Supporting Architecture Evolution by Mining Software Repositories

Vanya, A.

2012

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

Conclusion

6

The architect of a large, evolving system may wish to revise its decomposition from time to time, for instance because the structure has deteriorated over time, certain components need to be outsourced to another site, and so on. One way to assess the current decomposition is to consider the past evolution, searching for components that often changed together. We iteratively devised and implemented a process for doing so at Philips Healthcare MRI. In this chapter we describe the lessons learned on how to effectively support architects improve their system decomposition.

6.1 Introduction

Loose coupling between software components is a well-known design consideration [Yourdon and Constantibe, 1975]. Architects aim to decompose the system such that components are loosely coupled, to enhance modifiability. During evolution, the decomposition tends to degrade [Lehman et al., 1997], and the architect may wish to revisit the decomposition to improve its future modifiability.

One way to assess the current decomposition is to investigate the past evolution of the system. We can measure the coupling between components by considering actual changes that involved multiple components. The common static notion of coupling between components is then replaced by a more dynamic one. Components that change together because of the same development task, such as a feature modification or a problem resolution, are coupled, from an evolution point of view. Gîrba et al [2004] used the term *yesterday's weather* to characterize this idea: if we have no further information, we may guess that today's weather will be like yesterday's. Next, the architect may decide to reduce the coupling between these components, for instance through refactoring.

In a large system, with many components, not all of these couplings between components need to be judged equally critical. For instance, a coupling between two components that are developed and maintained at two different sites may be judged more critical than a coupling between two components that are developed and maintained at the same site, since communication between developers at different sites is often more problematic than communication

Parts of this chapter has 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

between developers at the same site. In an embedded system, a coupling between components that reside on different pieces of hardware may be judged more critical than a coupling between components that reside on the same piece of hardware, since the former makes hardware replacement more difficult. Either type of coupling, or both, may be deemed undesirable. This is an instance of the well-known multidimensional separation of concerns [Tarr et al., 1999]. Since architects usually have limited time available, they can only handle a few of the unwanted couplings, so we have to be careful in selecting the couplings they are most interested in. If our process results in, say, a list of 100 unwanted couplings that the architect needs to study in order to identify the ones he really cares about, our process will not be used. The ultimate value of the process lies in its ability to offer the architect a short list of unwanted couplings that really matter to him.

In the past four years we have developed and experienced with a process to support architects in assessing unwanted couplings in a large industrial system. Our efforts have led to the identification of a number of unwanted couplings that the organization did not know of yet [Vanya et al., 2008], and helped in resolving them [Vanya et al., 2010]. Of the top 10 unwanted couplings identified in [Vanya et al., 2008], all were confirmed by the architect. Nine of them were found to point to evolutionary problems; of those, five denoted known issues, and four yet unknown ones. One unwanted coupling had been fixed by the time the analysis was done. For the 10 unwanted couplings analysed in [Vanya et al., 2010], a resolution was found for 6 unwanted couplings. For one unwanted coupling, it was decided to take no action, while 3 remained unsolved.

In this chapter we describe the lessons learned on how to effectively support architects improve their system decomposition, from applying this process to a large industrial software system at Philips Healthcare MRI.

6.2 Process Outline

The process for helping the software architect improve the decomposition of the software system gradually emerged. The process evolved based on our observations and the feedback we received from the architect and developers involved. We carried out three case studies, each lasting 6-8 months. In each case study, we executed all steps. In the first case study, we used a simple version of steps 4 (select potentially unwanted couplings) and step 5 (analyze potentially unwanted couplings). In the second case study, we extended step 4 to allow for a richer, multi-dimensional, selection of unwanted couplings. Finally, in the third case study, we elaborated step 5, and used an interactive visualization tool to help analyze the unwanted couplings. Within each case study, there were many smaller iterations with the architect and developers, to try out ideas, improve individual steps, and the like. All case studies involved the same software system.

We kept a careful log of events during the execution of the case studies. These logs were used to improve the process for the next round, and acted as input for identifying the lessons learned.

We help an architect by identifying unwanted couplings. We do so by analyzing the version

history of the system. Since architects are pressed for time, we have to be careful in presenting them the most critical unwanted couplings only. Next, we allow them to interactively explore each such cluster to identify underlying causes and possible remedies. This is then followed by actually resolving the issue, e.g. through some refactoring. In somewhat more detail, the steps of the process as it has evolved are as follows:

Step 1: Raw Data Collection In identifying the unwanted couplings we follow a retrospective approach; we assume that if components changed together a lot in the past then they will also change together in the near future. The assumption we use here is also commonly known as the Yesterday's weather assumption, see also [Gîrba et al., 2004]. In this step we first collect historical metadata from the version management system about the changes made to files. We next process the raw data, for instance to ensure data consistency. The goal of this step is to obtain a representation of the data in a database that is suitable for later querying.

At the lowest level in the version management system, we have files. A file can be many things: a C header file, a C source file, a set of configuration settings, a test case, and so on. Not all files are of interest to us; for instance, files corresponding to test cases can be filtered out at this point already. Similarly, files resulting from a build, such as executables and DLL's, are filtered out.

In our environment, files are grouped into building blocks, and building blocks are grouped into subsystems. In the directory structure then, we may have a path `X\Y\Z`, where `Z` is a file in building block `Y` in subsystem `X`. There may of course be various other abstraction levels in between, but these are the ones we used in our studies.

In most similar studies, each file in the version management system is treated as a separate component. In very large systems, this abstraction level may be considered too low, and one may choose a higher level of abstraction. In our environment, for instance, we used building blocks as components. We then say that a building block changes whenever any of its files changes. Our method, though, is independent of this abstraction level. In the sequel, we use the term *component* to denote the level at which changes are recorded.

Step 2: Recover Change Sets Developers are typically working on development tasks, like a feature modification or problem report resolution. All components which are modified because of the same development task form a *change set*. Determining which components changed together because of the same development task is not straightforward. This type of information is retrieved from the version management system. When developers change files and check them in again using the version management system, they do not always capture the reasons for the change. This is also true for our study environment, since direct links between modifications to components and development tasks are not captured. In our environment, the only available metadata is the developer's id and the timestamp. Our approximation is an adaptation of the work of Zimmermann et al. [2004]. We put consecutive check-ins with the same meta-data in the same change set, unless the difference of their time stamps is larger than δ (typically, we choose δ =200 msec). Further details on how we approximate change sets is given in [Vanya et al., 2011].

Step 3: Cluster Similar Changes For a relatively large software system, like that of Philips Healthcare MRI, the previous step results in a huge amount of information on when

components changed together. In this step, we take the change sets approximated in the previous step and identify those groups of components which were changed together relatively frequently. We use the Jaccard coefficient [Abreu et al., 2006] to measure the similarity between components. We use the Agglomerative Hierarchical Clustering Analysis [Wiggerts, 1997] to iteratively cluster the components.

We call the identified groups of components *evolutionary clusters*. Since we are interested in analyzing frequent changes involving different subsystems, we prune the set of evolutionary clusters by removing all that involve a single subsystem only. The result is a set of evolutionary clusters involving more than one subsystem. For the system we analysed, we ended up with 595 evolutionary clusters involving more than one subsystem. Note that we started this step with a collection of change sets, and end with evolutionary clusters which are sets of components. Further details of this clustering are given in [Vanya et al., 2008].

Step 4: Select Potentially Unwanted Couplings Not every evolutionary cluster denotes an unwanted coupling. In this step, the evolutionary clusters are characterized and next queried based on the interest of the software architect. For instance, we may characterize evolutionary clusters by aspects like:

- whether or not they involve more than one site, development group, or piece of hardware
- the distribution of changes over time
- the size of the cluster

An example query on the set of evolutionary clusters then is: "I am interested in clusters that involve more than 7 components, involve more than one site, and that occurred relatively recently". The goal of this step is to identify those evolutionary clusters that are deemed most unwanted by some architect at some point in time. The architect cannot investigate all evolutionary clusters (almost 600 in our case), so we need a method to reduce this set to a much smaller set which only contains the evolutionary clusters the architect is interested in. The characterization we employ extends previous work of, for example, [D'Ambros et al., 2009], which only considers the number of times components changed together as a selection criterion. The additional criteria we use are based on discussions with the architect, as well as common sense reasoning. For example, if components in an evolutionary cluster have changed together many times, but the frequency of their co-changes has become quite low by now, the architect may not be that interested in this particular evolutionary cluster since its constituent components are less likely to change in the near future (the yesterday's weather assumption). Conversely, an architect may be interested in evolutionary clusters that contain many components, irrespective of the frequency of co-changes. In [Vanya et al., 2009] we justify the dimensions we use to select the evolutionary clusters, as well as how we next select the evolutionary clusters the architect is interested in.

Step 5: Analysis of Potentially Unwanted Couplings The evolutionary clusters which we selected in the previous step need further investigation to (1) find out why the components involved changed together, (2) decide if the cluster indeed denotes an unwanted coupling, (3) identify possible solutions and (4) select one of the alternative solutions identified. In this step

the components and their relationships of the selected evolutionary clusters are visualized and interactively explored by the software architect and developers. The visualization allows the user to zoom in/out, set the level at which entities are to be visualized, set the lower bound of co-changes that need to be shown, hide or show internal changes within components, drag entities around, and the like. The capabilities of iVIS are similar to those of other tools like those described in [Gall et al., 2003, Pinzger et al., 2008]. It is the interactive usage to investigate evolutionary clusters and identify their underlying causes that makes the difference. The capabilities of our visualization tool iVIS and the descriptions of some sessions we had with the architect and developers can be found in [Vanya et al., 2010].

Step 6: Change the Decomposition Based on the decisions the architect made about the evolutionary clusters analyzed, the changes to improve the decomposition of the software system have to be carried out.

The process to help a software architect dealing with unwanted couplings by and large is a sequential process where the steps described above are executed in the order given. However, we have experienced that when the software architect and developers become more involved in the process (Step 4 and Step 5) they have many observations which may force one to return to one of the previous steps and improve the results of those steps. Typically, Step 1 and Step 2 need to be iterated more than once. Step 3 is relatively straightforward and can be largely automated. Figure 6.1 illustrates the process described and the artifacts used and/or created during the process steps.

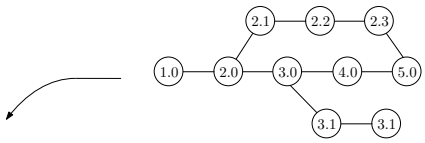
6.3 Categories of lessons learned

In Philips Healthcare MRI, we executed the first five steps of the process outlined in Section 6.1 in three successive case studies. The sixth step was also executed but we were not involved; it was done by the software developers. During the case studies, we have kept careful logs about the preparation, execution and results of those studies. For example, we captured how we extracted data from the version management system, what the participants of our working sessions told and what we could / could not achieve with our case studies.

We carefully went over the logs, marking incidents and labeling them as lessons learned. For example, our logs tell that every time when we had to transfer knowledge from the developers to the architect, we had difficulties to decide what exactly to capture and how to do so such that it is useful to the architect. On the other hand, architects and developers could intensively work together when they were all present. We grouped these incidents into a category that architects and developers need to be involved in the sessions at the same time. This categorization step resembles the classification phase of grounded theory [Corbin and Strauss, 1990] and similar qualitative methods. We did not follow a strict grounded theory approach, though. Many of the lessons learned were picked up already during one of the case studies, and had been incorporated in the process as described in section 6.2. For instance, the lesson *"Architects' concerns vary"* is a major reason for having Step 4 of the process. In those cases, we searched for evidence in the logs.

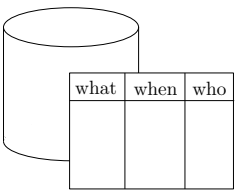
Having the lessons learned identified, we reviewed them in order to remove well known

Figure 6.1: The artifacts generated and used by the process (C_i $i \in [1..N]$ denote components)



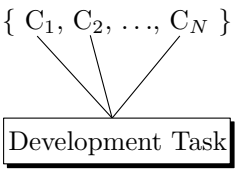
CMS

Step 1: Raw Data Collection



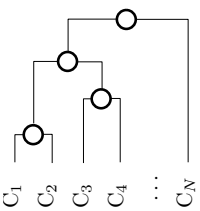
Database

Step 2: Recover Change sets



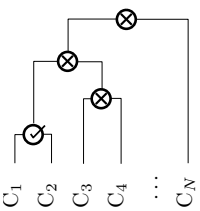
Change Set

Step 3: Cluster Similar Changes



Clusters

Step 4: Select Potentially Unwanted Couplings



Potentially Unwanted Couplings

Step 5: Analyse Potentially Unwanted Couplings

Visualization

Step 6: Change the Decomposition

or trivial ones. The remaining lessons showed some commonalities in how they support the software architect improve the decomposition of the software archive. Based on these commonalities, we classified those lessons into three major groups. The marking of incidents was done by one researcher, the categorization into lessons learned, the filtering thereof, and the classification into the three groups was done by two researchers. The groups distinguished are:

1. *Knowing the process* of how files are modified helps to interpret the data gathered from the software archive. If one does not know enough of the software process one is likely to mis-interpret the data. For example, in our environment the merging of (temporary) branches in the version management system may result in couplings which do not correspond to a development task. If such an (artificial) coupling is next presented to the software architect as an unwanted coupling that needs fixing, it will decrease the interest and belief of the architect in the approach, and he will be reluctant to initiate any modifications based on the suggestions made.

2. *Knowing the data* is another major success factor. The historical data extracted from the version management system must be processed carefully before it can be used in the next process steps. For instance, file paths in the directory structure need to be mapped onto file names used in the archive meta-data.

3. *Frequent feedback loops* with stakeholders (e.g. architects and developers) are necessary so that they understand the approach. This is a necessary prerequisite to build trust in the results. For instance, the stakeholders need to understand how change sets are approximated, and how change sets are clustered. Otherwise they will not be convinced the unwanted couplings are the ones that really matter.

We use the three major categories to classify the lessons learned in Section 6.4. An overview of the lessons and the process steps they relate to is given in Table 6.1. The symbol $\surd$ indicates which process steps a lesson is related to.

The lessons learned in the category *"Knowing the data"* all relate to the first procss step (Raw data collection). This process step is relatively independent of the actual analysis of unwanted couplings, and so are the lessons learned. The same or similar observations can be made for other analyses that involve mining software repositories.

The categories are not independent. For instance, knowing the development process means that one has a better understanding of how files were modified and as a consequence a better understanding of the data to use. Knowing the historical data better helps to reconstruct what happened in the past and therefore it improves the understanding of the software process. Also, having frequent feedback loops with stakeholders results in a better knowledge of the software process and/or the data itself. Figure 6.2 depicts the above relationships between the categories.

Table 6.1: Lessons Learned and Their Mapping to Process Steps

Lesson Learned	Process Step 1	2	3	4	5
Know thy process		√			
Involve both architects and developers					√
Set the right abstraction level	√				
Processing the raw data takes time	√				
Reduce data early	√				
Architects need to understand the computations		√	√		
Multiple simple measures are better than a single complex measure			√	√	
Architects' concerns vary				√	
Couplings should be traceable to the raw data					√
Don't rely on developers' first impressions					√

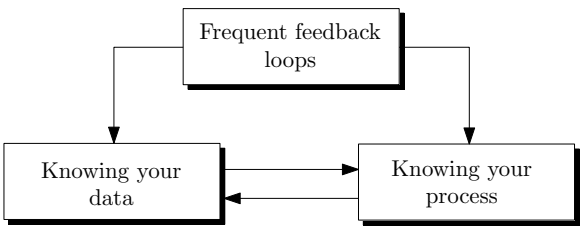


Figure 6.2: Impact relationships between categories

6.4 Lessons learned

6.4.1 Knowing the process

Know thy process

Anecdote. In our study environment developers captured modifications related to a development task in many different ways. Some did it every hour, others only when a task is finished. In our case study environment, we also found that certain change sets contained a very large number of files, sometimes up to a thousand. During discussions with developers it turned out that these change sets were a side effect of merging branches. Such tasks are "artificial" in that they do not correspond to real development tasks, and the corresponding change sets must be disregarded. Furthermore, while analyzing the meta-data from the version management system, we found that many branches contained the words "task" and "team". What also struck was that these branches had a much shorter lifetime than others. When asking the local expert after the purpose of these branches, the answer we got was

> *We started to create those branches only a few years ago. When working on a development task, a single developer or a group of developers check-in related modifications to a "task" or "team" branch. Recently, we started to use such branches more frequently.*

Unfortunately, we could not use task and team branches to approximate change sets, since their still scarce usage provided us with a small amount of historical evidence. Therefore, we decided to approximate change sets from check-in related meta-data.
Conclusion. The way developers address development tasks has an impact on how modifications are captured. Therefore, when determining how to approximate change sets from the individual modifications (Step 2 of the process), one has to take the software development process into account. Not only is there a relation between the software process and how modifications are captured, but the process may change over time (as the above anecdote illustrates) and so the way change sets are approximated may also have to change over time. More frequent usage of post and team branches reflect such a change in the software process.

Involve both architects and developers
Anecdote. In one of the working sessions to interactively analyse unwanted couplings, only the concerned developer was present while the architect had other commitments. During this session, the developer examined the change sets at several layers of abstraction to attempt understanding why the files were co-changed. After the session, the writer of this thesis discussed the observations with the architect but was unable to answer all the architect's questions. Typically, the architect would ask "what did he [the developer] mean with ..."?
Conclusion. Analyzing unwanted couplings (Step 5 of the process) is a complicated task which requires developers and the software architect to work together. The developers can express the lower level reasons for changes and the architect can abstract away to the higher-level design. Moreover, the architect does then have the possibility to discuss the feasibility of solution alternatives together with the developers. Also, what the architect needs to know to understand an unwanted coupling may differ from case to case. Domain knowledge plays a large role in understanding such change-sets, and it is difficult to predict how much thereof the architect knows, and what needs to be explained.

6.4.2 Knowing the data

Set the right abstraction level
Anecdote. Initially, we tacitly assumed the architect was interested to know of unwanted couplings at the file level. Existing research usually adopts that abstraction level too. It is also somewhat easier to reason about co-changes at that level, because the granularity of check-ins to the version control system is at the file level. However, clustering changes to more than 40,000 files turned out to be computationally infeasible. Only upon checking with the architect did we learn he was more interested in unwanted couplings at a higher level of abstraction.
Conclusion. It is not obvious which abstraction level the architect is interested in. In our case the architect wanted to assess unwanted couplings at the level of building blocks. Building

blocks are directories in the code structure representing the next level of abstraction above files, see also [van der Linden and Müller, 1995]. Doing the analysis at the building block level not only saves a lot of compute cycles, but also allows one to be more responsive when supporting architects. Setting the abstraction level is done during the raw data collection step (Step 1).

Processing the raw data takes time
Anecdote. In our case study environment, data is not always stored consistently. For example, we had to map the actual file names from the version management system onto the filenames as they appear in the filename–building block mapping. The next time we had to do this mapping, things had changed a little bit: new files had been created, and new rules had to be followed for the mapping (including new exceptions to these rules).
Conclusion. After extracting the raw historical data from the version management system it has to be processed before we can use it to approximate change sets. This processing (Step 1 of the process) takes quite some time. As part of the processing one has to make sure, for instance, that (1) one does not use historical data related to files the architect is not interested in, (2) one matches the path of the files to the meta-data with the paths used to define what belongs to the same component. In addition, in our environment we had to transform the file paths and file names to the name of the containing building blocks.

While the system evolves, new historical data is generated because of the continuous evolution of the system. It may at some point in time become relevent again to assess how well the current decomposition still supports independent evolution. Therefore, the data collection step (Step 1) needs to be repeated. To that end, it had better be automated as much as possible. Automating the data extraction not only helps to repeat the process but also to maintain consistency between the previously and newly extracted historical data.

6.4.3 Frequent feedback loops

Reduce data early
Anecdote. When we first clustered similar changes, the developers analyzing those clusters pointed out that some files are not considered part of the software system proper. One reason being those files belong to administrative tools. We could safely discard those files in the clustering and subsequent analysis.
Conclusion. In our study environment we extracted hundreds of thousands file check-ins, which is a challenge to handle. Software architects often have a clear idea about which files are surely not worth investigation. By consulting software developers and architects in the first process step already, one not only reduces the amount of data to work on, but one also buys in confidence. Reducing the raw data (in Step 1 of the process) also results in a decreased computation time in the next steps of the process.

Architects need to understand the computations
Anecdote. In the first case study, the architect often questioned the unwanted couplings identified. It turned out he did not really understand the reasonably complex computations used to

derive the unwanted couplings. Only after we carefully explained the meaning and role of the Jaccard coefficient and the algorithm used to cluster similar changes, did his attitude change. At subsequent meetings, his attitude changed from *"Why is this identified as an unwanted coupling"* to *"Why do the files in this unwanted cluster change together"*.

Conclusion. The way change sets are clustered needs to be understood and accepted by the software architect. If the resulting clusters are computed using "magical" formulae, one runs the risk that the architect will not understand and therefore will not trust the results. The architect did not know whether the unwanted couplings were real or just the result of our seemingly "magical" way to compute them. The clustering is indeed somewhat complicated; both the Jaccard similarity coefficient (used in Step 2) and the Agglomerative Hierarchical Clustering Analysis (Step 3) need to be carefully explained.

Multiple simple measures are better than a single complex measure

Anecdote. The output of the clustering algorithm is a large set of evolutionary clusters. Since the architect has limited time to handle issues, we need to identify the most important clusters. At first, we sorted them according to the frequency with which the files in the cluster changed together, and we presented the architect the clusters whose files changed most often. At a subsequent meeting, it turned out he had other criteria too to decide on which clusters to investigate. For instance, clusters that involve more than one site were considered more important than clusters involving a single site. We next worked on more complex metrics to use in the clustering algorithm, for instance metrics that combined change frequency and the number of sites involved. This was however rejected by the architect, since he thought it would then become too difficult to interpret the results of the clustering.

Conclusion. The process of supporting a software architect improving the decomposition of the software system has to be made as simple as possible. The more simple such a process is, the more responsive we can be to the software architects. It may seem that using a single measure (in Step 3) which combines multiple properties of relations between file pairs would simplify the filtering step (Step 4). While this is true, we then loose the possible reasons of why a coupling is important. Furthermore, if we can select unwanted couplings one way only, we loose the possibility to support different architects with different interests. In our environment, for instance, one simple measure we used is when files changed together the last time. Another such measure indicated the size of (i.e. number of files in) a cluster. A third simple measure is the number of sites involved. By presenting all these measures to the architect, he can himself decide how to balance them. These measures are computed in step 3 of the process, and next used to filter the unwanted couplings in step 4 of the process.

Architects' concerns vary

Anecdote. A main concern of the architect we worked with was to ensure that development groups at different sites could work as independent as possible. So he was interested in evolutionary clusters that involved more than one site. We initially focussed on that concern when selecting the couplings most unwanted. At a subsequent meeting, we however learned that a main concern of another member of the architect team was to make sure that certain chunks of the software could be deployed independently.

Conclusion. Huge software systems, like the one developed at Philips Healthcare MRI, have more than one architect. Those architects share the responsibility to create and maintain the architecture of the software system. This may lead to a situation (as in our environment) where architects are responsible for different decompositions of the same software system. For instance, one architect may be responsible for the decomposition of software components over different sites, while another may be responsible of the assignment of software components to different pieces of hardware. A coupling which concerns different sites according to the first decomposition may relate to the same piece of hardware in the latter decomposition. Therefore, one coupling may be undesirable to one of the architects and it may at the same time be a non-issue to another architect. Also, one and the same architect may have different concerns over time, resulting in different interpretations of the criticality of an unwanted coupling. These concerns are used in Step 4 of the process to select the unwanted couplings.

Couplings should be traceable to the raw data

Anecdote. In one session with the architect and two developers, a developer asked for the historical information used to derive the unwanted coupling. He wanted to know the person who had changed the files, on which branch, and when exactly. When this information was brought to the table, he could recall the tasks in which those co-changes occurred. This allowed him to gain a deeper understanding of the relationship of the files involved.

Conclusion. The visual exploration of couplings (in Step 5 of the process) has to be backed up with the original data. When developers have questions about the presented relations, they must be able to see/judge the change sets and eventually the check-in information used to derive the relations. During most of the analysis sessions we had, developers asked us to present the underlying historical information.

Don't rely on developers' first impressions

Anecdote. At one session, after glancing through an evolutionary cluster at stake, a participating developer identified the co-changes presented as a known issue, where certain patient data and the components using that data were not well separated. They had resolved the issue already, they thought. On the one hand, this built trust in our approach (*"This approach really points at important unwanted couplings"*). Since the approach is based on analysing historical data, it is only to be expected that, from time to time, it points to issues already resolved. Closer inspection however revealed that the solution implemented did not completely resolve the issue.

Conclusion. When we analyzed unwanted couplings (Step 5 of the process) together with the developers they initially considered high-level information, like which components are part of the cluster presented. Since developers are working on the source code on a daily basis, they do have knowledge about unwanted couplings. In most sessions, developers initially expressed a hunch about the unwanted coupling and its causes. In some cases though this initial idea of the developer was different from what a more detailed analysis at the level of relations between individual files showed. Therefore, one has to cross check the perceptions of the developers. This can be achieved by keeping looking at the details of the clusters. This lesson is critical since it influences how the method to support software architects is judged. If the architects perceive one is only presenting known unwanted couplings, they will quickly loose interest.

6.5 Related Work

6.5.1 Modularity for Software Evolvability

Maintaining the smooth evolution of a software system requires one 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 major concerns of software architects when they design for and maintain evolvability.

In this chapter, we describe how we support an architect to improve the decomposition of a software system. A software system is well modularized or decomposed if the components have a high cohesion and a low coupling [Yourdon and Constantibe, 1975]. The quality model presented by Brcina et al. [2009] shows that modularity positively influences analyzability, changeability, extensibility and testability. They are four out of the eight direct characteristics of evolvability. So modularity is an important quality attribute to be addressed.

Related work addresses modularity in different ways. The main reason is that different type of relations between software entities are considered. Those relations are used to compute cohesion and coupling. Hoffman and Eugster [2008] relate software entities if they implement the same concern. Kuhn et al. [2005] identify which software entities are semantically related with the technique known as latent semantic indexing. Du Bois et al. [2004] consider the static relations (import, include, call) between software entities. Similar to Ratzinger et al. [2005a], we are interested in which software entities have changed together in the past, i.e. co-changed.

When using co-change information to make components more independent, a series of steps have to be implemented. First, historical meta-data has to be extracted [Fischer et al., 2003a]. Then co-changed entities have to be identified [Vanya et al., 2008, Antoniol et al., 2005, Zimmermann and Weißgerber, 2004]. Finally, unwanted couplings between components have to be found and handled [Vanya et al., 2010, D'Ambros et al., 2009, Ratzinger et al., 2005a]. In this chapter we describe how the above steps form a process.

6.5.2 Critical Success Factors

Most of the lessons learned presented in this chapter are about factors affecting the success of supporting the software architect to improve the decomposition of the software system. In general, every process or activity having a clearly defined goal also has its success factors associated. For instance, when a software metrics program is introduced, an incremental implementation had better be followed [Hall and Fenton, 1997].

Once the success factors for a certain type of activity are found, they can be reused later on to help achieve the goals of newly initiated activities of the same type. Therefore, identifying and describing success factors is the goal of many research activities [Bullen and Rockart, 1981, Hall and Fenton, 1997, Caralli, 2004, Baddoo and Hall, 2003]. It is also the focus of this chapter.

There are many papers describing critical success factors for software process improvement. Some of them focus on the positive success factors, i.e. what has to be taken care of in

order to reach a goal. The works of Hall and Fenton [1997] and Niazi et al. [2006] are examples thereof. Other papers focus more on negative success factors, i.e. what has to be avoided in order to reach a goal. See, for instance, the work of Baddoo and Hall [2003].

Our work is similar to the ones identifying critical success factors for software process improvement. Most of the lessons learned we report in this chapter are in the form of what one should or should not do in order to successfully execute a process. In that sense, those lessons learned resemble the critical success factors for software process improvement. Also, both those critical success factors and our lessons learned address a similar question: how to collect and use data such that it will eventually bring benefits to the organization where the data is collected and analyzed.

Although they are similar, there are some differences between the lessons learned reported here and the critical success factors for software process improvement. First, we gained our experiences by (repeatedly) executing a process, while critical success factors are typically identified based on interviews with developers, experts and managers. Second, our lessons learned come from a single study environment. Critical success factors on the other hand are identified after gathering experiences from multiple projects / study environments.

6.6 Conclusion

Having a decomposition of a software system where components can evolve as independent as possible may improve evolvability in many ways. It may decrease delays in development, communication costs, the number of tests needed and alike.

In this chapter, we report the lessons learned while supporting architects at Philips Healthcare MRI improve the evolvability of their software system. In several iterations, we developed a process for doing so. The lessons learned 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 is crucial for 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. The key for deriving useful information is in the details

Section 1.6 defines the three concrete research questions of this thesis (**RQ-1**, **RQ-2** and **RQ-3**). All these research questions have been addressed in the previous chapters. The answers to these research questions can be summarized as follows:

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

Answer: Knowing which software entities changed *frequently* together requires us to know first which software entities co-changed. There are several industrial environments, for in-

stance the one studied, where that knowledge is not even captured. In other words, change sets are not always recorded. Furthermore, in the studied environment the meta-data related to individual changes is rather incomplete and the process to introduce changes to the software system (i.e. using postlists) is not completely according to the standard assumptions of previous work. Therefore, in such environments one needs to think about alternative ways to approximate change sets for retrieving co-changes.

In Section 2.4 we described five variants of the well-known sliding window algorithm, see also the work of Zimmerman et al. [2004], to approximate change sets from the historical meta-data of Philips Healthcare MRI. The development process executed had a significant impact on which change set approximation alternatives we identified. The approach described in this thesis to help software architects relies on the accuracy of the change sets approximated. For instance, change sets are used to derive co-changes. Co-changes are used to create evolutionary clusters which are further selected and analyzed by the software architect and engineers. At the end, the accuracy of change sets approximated has an impact on the decisions of the software architect. Since those decisions have to be taken with care, it is important to evaluate the accuracy of the change sets approximated.

We have described two approaches to evaluate the change sets approximated, see Section 2.5. One of them makes use of an on-line survey which developers need to fill in, while the other one is based on carefully selected postlists, i.e. lists of modifications sent to the system integrator. The results from both approaches were very similar. Based on those results we could select an approximation alternative for further usage where the precision and recall values where optimally combined (89% and 84% respectively).

Having the change-sets accurately approximated, the next step is to group those software entities which changed frequently together. For this, we described a clustering approach in Chapter 3. The result of this clustering activity is a binary parameterized tree or dendrogram where nodes, also referred to as evolutionary clusters, are sets of software entities changing together. The position of the evolutionary clusters in the dendrogram shows how frequently the related entities co-changed.

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

Answer: To answer this research question, we present an approach in Chapter 4 which we executed in Philips Healthcare MRI and on an open source, called ArgoUML. The approach first characterizes evolutionary clusters based on a few justified properties, for instance, the number of co-changes or co-change tendencies. The architects are contacted to elicit costly past evolution anti-scenarios which should be avoided in the future. Based on those anti-scenarios, a query is formulated on the evolutionary clusters characterized. The goal of that query is to find only those evolutionary clusters which point to unwanted couplings. The result of that query gives us the potential unwanted couplings which need to be further analyzed.

It is key in the approach presented that it makes it possible to consider the input from the software architects on unwanted couplings. As discussed in Section 4.3, what is considered to be an unwanted coupling is subjective and therefore every architect needs to be helped to

find unwanted couplings in alignment with their definition of unwanted couplings. Extracting evolution anti-scenarios from the architects is our way to explore what they find to be an unwanted coupling.

The results of the approach show that the selected unwanted couplings were pointing to unwanted couplings. Also, we rejected many such evolutionary clusters which the architect did not want to analyze for a good reason but which previous approaches would not have filtered out. The ability to filter evolutionary clusters is crucial also because there are hundreds of evolutionary clusters and architects are interested only in a few of them. Those few which point to unwanted couplings.

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

Answer: Interactive visualizations provide a means to its user to customize what and how to visualize. By interacting with visualizations the user gets information iteratively. In every iteration, information is collected first from the visualization and later on the user interacts with the visualization to get even more information. The interactions are guided by the interest of the user and his interest is influenced by the information previously collected.

In Chapter 5 we investigated how such an interactive exploration process can help architects and software engineers analyze and resolve unwanted couplings. In Section 5.3.2 we presented a range of possible interactions which we also implemented in our interactive visualization tool called iVis. We used this tool to visualize those evolutionary clusters which we previously filtered out for the software architect supported. For each of those clusters working sessions were organized with the architect and engineers to analyze them. During the working sessions we observed the benefits of using interactive visualizations. It turned out that interactive visualizations can not only help to reason about unwanted couplings but they can also help to find solution alternatives and assess their impact. In Section 5.5 we provide a list of observations and key lessons learned.

6.7 Future Work

Doing research is a continuous problem solving activity; while working on a research question new questions are encountered which need to be answered at a later stage. In that sense, a piece of research can rarely be finished without leaving some open questions behind. So is it with this thesis. The major areas of the envisioned future work is further elaborated on in this section.

From the laws of Lehman [1997] follows that improving the decomposition of a software system needs to be done continuously or periodically rather than only once. Future work would therefore need to investigate how the architect can be supported to *maintain* the required status of the decomposition over time. In other words, how to monitor the “health” of the software system’s decomposition.

In this thesis, mainly co-change information derived from a version management system

was used to analyze unwanted couplings. It would be interesting to research how the detailed analysis of such unwanted couplings can be helped by presenting multiple type of relations to the architect and developers. Such types of relations could be, for instance: run-time couplings, static relations, semantic couplings.

All the studies in this thesis have been carried our in a single development environment of Philips Healthcare MRI. Applying the work presented in this thesis in other environments could result in a better understanding about how to generalize the results.