Supporting Architecture Evolution by Mining Software Repositories

Vanya, A.

2012

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

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.

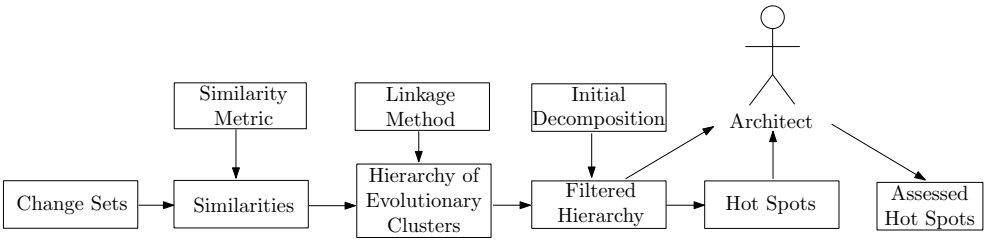

Figure 3.1: The evolutionary cluster approach

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,

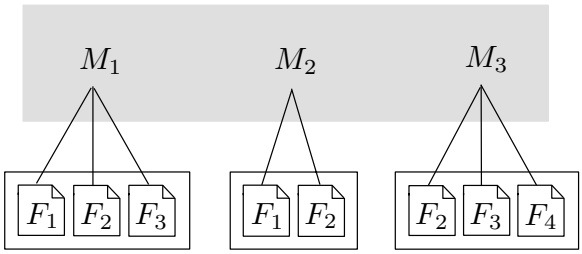

Figure 3.2: Change Sets

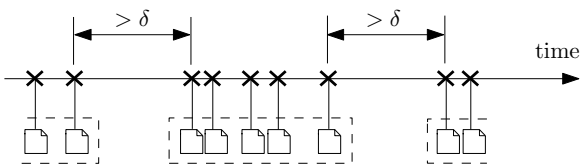

Figure 3.3: Change Set approximations

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.

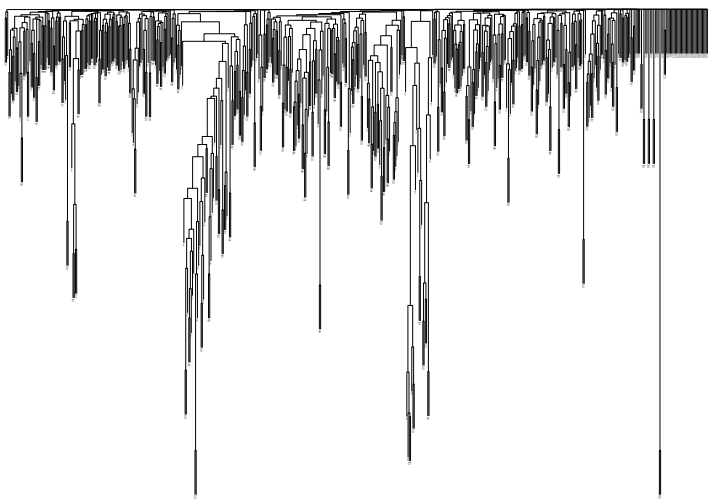

Figure 3.4: The complete dendrogram

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–

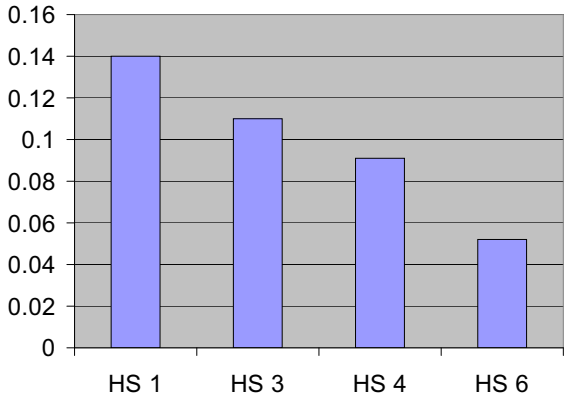

Figure 3.5: Cohesion levels

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-

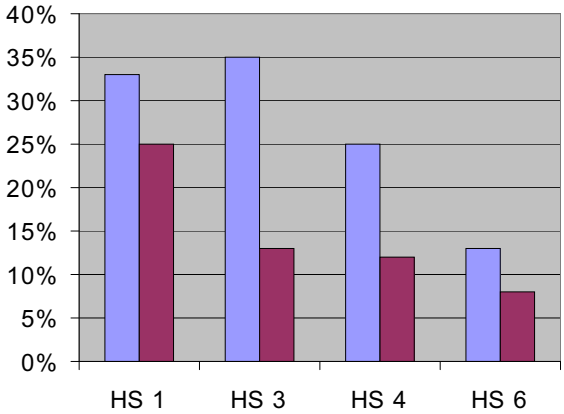

Figure 3.6: Modification Ratios

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

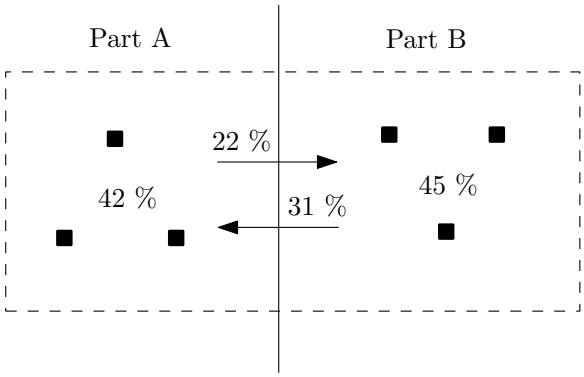

Figure 3.7: Example Hot Spot

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

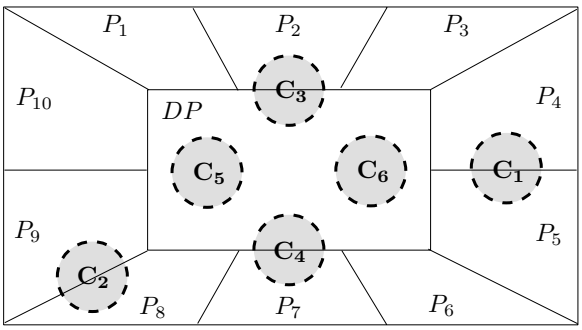

Figure 3.8: Approach generalization

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.