Supporting Architecture Evolution by Mining Software Repositories

Vanya, A.

2012

document version
Publisher's PDF, also known as Version of record

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

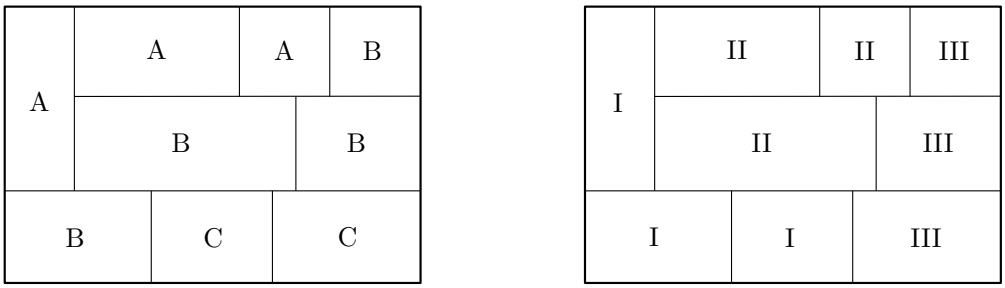
As mentioned before, evolving software systems, like the one studied, is a difficult task to achieve. One of the reasons is that software systems are typically huge and complex. It is not uncommon that industrial software systems contain thousands of files and an order of magnitude more functions. When 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