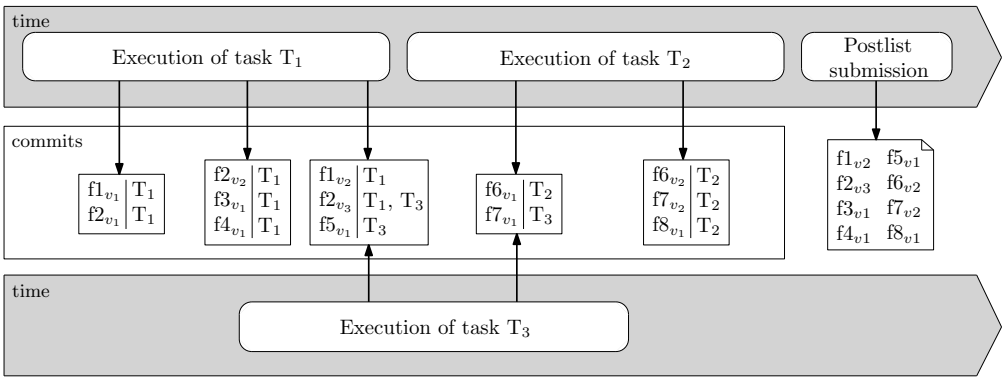


Figure 2.1: An example process instance executed by a developer

2.3 Case-Study Environment

In order to approximate the change sets, familiarity with the workflow in the case study environment is crucial. The manner in which changes are recorded depends on the software development process followed in the company. In the case of Philips Healthcare, the following practices are commonly followed in their the development process:

- During the execution of a single development task, developers typically commit files more than once.
- Developers do not need to make sure that each of their commits results in a buildable system.
- From time to time, developers work on more than one development task, especially in the problem report (PR) resolution phase of the software process. In such cases, a commit can concern more than one development task.
- After the execution of a few development tasks the summary of the modifications (a postlist) is sent to the integrator who checks if the system can be built with the modifications introduced.
- Especially when a complex feature is developed, several developers are working on the same development task.
- Developers are not obliged to annotate commits.

Figure 2.1 depicts a scenario where a developer executed three development tasks and submitted the related modifications to the system integrator (in form of a so called postlist). As we can see, during the execution of those development tasks files f1–f8 were committed.

In each commit Figure 2.1 indicates the files changed, the versions of those files and the id of the development tasks which served as a reason to modify those files. In the third commit, for instance, $f2_{v3} \mid T_1,T_3$ indicates that version 3 of file f2 was changed and committed because of development tasks T_1 and T_3. In the following paragraphs we elaborate further on some of the process characteristics listed. Note that the tasks (T_1,T_2, and T_3) presented in the figure are for illustrative purposes only and are not available in the version repository.

In the work culture at Philips Healthcare, software developers most often commit change sets to the repository via a series of intermediate commits with smaller changes rather than a single and complete commit. In Figure 2.1 we can see, for instance, that related to task T_1 three commits were made.

Many reasons prompt developers to complete change sets across a series of commits. One such reason that we identified by communicating with the developers is that their work is often disrupted by meetings, breaks, etc. and there is a general preference to commit all available changes before any such break to ensure against data loss. Another important reason identified is time to market pressure. Development tasks are parallelized in the company as much as possible. This encourages committing smaller iterations to maintain the workflow in the company. For instance, a developer may initially commit the interface of a component so as to enable development of other dependent components in parallel. Commits corresponding to the actual implementation of the component follow later. As a consequence, the complete set of changes to complete a single task may be separated over a number of commits. In the case of embedded software systems, the time interval between such commits may be much more than a few minutes because the life-cycle of some such changes involves hardware changes as well (which may take up to a month).

Furthermore, developers at Philips Healthcare are not obliged to annotate their commits; many of the commit messages in the version repository are left blank. As a result, it is often not possible to identify the motivations behind the check-ins by solely examining the repository. The only meta-data available for every check-in are at the file-level and include the developer who committed the change and the respective time-stamp.

Commits at Philips Healthcare may not only capture a part of a change set as described, but there are cases when the files modified because of more than one development task are committed together. The reason for inhomogeneous commits is that developers from time to time work on more than one development task in parallel. Working on more than one development task at the same time is done typically when the sets of files to be modified related to the developments tasks overlap. In Figure 2.1 we can see that the third commit (from left to right) is related to files which were modified because of tasks T_1 and T_3. There, file f2 was modified for both tasks.

As described, commits can be both incomplete and heterogenous with respect to change sets. This, however, does not introduce a problem when maintaining the consistency of the code. The reason is that the results of commits are not directly used to build and test the software system. The results of commits from one developer are not even directly accessible to other developers. Instead, typically after a series of commits a developer fills out a simple form where he indicates which files he modified and which the latest versions of those files are.

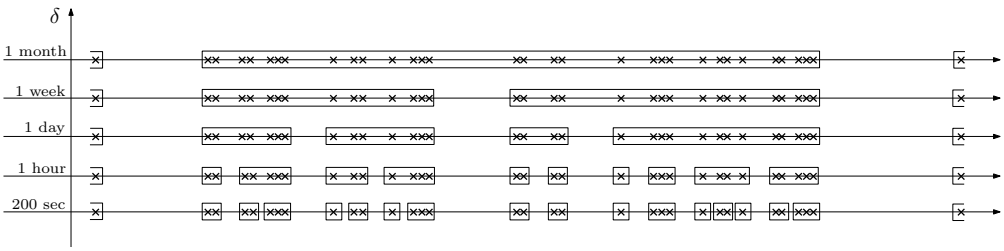


Figure 2.2: Change sets approximated with different time interval. A cross denotes a file checked-in to the version repository by a developer *D*

Such forms are also known in Philips Healthcare as *postlists*. In Figure 2.1 we can see that a postlist is created with all the latest versions of the files modified related to development tasks T_1, T_2 and T_3. Postlists are then sent to integrators who build and test the software system with the modifications in the postlist. If the system can be built and successfully tested then the modifications in the postlist will be consolidated. It mainly means that the modifications introduced to the files in the postlist will be made available to all developers. Postlists are further described in Section 2.5.2.

2.4 Approximating Change Sets

The most commonly adopted approach to approximate which files were checked-in together into a CVS-like version repository was proposed by Zimmermann and Weißgerber [2004]. They regarded files that were checked-in by the same *developer*, bore the same *commit message*, and updated in the repository within a *time interval* of 200 seconds from each other as a single *transaction*, i.e. an approximation of the result of a commit. The 200 second threshold was motivated by network latency arguments.

However, at Philips Healthcare, a single such transaction is unlikely to represent a complete change set because of their workflow as described in the previous section. Hence, Zimmermann and Weißgerber's approach was inapplicable to our environment since we are interested in change sets, and that too with no access to commit messages.

We hence adopted a different approach. Due to the lack of commit messages, we considered files checked in by a single developer within five different time intervals as a change set. These time intervals were chosen after consultation with the developers at the company and were selected as follows: 200 seconds, 1 hour, 1 day, 1 week, and 1 month. The longer time intervals are reflective of the workflow at Philips where change sets may spread across sometimes up to a month. Our task is to now investigate which one of the five time intervals can be used to approximate change sets as correctly as possible.

Our approach of approximating change sets is illustrated in Figure 2.2. On the time axis, a cross denotes a file checked-in to the version repository by developer *D*. Files that are checked-in to the repository within a set time interval (δ) are grouped together and considered as an

Table 2.1: The Approximated Change Sets (ACS)

		#Check-ins per ACS			
δ	#ACSs	Min.	Max.	Med.	Avg.
200 sec	115487	1	14002	2	8
1 hour	82571	1	14551	2	11
1 day	42447	1	14551	4	22
1 week	13568	1	19404	9	69
1 month	3408	1	27502	27	275

approximated change set. Figure 2.2 indicates approximated change sets identified with the five different δ values that we experimented with. As we can see, change sets approximated with a higher value of δ are often a union of change sets approximated with a lower δ value.

The number of approximated change set using each δ values is given in Table 2.1. As expected, the number of approximated change sets decreases for larger values of δ, while the average number of changes files included in the change set increases. From Table 2.1 we can also infer that certain change sets very likely do not refer to a single development task (e.g. the ones having around 14000 check-ins that may be results of merges).

But the approximated changes in Table 2.1 must be validated to be able to use them reliably for decision making. In the following section, we estimate the accuracy of the approximated change sets and investigate which time interval delivered change sets closest to the actual change sets.

2.5 Evaluation of Change Sets

After having approximated the change sets from the repository, we now perform an evaluation to judge their accuracy, i.e., how similar are the approximated change sets to the actual change sets. At Philips Healthcare, we had two options to conduct such an evaluation — these are presented below. We then reflect on the results from both evaluations to determine a suitable time interval.

2.5.1 Cross-checking with developers

The first option to evaluate the integrity of the approximated change sets is to check with the developers at Philips Healthcare. Given their experience and knowledge about the system, they are likely to be able to recall the development tasks performed and the associated changes.

We performed this evaluation by asking selected developers to participate in an on-line survey (screenshot in Figure 2.3). In the survey, developers were presented with the set of files (branch information and absolute file names) from an approximated change set. Those files were checked-in by them to the version repository. The task of the developers was to

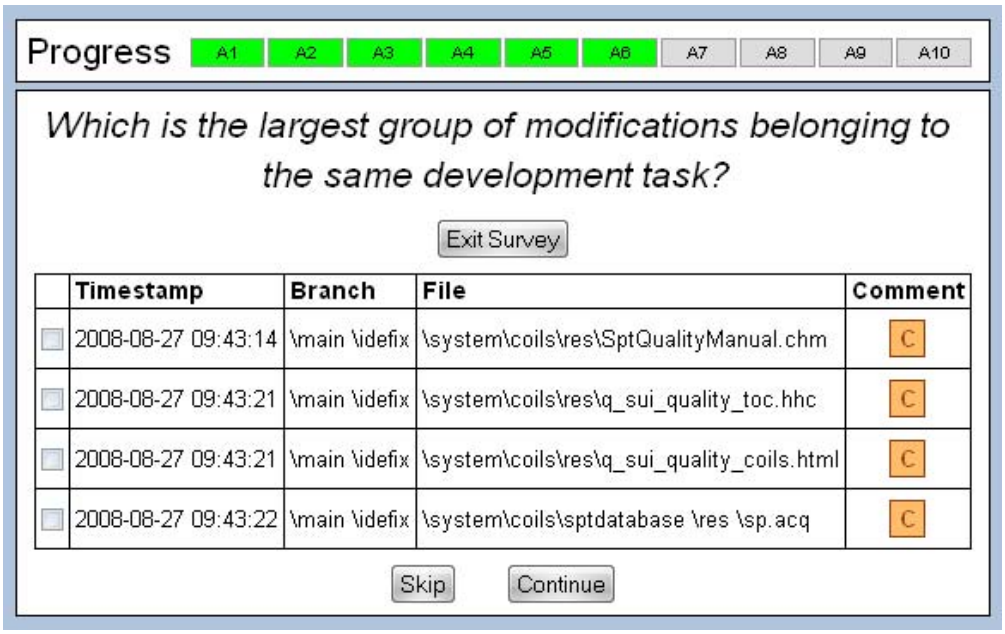


Figure 2.3: Screenshot of the on-line survey to evaluate change sets

recall which files were indeed modified for the same development task and mark them using the check-box on the left most column for each row. In order to provide some assistance to the developers in recalling the change sets, we provided the timestamps for when the files were checked-in to the system and a commit message (comment column) if available that summarized the rationale behind the change. Recall that in our study environment, such comments were extremely rare and hence were seldom provided in the survey. We also ensured that the change sets presented to the developers were from the recent past such that developers could more reliably draw from their memory to identify the change sets.

At the beginning of the survey, the survey participants were provided with a description of the purpose of the survey and explicit instructions including a demonstration on how to use the on-line form. In addition, we also made clear to the participants our definition of a change set and how the approximated change sets were collected. However, in order to reduce chances of bias on part of the respondents, we did not indicate the time interval used to approximate the presents change set for evaluation. The participants were then presented a series of approximated change sets one after another and they were required to mark the largest set of files that they recall being changed together for a specific task. The presented change sets were randomly selected from a single pool that contained change sets committed by the respective participant and approximated using the different time intervals: 200 seconds, 1 hour, 1 day, 1 week, and 1 month. The change sets from the pool were randomly presented one after

Table 2.2: Precision estimated with help of developers

Time interval	Precision (in %)			#ACS*	
(δ)	Max.	Min.	Avg.	Analyzed	Skipped
200 seconds	100	50	91	19	3
1 hour	100	33	91	15	4
1 day	100	40	78	21	4
1 week	100	6	66	14	7
1 month	100	2	36	6	8

*ACS stands for Approximated Change Sets

another until 10 approximated change sets were evaluated. Respondents were made aware of the option to skip questions pertaining to change sets that they were not certain about and also that they could exit the survey before answering 10 questions. The progress bar on top of the survey indicated the number of answered questions (Figure 2.3).

Considering that the data in the version repository spanned across nine years, the main criterion to select developers and invite them to participate in our survey was that they are currently working in the company. We narrowed the list of developers to invite by looking at their level of activity, which was measured as the number of check-ins made in the year 2009 (from January to May). We identified 25 developers with the highest number of check-ins ranging from 7,300 to 441 during the selected time period (see Figure 2.4). In the figure, the *x-axis* represents the 25 anonymized developers and the *y-axis* represents the number of check-ins made by them. The top-4 developers immediately stand-out in the figure owing to the marked high number of check-ins made by them. An investigation into the role of these four developers revealed that they were often given the task to merge one version branch to another that resulted in many check-ins. Since such merges are of little interest to our goal of approximating change sets, we chose to approach the next ten most active developers indicated in the figure.

The data recorded from the survey were the number of files presented to the developer as an approximated change set and the number of files selected by the developer. Eight of the ten invited developers responded to our survey. All but one developer who took the survey analyzed ten approximated change sets. The only exception analyzed five sets before exiting the survey. Altogether, developers evaluated 75 approximated change sets and skipped 26.

Results from the survey are presented in Table 2.2. We compute the accuracy of our approximated change sets as the ratio of files checked by developers in a presented change set and the total number of files in the change set. In terms of information retrieval, this is equivalent to measuring the precision. For each time interval, we note the minimum, maximum, and average precision and report them in Table 2.2. Additionally, we also note the number of analyzes and skipped change sets for each time interval. The following observations can be made from the table:

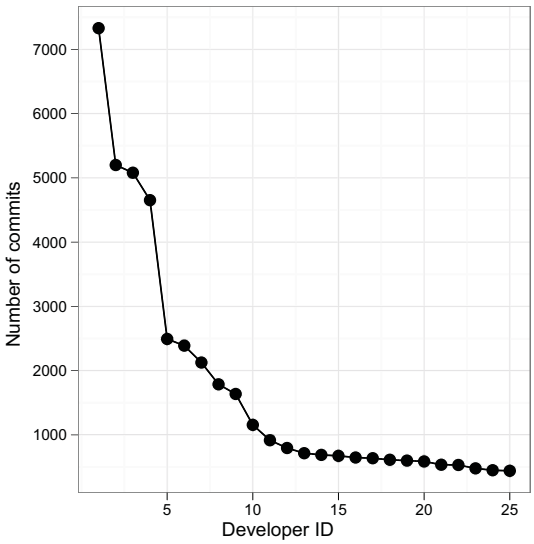


Figure 2.4: Number of check-ins made by most active developers at Philips Healthcare. We invited Developers 5–14 to evaluate the approximated change sets.

- The change set approximations inferred from time intervals 200 seconds and 1 hour have high precision. Average precision decreases with increasing intervals.

- Given that the precision for both time intervals 200 seconds and 1 hour are the same, change sets inferred using the hour interval are more useful because they capture a larger set of files relevant to the task.

- It is noteworthy that despite the low average precision for the 1 month time interval, in selected cases change sets are still correctly approximated (max. precision observed was 100%). This suggests that the life cycles of some of the development tasks are indeed very long.

At this stage, it is important to note the purpose of solely focussing on precision values for this evaluation. Recall that architects at Philips Healthcare are interested in reducing cases where changes to accomplish a task are made to files across different sub-systems. So it is more important that precision of the approximated change sets is high in comparison to higher values of recall. Approximated change sets with higher precision allows architects to reliably identify cross sub-system changes that can then be examined. Increasing recall will pollute the change sets with changes made for several different tasks and be detrimental to any further analysis. To add to this, to reliably compute recall values, it is important to know the complete

```
            Unique ID | <POSTLIST_NAME>nly95872_RFAmpSim_20060103T150114
     Stream information | <DEVSTREAM>FEMAIN
              Developer | <USER>Anna
                   Date | <DATE_DD_MMM_YYYY>3 JAN 2006
Is it a problem report? | <PR_SECTION>Y
  Problem report number | <SOLVED_PR>MR00035599
Rationale behind change | <REASON_TEXT>Improve simulation of the RF-Amplifier
                        | <CODING_STANDARD>N
                        | <PNS_SAR>N
     Developers playing | <BBLOCK_OWNER>James
          a role in the | <REVIEWER>Robert
         review process | <TEAMLEADER>David
                        | <DOC_SECTION>N
Changed files submitted | <POSTLIST_FILE>\path.to.file.one\BDRFAmplifierNorf.h@@\main\femain\5
             for review | <POSTLIST_FILE>\path.to.file.two\bdtransmittercsinterfaces.hcf@@\main\3
                        | <POSTLIST_FILE>\path.to.file.three\BDRFAmplifierNorf.cpp@@\main\femain\5
                        | <TEST_SECTION>Y
                        | <TEST_DONE>@OTM6
 Last versions of files | <PREVIOUSLY_CONSOLIDATED_FILE>\path.to.file.one\BDRFAmplifierNorf.h@@\main\femain\2
   used to build system | <PREVIOUSLY_CONSOLIDATED_FILE>\path.to.file.two\bdtransmittercsinterfaces.hcf@@\main\1
                        | <PREVIOUSLY_CONSOLIDATED_FILE>\path.to.file.three\BDRFAmplifierNorf.cpp@@\main\femain\4
                        | <ACTION_POSTLIST>20060103T150117
                        | <ACTION_POST>20060103T150901
                        | <ACTION_TAKE>20060103T210839
                        | <ACTION_CONSOLIDATE>20060109T161257
                        | <SWID>29
```

Figure 2.5: A sample postlist presenting a change set.

change sets in advance and accurately, which is exactly the purpose of our case study in the first place.

2.5.2 Postlists

The second option available to us to evaluate the accuracy of the approximated change sets are postlists. They are messages manually created by developers to notify the system integrator that a certain set of files have been added or modified and are ready to be integrated into the system (see Figure 2.1). After adequate testing, the changes are approved and consolidated into the latest version of the system.

A sample postlist used at Philips Healthcare is presented in Figure 2.5. Lines that add no value in explaining the postlist are grayed out in the figure. At the top, the postlist contains basic meta-data such as a unique identifier, the name of the developer, date of the message, and whether there is problem report (akin to a bug report) filed that links the changes to the problem. In the meta-data, the developer are also encouraged to submit an explanation of the changes. It then lists the people in the team who will be involved in reviewing the changes. Next, the files that have been submitted for review are listed along with the complete absolute path, branch information, and the new version numbers of the files. Lastly, the postlist also

presents the existing version numbers of the files that are already consolidated in the system.

We would ideally expect these postlists to be the best resource to use to approximate change sets given their content. However, on inspection we found that these postlists were not change set specific, noisy and unreliable for the purpose. They often contained changes made to files for more than one task and provided no means to map the changed files to the respective tasks. Further, change sets often spread across several postlists similar to the culture of committing code iteratively to the version repository. Many postlists also contained generic reasons which could not be reliably mapped to any development task. Lastly, postlists do not capture the range of changes made to files but only the snapshots of the versions that are ready to be integrated. Hence, the change history in the postlists is likely to be incomplete. For instance, in Figure 2.5, version 5 of file `BDRFAmplifierNorf.h` has been submitted in the postlist, but the latest consolidated version of the same file is 2. These reasons ruled out the possibility of an extensive evaluation of the accuracy of approximated change sets by leveraging postlists.

However, we made an attempt to use some carefully selected postlists to evaluate the change sets. Keeping in mind the above cited challenges with postlists, we made sure that the selected postlists were indeed specific to one change set only. In order to select these change-set specific postlists, we collected a random sample of 100 postlists and discarded those that met one of the following criteria and replaced it with another postlist deemed fit (to keep the total number of postlists 100 for the evaluation):

- The reason field that the files in the postlist were modified because of multiple development tasks For instance: *"Fixed two problems: [...]"* and *"Add [...], Remove [...], Add [...]"*)
- The reason field indicated that a postlist came from a division of the company where they almost always group more change sets into a single postlist to reduce interaction costs. For instance: *"Cleveland delivery"*
- The name of the postlist or the reason field pointed out that the postlist was created only because of a merge activity. For instance: *"Merge of [...] to branch [...]"*
- The description in the reason field was too abstract. For instance: *"Update"*. Abstract reasons may be an indicator that a developer worked on multiple tasks and therefore he could not provide a single concrete reason.
- The postlist was created to submit only a partial solution of a development task. We knew that this was the case if the reason field contained descriptions like: *"First part for [...]"* and *"part 2"*.

While we meticulously examined the postlists and arrived at the final set of 100 for our evaluation, we realize that in a handful of cases postlists may refer to more than one change set. But this risk is minimal given our familiarity and experience with the system.

Recall that one would often see a difference greater than one between version numbers of a changed file and its previously consolidated version. We accounted for these missing file

Table 2.3: Precision estimated using postlists

Time interval	Precision (in %)		
(δ)	Max.	Min.	Avg.
200 seconds	100	50	93
1 hour	100	20	89
1 day	100	1	69
1 week	100	<1	31
1 month	95	<1	8

Table 2.4: Recall estimated using postlists

Time interval	Recall (in %)		
(δ)	Max.	Min.	Avg.
200 seconds	100	20	74
1 hour	100	20	84
1 day	100	22	92
1 week	100	24	94
1 month	100	25	94

versions by interpolating them in the postlists assuming that all intermediate changes to the file were related to the same development task.

Using each selected postlist, we examined the set of added or changed files and their version numbers. For each such version of a file, we queried which approximated change set contains that version. We considered the approximated change set with the largest overlap of file versions as the dominant change set and evaluated it.

By comparing the (dominant) approximated change set and the list of file versions from the postlist, we measured precision and recall values as measures of accuracy (presented in Tables 2.3 and 2.4 respectively). Precision was computed as the ratio of the number of file versions common to the two change sets and the number of file versions in the approximated change set. Recall was computed as the ratio of the number of file versions common to the two change sets and the number of file versions in the postlist. The following observations can be made from the tables:

- The precision values suggest that on average the approximated change sets with time intervals 200 seconds and 1 hour are precise (with nearly 90% or more).
- The 1 day time interval gives an acceptable level of precision. However, increasing time intervals seem to result in a substantial decrease in precision values.
- From the recall values in Table 2.4, we see that the 200 second time interval on average

missed one out of four files belong to a change set. Given the time needed to execute development tasks at the company, such high recall is a surprising result.

- The value of recall increased further with increasing time intervals, but this occurred at a cost of falling precision.

2.6 Discussion

In the previous section, we evaluated the approximated change sets using two different ways: cross-checking with developers and using postlists. Both evaluation measures have provided similar results.

We used five discreet time intervals for the change set approximation method which yielded five sets of approximated change sets. During the evaluation of the approximated change sets, it turns out that accuracy is optimal when the time interval is set to 1 hour. We expect, however, that using more granular time intervals to approximate change sets may result in more fine tuned results, but this remains to be explored in our study environment. Our motivation to use only five discreet values was to limit the effort spent on the evaluations. We would like to point out that the 1 hour interval is not a recommended universal time interval to approximate changes. Instead, we encourage researchers working in environments similar to ours to conduct a similar exercise to establish an optimal time interval for use.

2.7 Threats to Validity

When evaluating the accuracy of the approximated change sets, we assumed that developers have a good recall of their own change sets and that the postlists selected represent actual change sets. The results from our evaluation could have been affected if developers remembered their change sets incorrectly or the selected postlist did not represent single and complete change sets. We have done our best to mitigate such effects. We presented all meta-data available on the change sets approximated to the developers so that they could recall what belonged to the same development task. Postlists were carefully analyzed to select those that represent complete change sets for our use. The similarity in our results from both evaluation measures suggest that they are valid.

Our method approximated change sets with an implicit assumption that a single developer completed a development task. However, in cases where more developers worked on the task, we could recover only parts of the change set. In our study environment, however, we found no means to relate the modifications of developers who worked on the same development task. Such cases may effect the recall of our approximations. Architects at our study environment were fully aware of this limitation and were supportive of our approach to approximate change sets because precision of the approximations was far more important to them for further analysis than approximating complete change sets.

2.8 Conclusion and consequences

It is important to identify change sets for many evolution-related studies. Change sets (that are different from transactions) are used, for instance, to assess the evolvability of the architecture. Change sets need to be approximated if they are not available directly from the version control system applied. To do so properly, one needs to carefully take into account the development practices used.

We have conducted a case study at Philips Healthcare to approximate change sets from their version repository. Approximations had to be made amongst several constraints at the company, most severe one of them being lack of meta-data. Our approach comprised using only developer information and check-in timestamps of files to recover change sets. Five time intervals were experimented with to approximate the change sets, whose evaluations using developers and postlists resulted in reasonably high accuracy. The optimal time interval found in the study environment was 1 hour. Our evaluation has helped us choose the right set of change sets approximated from the version archives of the company, which has already performed successfully in previous evolution-related studies [Vanya et al., 2010].

Our research has shown that it is possible to approximate change sets even in environments where meta-data is limited. Furthermore, differences in accuracy of the approximations across different time intervals emphasizes the importance of evaluation. As mentioned earlier, decisions based on these change sets can be costly and have far-fetching consequences. Hence, an evaluation of the data is vital for further reliable results.

Assessing Software System Decompositions with Evolutionary Clusters

3

The way in which a software system is decomposed influences the evolvability of that system. The decomposition, e.g. subsystem decomposition, is mostly assessed by looking at the static (include, call) relations between the parts of that decomposition. In the literature history information is also taken into account to assess the decomposition. In this chapter we describe our history-based approach to (automatically) assess the extent in which a certain decomposition allows its parts to evolve independently. We use the assumption that software entities which changed often together in the past are likely to be modified together in the near future as well. Hence, the elements of such a set should in principle belong to the same part of the system's decomposition. Our approach, therefore, identifies sets of co-changing software entities, where each set has elements from more than one part of the decomposition. By doing so, we execute Step 3 of the process described in Section 1.7 to help the software architect. We illustrate our approach with a case study of a large software system that evolved during more than a decade, and has over 7 million lines of code.

3.1 Introduction

Software degrades as it evolves unless effort is invested to keep the structure of the software clean [Lehman et al., 1997]. Even if we carefully design a system such that its parts can be independently developed and maintained, this structure will erode over time. Our systems become legacy systems that are difficult to comprehend and maintain. For example, software entities may have ended up in a part of the system decomposition where they do not really belong. Since the decomposition of the software system is often used to assign development or test tasks, a degraded decomposition hampers evolution. Re-arranging or improving the decomposition of a legacy software system and assigning separate, independent development life cycles to the identified parts of the decomposition may help mitigate maintenance problems.

Such an improved decomposition allows one to exploit the benefits of parallel development, like shorter time to market. Since a modification in a given part of the decomposition has a known scope, tests can be applied in a more focused manner and consequently can result in a software product with better quality attributes.

Parts of this chapter has been published as:

Vanya, A., Hofland, L., Klusener, S., van der Laar, P., and van Vliet, H. Assessing Software Archives with Evolutionary Clusters. In 16^{th} *International Conference on Program Comprehension* (ICPC '08), pages 192–201, IEEE Computer Society, 2008.

Although many other possibilities exist, we assess the decomposition by considering the *history* of the software system. History information on a software system tells us how, when and why a given software entity was modified. This information typically spans many years and is stored in version management systems, like ClearCase or CVS. We address the following research question in this chapter: How can the available history information help experts assess a decomposition of the software system from the independent development point of view?

Our approach to deal with the posed research question (1) identifies a structure of evolutionary clusters (sets of software entities which changed frequently together in the past), (2) relates the evolutionary clusters to the current decomposition, (3) indicates unwanted couplings which need to be resolved to arrive to the desired decomposition, (4) proposes an ordering of the indicated problems based on their severity.

Similar to Gîrba et al. [2004] we use the Yesterday's Weather assumption: if software entities were modified together in the past, they will most likely be modified together in the near future as well. We illustrate our approach with a case study of a large software system containing more than 9 million lines of code, developed over a period of more than a decade.

Previous works identifying groups of co-changing files based on historical information relate files if they were checked in together nearly at the same time or if the check-ins of those files formed a coherent sequence. We take these ideas one step further by generalizing to the essence: we consider two software entities to have a co-change relationship if they were modified together due to the same motivation. In practice the connections between modifications of software entities and the underlying motivations are often not captured and we need approximations to recover them. One of the approximations is to relate check-ins which happened nearly at the same time, but other approximations might also exist as well depending on the applied development process.

Our approach not only identifies clusters of co-changing software entities and the relationships of those clusters but our approach also maps these clusters to the current decomposition of the software system and automatically identifies an ordered list of potential unwanted couplings. The case study shows that we can effectively point out unwanted couplings.

In Section 3.2 we describe how our work relates to the state-of-the-art. Section 3.3 provides an overview of our approach and introduces the used terminology. In Section 3.4 we describe our approach step by step. Section 3.5 details the application of our approach in a case study. In Section 3.6 we discuss the lessons learned as well as possible directions for future research. In Section 3.7 we describe our conclusions.

3.2 Related Work

3.2.1 History-Related Research

History information about how a software system evolved over time is widely used in the literature, with different purposes. On the one hand, some analyze the history of individual software entities or their relationships to identify trends and patterns. They do so with the purpose to better understand the system and / or to classify its entities. On the other hand, a

significant part of the literature using history information identifies modification similarities or couplings among software entities (files, classes, packages).

From the first category, Greevy and Ducasse [2005] analyze how the number of features a given class implements changes over time. Their goal is to show how features evolve with respect to their implementations. Lungu and Lanza [2007] visualize the evolution of static dependencies, like class inheritance or method invocation between modules, to enrich software exploration. Their tool makes it possible to classify relations based on the underlying patterns of evolution. Others analyze how the ownership of files [Gîrba et al., 2005] or how the lines in a file [Voinea et al., 2005] have changed over time. Both works aim at a better understanding of past changes and developer interactions.

In this strand of research, the history information is not clustered. For instance, if the history information is at the level of classes, it is not grouped to obtain knowledge at the package or subsystem level.

As for the second category, Gall et al. [2003] uncover logical couplings of classes to pinpoint some of the possible unwanted couplings. In contrast to our work, they relate pairs of software entities only. Zimmerman et al. [2005] use the identified modification similarities among fine-grained software entities (functions, variables) to warn developers for inconsistent changes. Fischer et al. [2005] use versioning information of closely related software products to identify the platform of a future product family. Included in the second category, there are works which cluster the history information of software entities to predict future changes and / or identify dependencies. Antoniol et al. [2005] relate files if they were checked-in nearly at the same time in a sequence for a couple of times. Gîrba et al. [2007] apply concept analysis to group software entities if their properties (size, number of contained methods) changed along a similar pattern over time. As described, the resulting lattice of clusters are huge and often contains redundant or highly overlapping nodes. Therefore it is difficult to use the resulting lattice to assess the decomposition of the software system; the co-change information for a set of software entities are distributed in the lattice.

3.2.2 Recovering Software Components

Many papers address the issue of isolating parts of a legacy software system that can evolve independently. Mehta and Heineman [2002] for example identify fine-grained components of a legacy system. In order to reach their goal, they execute regression tests related to a given feature.

Reverse engineering subsystems and other higher level entities of a legacy system based on the similarities between identifier names and comments is the goal of Kuhn et al. [2005]. They use Latent Semantic Indexing to cluster parts of the legacy system having the same meaning. A similar approach is used by Anquetil et al. [1999] but they rely on the names of source files only.

Schwanke [1991] uses static information (call relations, returns, common variables) among procedures to calculate their similarities. Based on that, they execute hierarchical, agglomerative clustering to identify the modules of the software system. Wiggerts [1997] provides a clas-

sification of clustering algorithms and a description of how they can be applied to re-structure a legacy system. Maqbool and Babri [2007] review and characterize clustering algorithms which might be used for recovering architectural components. They also assign semantics to the parameters of the clustering algorithms.

Mitchell and Mancoridis [2007] identify modules by creating an initial partition of source files and by applying optimization algorithms to decrease the dependencies between the modules and increase dependencies inside the modules. The dependencies considered are static relationships.

Wierda et al. [2006] present a case study to recover subsystems of a legacy software by clustering static relations of classes using different versions of the system. They show that using different versions of the system as an input can improve the accuracy of the results.

History information was also used to recover high-level abstractions, like subsystems. Beyer and Noack [2005] propose a two dimensional layout of files such that often co-changed groups of files are plotted closer than groups of files which were rarely checked-in together. The fact that co-change information is only visualized makes it difficult to determine the scope of the unwanted coupling and their relative importance.

In general, approaches which propose a completely new decomposition for the software system might be difficult to implement and are costly. Smaller, but effective improvements are often preferred because new releases of the software can still be delivered to the market on time to stay in competition. Also, the decompositions of existing software systems were developed for good reasons and they should be used to determine the new decomposition rather than thrown away. In [Beyer and Noack, 2005], for instance, we can see that the software systems analyzed have a rather good decomposition.

3.3 Approach Overview

In this section we introduce the terminology used in this chapter and give an overview of our approach. A more detailed, step-by-step description is the topic of the following section.

We assume that an initial idea of how to decompose the software system into independently modifiable parts is available and we refer to it as the *initial decomposition*. This initial decomposition may come from the documentation, expert knowledge, or it can be the current decomposition of the software system.

To validate the initial decomposition from the history perspective, our approach may use any type of input capturing which files were modified together in the past because of the same motivation. This motivation can be, for example, the tasks of the developer to modify or to create features. While describing our approach in general, we also specify how versioning information residing in Version Management Systems can be used as a specific input for our approach. Versioning information is available most of the time. It can be considered as meta–data of file modifications, like who altered a file, when, how and why. Although advanced Version Management Systems are capable of fine–grained versioning, our approach relies only on the basic functionalities of early, but still widely used, Version Management Systems like CVS.

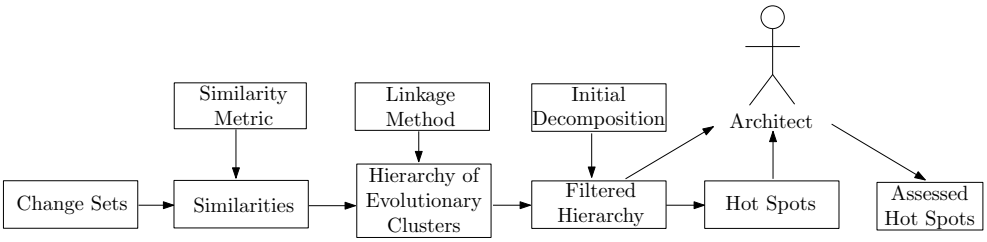


Figure 3.1: The evolutionary cluster approach

Eventually, we are interested in unwanted couplings concerning the initial decomposition, i.e., different decomposition parts that were intended to evolve independently, while in practice they co-evolved. These unwanted couplings are the result of an assessment process, in which the architects are asked to evaluate a set of *hot spots*. A hot spot indicates that a group of files from one part of the initial decomposition were often changed together with a group from another part. A hot spot does not need to become an unwanted coupling. For instance, the architect may know and accept that different parts of the initial decomposition co-evolve. Also, evolution is not the only criterion to be considered to separate the parts of the decomposition.

To fully achieve the desired decomposition of the software system from the evolution perspective, the unwanted couplings selected by the above process have to be addressed. Our approach, however, stops at assessing the initial decomposition. Describing possible solutions to address an unwanted coupling is not within the scope of this chapter.

Figure 3.1 depicts the steps of our approach. All steps of our approach are further elaborated in the next section. When doing so, we also describe how we designed the steps and what the considered alternatives were.

3.4 From Change Sets to evolutionary clusters

3.4.1 Change Sets and Their Approximations

The construction of our evolutionary clusters starts with the notion of a *change set* [Estublier et al., 2005b]. A change set is a set of modifications that are assumed to have a common motivation. The co-evolution of files can now be measured by their common change sets. If the modifications of files occur in the same change set then those files co-changed once.

Figure 3.2 on the next page gives an example of how change sets are related to files ($F_i, i \in [1..4]$) and motivations ($M_j, j \in [1..3]$). In the figure, each file symbol denotes a modification of a single file and rectangles denote change sets. The example shows that for instance F_1 and F_2 co-changed twice, as they were both modified because of M_1 and M_2 but not because of M_3.

Identifying change sets is not obvious, there are various ways to do this. Some change management systems, like Unified Change Management (UCM), provide change sets directly,

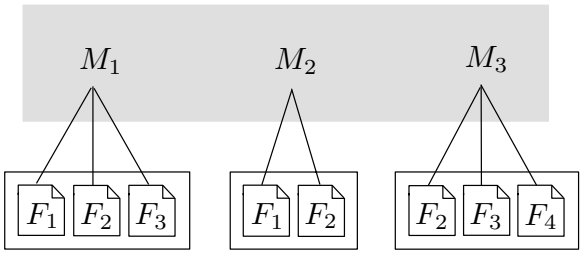


Figure 3.2: Change Sets

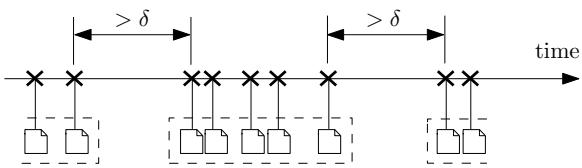


Figure 3.3: Change Set approximations

see also [Estublier et al., 2005b]. In most cases, however, change sets must be approximated from the available data.

In order to construct our approximated change sets, in this chapter we group modifications, based on developer and time. We sort all modifications of one developer, based on their time stamp. Consecutive modifications are put in the same change set, unless their difference in time is larger than δ. We assume some given δ, typically a couple of minutes. Figure 3.3 illustrates the relations between check-ins and change sets. This approximation of change sets is similar to the notion of a transaction by Zimmermann et al. [2004]. Note that we leave the rationale of each change set implicit, whereas Zimmermann applies a filter based on some rationale constructed from the comments of the modifications.

Approximations introduce false positives and false negatives by their nature. In our case, one developer working on different tasks in parallel may check in files at the same point in time without a common motivation, which would introduce a false positive. On the other hand, a developer may take a cup of coffee during a sequence of related modifications. This will result in a false negative as the sequence gets interrupted and the modifications will end up in two different change sets.

We do recognize that the way we obtain our change sets influences the construction of the evolutionary clusters. An in-depth discussion of this issue, in particular the dependence on the actual software process of an organization and how a project administration could be used as source of information, is outside the scope of this chapter.

3.4.2 The Similarity Metric

Having constructed the change sets, we can now compute the similarity coefficients for each pair of files. This results in a similarity matrix. Given a pair of file, a similarity coefficient close to 1 reflects that the files often co-changed, whereas a similarity coefficient close to 0 means that the files hardly co-changed. The metric we use to identify similarities is the Jaccard coefficient [Abreu et al., 2006]. Given the number of times when two files (f_1 and f_2) were modified together (a_{11}) and separately (a_{10}, a_{01}), the Jaccard similarity coefficient is computed as follows:

$$J_{(f_1,f_2)} = \frac{a_{11}}{a_{11} + a_{10} + a_{01}}$$

One of the main reasons why the Jaccard similarity coefficient is widely used in the literature and also by us lies in its intuitive semantics. In our case, the coefficient is the estimated probability that two files will co-change if one of them gets changed, see also [Maqbool and Babri, 2007, Abreu et al., 2006]. Also, the Jaccard similarity coefficient is symmetric which is needed to execute the next step of our approach (see Section 3.4.3).

3.4.3 Hierarchy of evolutionary clusters

The main contribution of this chapter is how evolutionary clusters are constructed from historical information. The Jaccard formula gives us the similarity coefficient between two files, the next step is to aggregate file-level similarity to the similarity between sets of files. Given two sets of files, there are various ways to aggregate the file-level similarity to the level of the sets. Those alternative ways of aggregation are also called *linkage methods*. For our purposes we use the *average linkage*. The average linkage value of two sets A and B is the average of the Jaccard similarities J_{XY} where X is a file from A and Y is a file from B.

We start with sets containing one file only and we join sets until only one set remains. In every step, the two sets with the highest linkage value are joined. This algorithm is known as Agglomerative Hierarchical Cluster Analysis (AHCA) [Tryan, 1939, Wiggerts, 1997]. The resulting hierarchy of clusters is a binary tree, also known as *dendrogram*.

To illustrate the clustering algorithm, let us consider an example, involving four entities and six change sets, as follows:

- The first change set includes files C_1, C_2, C_3 and C_4,
- The second change set includes file C_1,
- The third change set includes files C_1 and C_4,
- The fourth change set includes files C_2, C_3 and C_4,
- The fifth change set includes file C_4,
- The sixth change set includes files C_2 and C_4

	C_1	C_2	C_3	C_4
C_1		$1/(1+2+2) = 0.2$	$1/(1+2+1) = 0.25$	$2/(2+1+3) = 0.33$
C_2			$2/(2+1+0) = 0.66$	$3/(3+0+2) = 0.6$
C_3				$2/(2+0+3) = 0.4$
C_4				

Table 3.1: Jaccard similarities

alv($\{C_1\}$, $\{C_2, C_3\}$) = (alv($\{C_1\}$, $\{C_2\}$) + alv($\{C_1\}$, $\{C_3\}$))/2 = $(0.2 + 0.25)/2 = 0.225$
alv($\{C_1\}$, $\{C_4\}$) = 0.33
alv($\{C_2, C_3\}$, $\{C_4\}$) = (alv($\{C_2\}$, $\{C_4\}$) + alv($\{C_3\}$, $\{C_4\}$))/2 = $(0.6 + 0.4)/2 = 0.5$

Table 3.2: Average linkage values (alv) for $\{C_1, \{C_2, C_3\}, C_4\}$

As a first step, we measure the Jaccard similarity between each pair of files. For the above example, files C_1 and C_4 changed twice together (in the first and third change set), while C_1 changed once without C_4 (in the second change set), and C_4 changed three times without C_1 (in the fourth, fifth and sixth change set). So the Jaccard similarity is $2/(2 + 1 + 3) = 0.33$. The complete set of Jaccard similarities is given in Table 3.1.

The next step is to iteratively cluster files. For our example, we will cluster C_2 and C_3, since their Jaccard similarity is 0.66. In the next step, we have to consider the three average linkage values given in Table 3.2. So we next cluster $\{C_2, C_3\}$ and C_4, with average linkage value 0.5. And in the final step we cluster all four files, with a resulting average linkage value of 0.251.

There are many ways to aggregate file-level similarities to the level of sets of files. One can use, for example, the minimum of all pairwise coefficients (also known as *single linkage*), the average (*average linkage*) and the maximum (*complete linkage*). We have chosen the average linkage method since the other linkage methods result in a cluster hierarchy being very sensitive to even slight modifications to the strength of file similarities. A relative stability of results is needed if we want to base longer term decisions on the identified clusters. A relative stability of results is important if we want to base longer term architectural decisions on the identified clusters.

Furthermore, single and complete linkage methods represent two extremes. On the one hand, single linkage method is good at maximizing the cohesion inside clusters, but it tends to leave strongly related clusters separated. On the other hand, the usage of complete linkage results in clusters having low coupling, but also low cohesion. To overcome these problems we chose the average linkage.

3.4.4 Evolutionary clusters

In the resulting dendrogram the nodes are the identified evolutionary clusters. The vertical alignment of the evolutionary clusters in the dendrogram indicates the cohesion level: the nearer an evolutionary cluster is to the root of the dendrogram (the top of Figure 3.4), the less frequently the contained files changed together.

An evolutionary cluster is practically a set of files. Such an evolutionary cluster may be an unwanted coupling in the initial decomposition if it contains frequently co-changing files coming from different parts of the initial decomposition. The following properties of the evolutionary cluster might indicate the relevancy of the identified potential unwanted coupling:

- The number of the frequent co-changes in the evolutionary cluster relating files from different parts of the initial decomposition
- The frequency of the co-changes between files of the evolutionary cluster
- The number of the parts where files of the evolutionary cluster belong to

Ideally, such border crossing evolutionary clusters should not exist at all. At the minimum they should be known.

We are not only interested in sets of files which were co-changed very often, but also in others which co-changed less frequently. This makes it possible to perform a finer tuned assessment of the initial decomposition.

3.4.5 The Filtered Hierarchy

Evolutionary clusters containing files from the same part of the initial decomposition do indicate the correctness of that decomposition. To identify unwanted couplings they are of less interest, and therefore we call them *trivial*. We may remove these trivial evolutionary clusters from our dendrogram, leading to a pruned one. In the sequel we will focus on *non-trivial* evolutionary clusters, clusters that contain files from different parts of the initial decomposition.

3.4.6 Identifying Hot Spots

Even when the initial decomposition is created by people who have a deep knowledge of the software system, it is very well possible that there are modification similarities between the parts that may hamper evolution. As described in sections 3.4.4 and 3.4.5, such potential unwanted couplings are indicated by non-trivial evolutionary clusters.

By pruning the dendrogram in the previous step, we not only identify evolutionary clusters which can be indicators of unwanted couplings in the initial decomposition, but we also provide a structure for the evolutionary clusters. The leaves of the tree indicate the hot spots of potential unwanted couplings and by moving toward the root of the tree (top of Figure 3.4) we get more contextual information (the scope of the unwanted coupling), more files, but less cohesion. Also, between leaves there are typically differences in cohesion level. The cohesion level

specifies how frequently a set of files co-changed. By sorting the hot spots in decreasing order of their cohesion level, it is possible to give an ordered list of the hot spots.

The ordered list of hot spots, which is the output of this step, is a suggestion to experts and developers as to which potential unwanted couplings should be addressed first. On the other hand, experts are free to pick any of the hot spots they want to start with.

3.4.7 Evaluation of Hot Spots

Only potential unwanted couplings are indicated by the elements of the ordered list, the output of the previous step. Experts may not consider all the strong modification similarities crossing the borders of decomposition parts unwanted. One of the reasons is that often when the initial decomposition is created, other requirements are also taken into account, like the execution architecture. In those cases typically a trade-off is made between localizing modifications and other requirements.

When a potential unwanted coupling is found to be a real risk for the future evolution of the software system, it has to be resolved. Our approach stops when the initial decomposition is assessed. By resolving the unwanted couplings we can get to the desired decomposition of the software system from the evolution point of view.

3.5 Case Study

We have applied our approach to the software system of Philips Healthcare MRI to

- show that our approach works in practice
- elaborate how effective our approach is
- generate and answer practical questions related to the application of our approach
- describe an example implementation

3.5.1 Our Approach at Work

In this section, for every step of our approach we describe our implementation, what kind of practical issues we faced and how we addressed those issues.

Recovering Change Sets

The used version management system from which we extracted versioning data was ClearCase. Using a ClearCase script, we queried what was checked-in, when and by whom. We extracted check-in related information from the last 6 years of development. The extraction resulted in a script file which we used to transfer check-in information into a table in our database.

At this point we realized that we faced a serious scalability issue. Among the 34,000 files which were checked-in the possible number of relations is more than 1 billion. This is more

than what computing environments can handle. We recognized, however, that working on the file level would be also too detailed. Fortunately, our approach can work on higher abstractions than files.

To overcome the scalability issue and remain focused, instead of files we considered the directories representing the next level of abstraction in the file hierarchy. This way, we consider the check-in of a file as the check-in of a directory. Massaging the table accordingly resulted in check-ins of more than 500 directories. These directories are the implementations of *building blocks* [van der Linden and Müller, 1995], [Jaring et al., 2004].

Another reason why we considered building blocks is that architects were first interested in the unwanted couplings at the building block level. Also, according to the architects each building block represent a semantically coherent set of files.

To identify change sets, we implemented a sliding window algorithm as a stored procedure and applied the stored procedure on the previously extracted check-in meta data. As a result, we identified 73,851 change sets which we put into a new table.

The analysis of the identified change sets revealed that there were very huge change sets, containing even more than 1000 check-ins. During the discussions with developers it turned out that those huge change sets were only the side effects of merging activities. In order to get rid of this kind of noise we discarded all change sets containing more than 100 check-ins. The threshold of 100 was advised by the developers.

Similarities and the Cluster Hierarchy

In-line with the step described in Section 4.2 we developed another stored procedure to identify modification similarities among building blocks. For that we used the formula from Section 3.4.2. The result is a table of similarities.

To execute the step detailed in Section 4.3 we transfered the similarity table into a statistical package. Using that package we executed the AHCA on the similarity table. The resulting dendrogram, before the reduction, is shown in Figure 3.4 (see next page).

Reduction and Hot Spot Identification

For the initial decomposition we took the subsystem decomposition as point of departure. Identifying subsystems was easy, because the building blocks of the software system are organized into a hierarchical structure and this structure reflected the subsystem decomposition.

To implement the step described in Section 4.5, first we reduced the dendrogram such that it contained non-trivial evolutionary clusters only. By looking at the leaves (hot spots) of the reduced dendrogram we identified the top 10 most interesting evolutionary clusters which may hinder independent evolution of the subsystems. The identified top 10 clusters are the first 10 elements of the ordered list described in Section 4.6, that is, the 10 non-trivial evolutionary clusters having the highest cohesion.

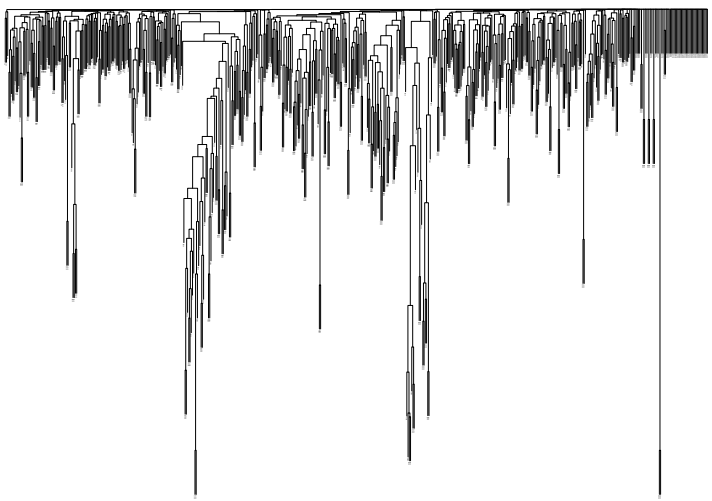


Figure 3.4: The complete dendrogram

Evaluation of Hot Spots

We gave the list of the top 10 hot spots to the architects and asked if the identified evolutionary clusters did indicate unwanted couplings, i.e. real obstacles to the evolution of the subsystems. We also asked whether the indicated unwanted couplings were known, or whether our approach taught the architects about new problems. We refer to this classification of hot spots with the following symbols:

- $\times$: non-issues (false positives)
- $\sim$: agreed-upon but already known unwanted couplings
- ! : agreed-upon yet unknown unwanted couplings

Using this annotation our top 10 list is presented in Table 3.3. The architects did not find major false negatives.

Table 3.3: Top 10 Hot Spots

Rank	1	2	3	4	5	6	7	8	9	10
Type	!	$\sim$	!	!	$\sim$	!	$\sim$	$\sim$	$\times$	$\sim$

All the agreed-upon unwanted couplings which were unknown before indicated surprising relationships between subsystems. In one of those cases (rank 3) our approach identified co–

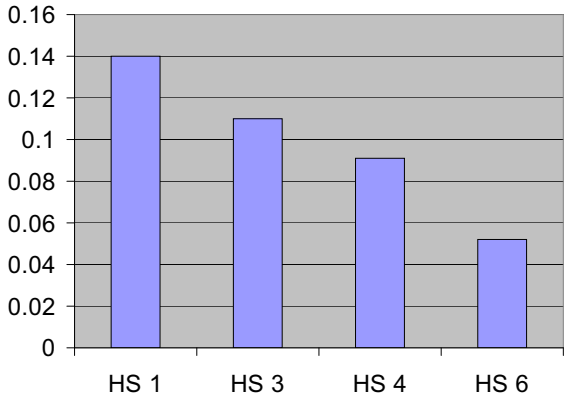


Figure 3.5: Cohesion levels

evolving building blocks from different subsystems which were designed to evolve independently. The software of these subsystems have been designed to co-evolve with the hardware they are associated with, but not to co-evolve with each other. The result was surprising also, because the contained building blocks were created relatively recently.

Except one, all top 10 potential unwanted couplings were found to be real unwanted couplings. The exception refers to a hot spot containing two building blocks. From those two, one was recently removed and consequently the set no longer points out an unwanted coupling.

The architects found our approach helpful to reach their goals. They considered to resolve the identified unwanted couplings in the future. The reason why they did not start with the investigations immediately is that they expected to resolve the identified unwanted couplings when effort becomes available.

3.5.2 The Analyzed Hot Spots

In this subsection we further analyze the four hot spots which taught the architects about yet unknown unwanted couplings. From now on, we will refer to those hot spots as *analyzed hot spots*. First, we start with indicating at which cohesion level we found the analyzed hot spots, see Figure 3.5. HS x refers to the hot spot with rank x in the top 10 list.

As can be seen even the analyzed hot spot with the highest cohesion has a rather low cohesion value (0.14). This means that whenever a building block is modified in that cluster the estimated probability that another one in the same cluster needs to be modified is 0.14 on average.

The reason why this happened is twofold. First, most of the change sets concern one subsystem only. We identified 73,851 change sets and out of them 59,622 change sets contained building blocks from a single subsystem. It means that 80% of the modifications were local-

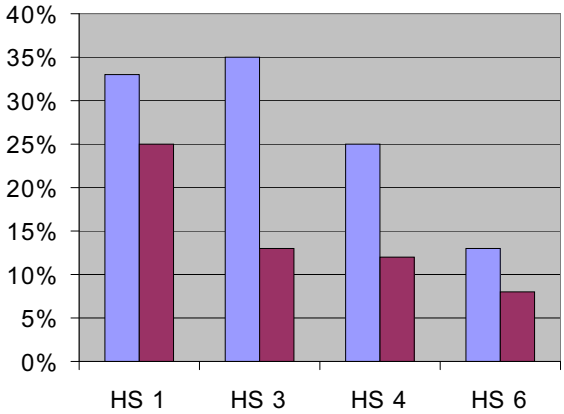


Figure 3.6: Modification Ratios

ized. On top of that, the Jaccard metric produces low values by its very nature, even if building blocks were modified together relatively frequently. For example, suppose building blocks B_1 and B_2 both changed 4 times. Once they changed together. Then the Jaccard parameters a_{11}, a_{10} and a_{01} are 1, 3 and 3 respectively. So the Jaccard similarity is 0.14 while 25 % of the changes related to B_1 or B_2 concern both B_1 and B_2.

The benefit of applying a symmetric similarity metric, like Jaccard, is the applicability of the clustering algorithm. Even though we have to work with low similarity values, we are still able to point out relevant unwanted couplings as illustrated in Table 3.3.

We continue the analysis of the analyzed hot spots by showing properties of the modifications when the directions are also taken into account.

Every analyzed hot spot indicates an unwanted coupling between pairs of subsystems. The bars in Figure 3.6 give the answer to the following question: How often did a modification of a building block from a subsystem induce a modification of a building block in the other subsystem? Figure 3.6 makes it more clear than plain Jaccard values that the subsystems from each of the four cases did co-change together relatively frequently. Compared to Figure 3.5 in Figure 3.6 more information is provided, because the direction of the co-change is also given. The fact of frequent co-changes indicated by Figure 3.6 may point out that the current implementations of the building blocks are not according to the original design decision.

In order to make the relation between Figure 3.5 and 3.6 clear we illustrate an example for an analyzed hot spot and explain how we would have included it in Figure 3.5 and 3.6.

Figure 3.7 shows an evolutionary cluster crossing subsystem borders. The small squares indicate building blocks and the large dashed rectangle shows the boundary of the evolutionary cluster. The vertical line indicates the border between the two parts (subsystems) of this evolutionary cluster. The arrows indicate that given the total number of change sets containing a building block from one part of the evolutionary cluster, how often a building block from

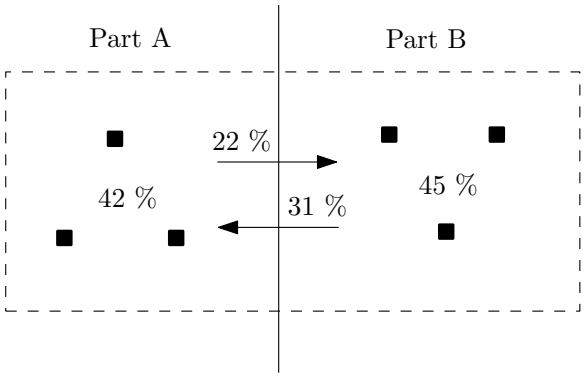


Figure 3.7: Example Hot Spot

the other part of the evolutionary cluster is also included in the change sets, on average. The numbers inside the triangles of building blocks show how often a modification of a building block from one part of the evolutionary cluster induces the modification of at least one other building block from that same part, on average. These percentages indicate the internal cohesion of the parts of the evolutionary cluster. The Jaccard parameters a_{11}, a_{10} and a_{01} in this example would be x, $\frac{100-22}{22}x$ and $\frac{100-31}{31}x$ respectively. This results in a cohesion level 0.14 between the parts. The corresponding percentages as depicted in Figure 3.6 would be 22 % and 31 %.

3.6 Discussion and Future Directions

During the development and implementation of our approach we learned about the following issues which need additional research:

- Active interaction with experts was needed in most of the steps of our approach. For instance, when it comes to the validation of the potential unwanted couplings or when the input for recovering change sets has to be cleaned.
- A deeper knowledge about the development process is needed to customize our approach. When change sets are identified, for example, we had the assumption that every modification related to the same reason is checked-in by the same developer. This might not be true for every development process. Also, the extent to which some change set approximation introduces noise depends on the applied development process.
- It is important to carefully select the appropriate formulas, see for instance Jaccard similarity or average linkage. Without investigating the effects of the applied formulas, the result of our approach would be meaningless.

- Massaging the real world data to fit the applied methods is important. For example, building blocks had to be considered in our case study instead of files due to scalability issues and because architects were interested in unwanted couplings at the level of building blocks. Now that unwanted couplings are identified at the level of building blocks, we can focus on them and to dig deeper to analyze dependencies at lower abstraction levels.
- Although the cohesion values of the analyzed hot spots are low, compared to each other they are helpful to get an idea about the amount of impact the underlying unwanted couplings have. This way, our approach can be used to indicate which part of the initial decomposition needs to receive special attention. When we know where to focus, our approach can be re-applied at a lower abstraction level.
- The dendrogram visualization of the non-trivial evolution clusters makes it easy to identify hot spots: they are the bottom leaves of the dendrogram which are easy to find.

Next to the identification of potential unwanted couplings the dendrogram created in our approach can be used for other types of analysis as well. The visualization of the dendrogram helps to spot parts of the software system which evolved differently from the rest. At the top right part of Figure 3.4, for instance, we can see building blocks which were never co-changed with the rest of the software system. Also, as can be seen from Figure 3.4, most of the clusters are joined at a rather low cohesion level. As discussed before, it is mainly the consequence of the relatively huge amount of local modifications.

Although this chapter describes the assessment of the initial decomposition, our approach can easily be generalized to help create the initial decomposition itself.

In some cases, it might not be possible to have the initial decomposition, because the software system is poorly structured, and there is a lack of expert knowledge available, for instance. Identifying the cluster hierarchy of co-evolving files, as described in our approach, and cutting the dendrogram at a given cohesion level then results in a partition of files which might be used as an indication of what the decomposition of the software system should be. In other words, our approach cannot only be used for assessment purposes but also to provide a decomposition to start with.

Furthermore, the identified cluster hierarchy can also be useful when there are files we cannot place into the parts. We may group these files into a *default* part of the initial decomposition. Using the cluster hierarchy from our approach, we can observe: (1) how the default part should be divided (2) how parts of the default part should be joined to other parts.

Figure 3.8 depicts the described generalizations. $P_i, i \in [1..10]$ and DP are parts of the initial decomposition, where DP denotes to the default part. $C_j, j \in [1..6]$ are clusters of entities identified by our approach. C_1 and C_2 are used for assessment, as described in this chapter. C_3 and C_4 can be used to refine the initial decomposition and decide which part of the default part should be assigned to the $P_i, i \in [1..10]$ parts. Further, clusters C_5 and C_6 suggest how to partition DP when no other input is available for the separation.

Although our approach stops at the point of assessing the parts of the initial decomposition, a way to address the identified unwanted couplings is still needed to improve the evolvability

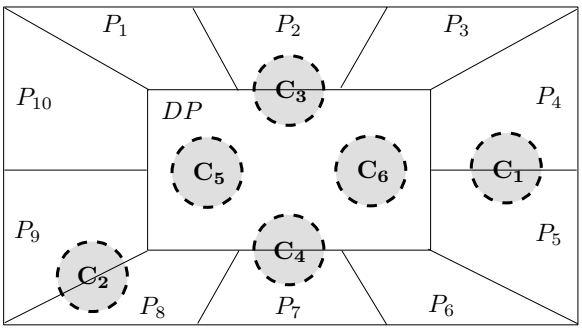


Figure 3.8: Approach generalization

of the parts. How we can actually reason about and help resolve the unwanted couplings is the topic of Chapter 5. We expect that by investigating underlying reasons for the frequent co-changes of files, for instance call or include relations, we may classify the solution types.

3.7 Conclusion

Defining or improving the decomposition of a legacy system such that the parts of that system decomposition can evolve as independent as possible is a key evolvability issue. Although this issue was already identified before, most previous work addressed it by looking at static relations (e.g. call relations, include relations) in the source code. However, considering static relations exclusively allows us to cope with part of the problem only. Software entities can frequently co–evolve even if they are not directly connected by a static relation. The underlying reasons for co-changes can also be run-time or semantic relations, for instance. To complement previous work, we elaborated how history information, stored in version management systems, can be used to assess a decomposition of the software system where the parts are expected to evolve independently in the future.

History information was already used in previous studies, but mainly for identifying patterns in the evolution of software entities and determining modification similarities. We used history information with the intention to assess a decomposition of the software system such that evolution is as localized as possible.

In this chapter, we described our history–driven approach to identify evolutionary clusters, i.e., set of software entities which co-evolved in the past. Based on the evolutionary clusters our approach help assess the decomposition of a software system by (automatically) identifying unwanted couplings. Furthermore, we applied our approach in a real industrial environment, considering a software system with multi million lines of code and a long development history. We described both how we implemented our approach and the lessons learned during the application. The application of our approach shows that it can be used effectively to assess the evolving parts of the software system.