Supporting Architecture Evolution by Mining Software Repositories

Vanya, A.

2012

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

Multidimensional Characterization of Evolutionary Clusters

4

Software architects regularly have to identify unwanted couplings between different parts of a software system. Identifying groups of software entities which frequently changed together in the past, i.e. evolutionary clusters, are one way to help find such unwanted couplings. However, there may be many such evolutionary clusters. Not all of them point to unwanted couplings. In this chapter we discuss how a multi-dimensional characterization of evolutionary clusters can help identify unwanted couplings. In addressing this question we describe (1) properties used for characterizing evolutionary clusters, (2) scenarios characterizing unwanted couplings, and (3) the mapping of such scenarios to queries on a set of evolutionary clusters, resulting in a subset denoting unwanted couplings according to that scenario. We apply the proposed characterization to the case of a large embedded software system having a development history of more than a decade. By doing so, we execute Step 4 of the process described in Section 1.7 to help the software architect.

4.1 Introduction

In [Godfrey et al., 2009], several researchers give their opinion about the future of mining software archives. One of them, Michael Godfrey, is of the opinion that ”the future of mining software repositories (MSR) lies in tying software development to the kind of sensemaking that managers and software developers perform daily, right now mostly on the basis of a ‘gut feeling.’” And Martin Robillard states ”Software developers and other stakeholders spend a lot of time searching for information to solve problems ...”. This article fits both these characterizations. We deal with an issue that software architects in our study environment had to face: identifying unwanted couplings between parts of a software system.

Imagine an embedded system with two hardware components A and B, whose software is decomposed into three major subsystems X, Y and Z. Suppose X and Y are developed at one site, while Z is developed at another site. Furthermore, suppose subsystems X and Y often change together during the evolution of the system, but subsystem Z changes independent from X and Y. The latter creates a dependency between X and Y. If X is associated with hardware component A, and Y and Z are associated with hardware component B, this situation may be

Parts of this chapter has been published as:

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

deemed undesirable. For, if we want to, say, change to another manufacturer for hardware component A, such may not be easy to accomplish because of the dependency between A's software (X) and the rest (Y and Z). We call this dependency an *unwanted coupling*.

This notion of an unwanted coupling is relative: in the above example, it is only an unwanted coupling from the point of view of wanting to be able to change hardware components easily. If we are not concerned about changing hardware components, but only about the amount of communication between development sites, then the above dependency probably would not be considered an unwanted coupling. A dependency between subsystems only is an unwanted coupling in the eye of the beholder.

In the above example, we talked about subsystems which change together. What happens of course is that certain parts of these subsystems (such as components, classes) change together. We can retrieve that kind of information from the version management system, as illustrated by the seminal work of Ball on change couplings [Ball et al., 1997].

Software entities (such as files or components) which changed together in the past can help find unwanted couplings [Gall et al., 1998b]. Antoniol et al. [2005] describe how to detect *groups* of software entities changing together frequently. We refer to those groups of entities frequently changing together as *evolutionary clusters*, see also our previous work [Vanya et al., 2008].

The identification of evolutionary clusters is based on the concept of change sets. Change sets contain modifications of software entities, like files, which are changed because of the same logical reason. Such a reason can be the modification of a feature or the resolution of a problem report. As change sets are not always captured explicitly, they often have to be approximated from the available historical information, like check-ins in version management systems. Previous work [Zimmermann et al., 2005, Gall et al., 2003, Antoniol et al., 2005, German, 2006, Ying et al., 2004, Beyer and Hassan, 2006] describe these approximation techniques. The approximated change sets are then used to derive the evolutionary clusters. In [Vanya et al., 2008] we used a hierarchical clustering algorithm to identify evolutionary clusters. This process is sketched in section 3.4.3. Other known algorithms are using dynamic time warping [Antoniol et al., 2005], concept analysis [Gîrba et al., 2007] and a visual approach [Beyer and Hassan, 2006] to identify evolutionary clusters.

A common property of most of the algorithms above is that they tend to identify a large number of evolutionary clusters. Unless the relevant attributes of these evolutionary clusters are captured and presented, it is difficult to select the evolutionary clusters which indicate unwanted couplings. As for the identification of unwanted couplings, current approaches [Beyer and Hassan, 2006, D'Ambros et al., 2009, Fischer and Gall, 2006] suggest to look at the clusters which (1) contain software entities from different subsystems and where (2) the entities where modified many times in the recent past. These two criteria seem to be somewhat simplistic. First of all, those two criteria need not fully capture an architect's notion of an unwanted coupling, as evidenced by the embedded system example discussed above. Second, previous work does not take into account the fact that different architects may have different concerns and therefore consider different couplings to be unwanted.

Furthermore, once we identified evolutionary clusters, it is expedient to carefully charac-

terize those clusters, and to retain this characterization. The knowledge of evolutionary cluster characterization can then be reused when a new state of the software system is to be assessed. An example hereof is discussed in section 4.7.3, where we investigate whether moving certain files from one subsystem to another would solve an unwanted coupling, or lead to different unwanted couplings elsewhere. To that end, we redo the characterization on a different subsystem decomposition and compare the outcome with that of the original decomposition. This 'what-if' type of investigation allows for a quick assessment of structural changes to the software archive.

Let us summarize the process we follow. We first approximate change sets – sets of software entities that are changed together for the same reason – from the version management system. Next, we use these change sets to determine evolutionary clusters, of which there can be a lot. These evolutionary clusters are then characterized along different dimensions, such as whether they involve more than one site, how many entities are involved, and so on. The particular interest of the architect, i.e. the type of evolutionary cluster he considers to denote an unwanted coupling, is next used to query and filter the set of evolutionary clusters.

In this chapter we present how a carefully prepared characterization of evolutionary clusters can help identify unwanted couplings. To address this question we make the following contributions:

- Elaborate on which properties of evolutionary clusters could be used for an initial characterization
- Describe what architects consider to be an unwanted coupling and how this knowledge can be captured in scenarios
- Discuss how these scenarios can be translated to queries on the set of evolutionary clusters

The remainder of this chapter is organized as follows. Section 4.2 describes the study environment from which we take our examples and experience. Section 4.3 elaborates on why it is relevant to characterize evolutionary clusters. Section 4.4 describes the properties used for characterization. Section 4.5 briefly describes the concept of evolution anti-scenarios, which captures the knowledge of software architects about what has to be considered as an unwanted coupling. Section 4.6 goes through the characterization properties and justifies why they should be used in the characterization. Section 4.7 provides three examples of how the characterization can be used in a real-life situation. Section 4.8 discusses the generalization of our approach to other environments. Section 4.9 describes the threats to validity. Section 4.10 presents related work. Section 4.11 concludes this chapter.

This chapter is based on one of our previous works [Vanya et al., 2009]. In this chapter, we have added the 'what-if' type of investigation mentioned above, together with an example thereof in Section 4.7.3. Furthermore, we have extended our previous paper with a study comparing the outcome of the different approaches to query evolutionary clusters, see Section 4.7.4. The outcome of the study further clarifies in which ways it is beneficial to apply the multidimensional characterization of evolutionary clusters proposed. We also extended our

previous paper by adding a new section, see Section 4.8 where we discuss the concerns one may have when applying our approach in a different environment. In this section, we also apply our approach to another system. Furthermore, we have added Section 4.9 to describe the main threats to validity.

4.2 Study Environment

When identifying evolutionary clusters we had to choose the level of abstraction for software entities. We decided to observe the co-changes of building blocks because architects were primarily interested in investigating unwanted couplings at that abstraction level. The algorithm described in the previous section resulted in 595 evolutionary clusters involving more than one subsystem. In the software system studied, architects were interested to know unwanted couplings related to multiple decompositions of the system, see also Section 1.4 describing the system decompositions of interest.

The lead architect in our study environment was actively involved in nearly every step of our research, including: the identification and justification of evolutionary cluster properties (Sections 4.4, 4.6), the extraction of evolution anti-scenarios (Section 4.5), translation of those scenarios to queries and the deeper analysis and validation of the evolutionary clusters selected (Section 4.7).

4.3 Need For Characterization

Software architects have many tasks to perform [Hofmeister et al., 2000, Bass et al., 2003, Clements et al., 2007]. These tasks include, amongst others, the communication with stakeholders, the translation of requirements to design decisions, and the documentation and assessment of the software architectures developed. Making sure that decomposition elements can evolve as independent as possible from one another, is only one of the many requirements architects typically need to deal with. Fulfilling this requirement is an important factor to maintain the smooth evolution of the software system [Huynh and Cai, 2007], which is required from many software systems [Breivold et al., 2008]. As architects are typically pressed for time, they have limited time available to maintain the system's decomposition by resolving unwanted couplings. During this limited time, architects seek to address the most severe unwanted couplings.

To determine whether a dependency between entities really is an unwanted coupling worth looking at, architects have to weigh the cost of leaving the decomposition of the system untouched against the cost of resolving that dependency. A number of drivers influence these costs; the various dimensions of the characterization we discuss in the remainder of this article are important cost drivers considered by the architects. In our case study environment, architects collect unwanted couplings into a list. This list is used to initiate refactoring/restructuring activities. For practical reasons, the list only contains 20 elements. This does not mean architects do not encounter more unwanted couplings; they just put a threshold on the length, so

that only a few unwanted couplings make it to the list.

Although in practice it is only feasible to resolve a few unwanted couplings, approaches described in the literature typically result in hundreds or even thousands of evolutionary clusters [Antoniol et al., 2005] that involve more than one decomposition element. This is not a surprise when we keep in mind that a large software system contains tens of thousands of files. A need therefore arises to filter the set of evolutionary clusters to retain only those that denote unwanted couplings to the architect.

In a large software company it is customary that many people are concerned with the decompositions of the software system developed. Different architects and developers have different responsibilities and therefore they each define the notion of unwanted couplings in a different way. Architects need to have a global view of the software system and they are usually interested in unwanted couplings from all over the system. Developers on the other hand often need to be more focused and they consider dependencies to be unwanted couplings if they are related to the decomposition elements they are working on. Architects themselves are also different. One architect may be interested in dependencies involving different development groups. Another architect may look for frequent co-changes between decomposition elements that are supposed to be released independently. So it depends on the viewpoint taken which evolutionary clusters denote unwanted couplings.

In order to express which evolutionary clusters are the most important/severe ones for a specific architect or developer, we first need to characterize each of the evolutionary clusters. Previous work [Antoniol et al., 2005] implicitly uses a characterization based on the number of times software entities changed together and the decomposition elements those entities are part of. This type of characterization seems to be too simplistic to properly express the level of severity. We know from our practical experience (see also Section 4.7.5) that such a simple characterization will point to dependencies which are, most of the time, not acknowledged by the architect to be unwanted couplings. Unless we properly characterize the evolutionary clusters we run the risk that the added value of the evolutionary cluster identification is deemed limited by industry.

Once an unwanted coupling is identified, and a possible remedy is found, the architect wants to assess whether the proposed remedy would actually resolve the coupling, or maybe introduce new unwanted couplings elsewhere. For instance, the architect might suggest to move certain software entities from one decomposition element to another to resolve an unwanted coupling. Without actually moving the files, we may identify the evolutionary clusters based on this newly proposed decomposition, and next identify the most important unwanted couplings using the same characterization. This type of 'what-if' analysis is straightforward if we retain the characterization.

4.4 Properties of Evolutionary Clusters

Characterization of evolutionary clusters involves identifying the properties of those clusters and measuring the actual values for those properties. In this section we elaborate on the properties we use to characterize evolutionary clusters. In the description of the properties we use

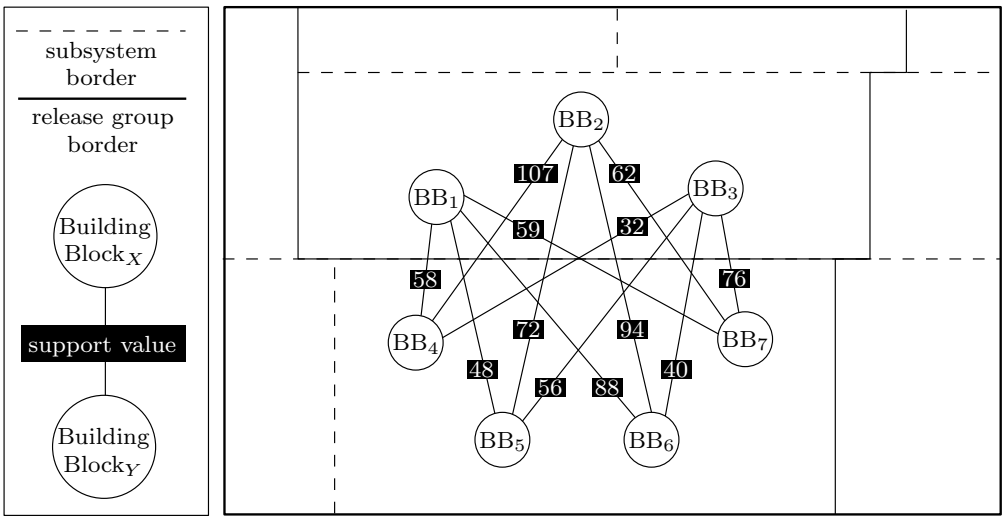

Figure 4.1: Illustration of the evolutionary cluster characterized

one real-life evolutionary cluster. Why exactly the properties described below are the ones which we propose is described later in Section 4.6.

Figure 4.1 illustrates one evolutionary cluster characterized in the context of the release group decomposition. The outer rectangle represents the software system. The solid lines illustrate the release group decomposition of the software system. Dashed lines further refine that decomposition by indicating the borders of subsystems. Recall that a release group is a set of subsystems. Crosses indicate those building blocks which are part of the evolutionary cluster characterized. So the evolutionary cluster contains seven building blocks from two subsystems. These subsystems are in different release groups. If a line connecting two building blocks (from different subsystems) is drawn, this indicates these building blocks were changed together; the number in the black rectangle indicates how often they were changed together.

Figure 4.1 does not show the borders of all subsystems in order to not clutter the picture. Also, the size of the rentangles has no meaning. The purpose is to highlight the dependencies in this evolutionary cluster in kind of a fish-eye view.

An evolutionary cluster such as the one depicted in Figure 4.1 is the result of (in general) many individual change sets. Not all of these change sets need to be identical (as evidenced by the different numbers in the black triangles in Figure 4.1). Some may involve more building blocks from each participating subsystem, others less. For instance, one change set may consist of building blocks $\{BB_2, BB_3, BB_7\}$, while another one just contains building block BB_5, and a third one contains all seven building blocks. For many properties, we therefore consider the maximum, minimum, average and the standard deviation of the property values over all change sets participating in the evolutionary cluster (as in Tables 4.1- 4.5). Since the number of building blocks in the set of change sets that defines an evolutionary cluster is from a relatively

small range, using the average is meaningful. Expressing the properties measured in terms of their minimum, maximum and average values is important to easily/automatically select those evolutionary clusters architects are interested in.

4.4.1 Property 1: Cluster Size

The size of an evolutionary cluster is the number of building blocks involved. The size therefore may be an indication of its complexity and changeability. The bigger the size, the more building blocks need to be changed together and the more complex the whole operation might be. In our study environment each building block contains 50 files on average. The evolutionary cluster we characterize in this section contains seven building blocks (BB_1-BB_7), see also Figure 4.1.

4.4.2 Property 2: Borders Crossed

Every building block in the evolutionary cluster can be identified using different decompositions. Every building block has its containing subsystem, development group, deployment group, release group and development site. A development group owns one or more subsystems and is responsible for the modifications introduced to those subsystems. A deployment group contains subsystems which need to be deployed independently. The independent release group consists of subsystems to be released independent of the rest of the system. As the observed industrial environment is multi-site, the building blocks are developed in different parts of the world.

The observed evolutionary cluster contains building blocks from two different subsystems and therefore the cluster crosses the borders of subsystems. These subsystems were part of different release groups, so the evolutionary cluster crosses the borders of independent release groups as well. See also Figure 4.1. As for the other decompositions, the observed cluster is located in a single decomposition element.

4.4.3 Property 3: Support Distribution

The support between two building blocks is the number of times those building blocks changed together, see [Zimmermann et al., 2003]. This property indicates the distribution of the support values for all the building block pairs $\{BB_a, BB_b\}$ in the cluster where BB_a and BB_b are from different decomposition elements. In this case, a decomposition element is a set of subsystems to be released independently.

This evolutionary cluster contains release group crossing relationships with a relatively high support, see Table 4.1. Building blocks from different release groups in this evolutionary cluster were changed from 32 to 107 times, and on average 66 times.

Table 4.1: Support Distribution

Support			
MAX	MIN	AVG	STD
107	32	66	21

4.4.4 Property 4: Jaccard Similarity Distribution

The Jaccard similarity expresses the probability of two building blocks changing together given that one of them gets changed, see also [Maqbool and Babri, 2007, Abreu et al., 2006]. The Jaccard similarity is a symmetric measure and it is a combination of the asymmetric confidence measures [Zimmermann et al., 2003] (number of common changes divided by the number of changes to one of the building blocks). This property measures the distribution of the Jaccard similarities of building block pairs from different decomposition elements.

To analyze the distribution, we measure the maximum, minimum, average and the standard deviation of the Jaccard similarity values. For the release group decomposition of our example the results are depicted in Table 4.2.

Table 4.2: Jaccard Similarity Distribution

Jaccard Similarity			
MAX	MIN	AVG	STD
24.71%	8.29%	15.42%	4.52%

4.4.5 Property 5: First Co-changes

We cannot only measure how many times building blocks changed together, but we can also observe when exactly they changed together for the first time. The date of the first co-change helps to understand since when the participating building blocks are related. This property captures, for a given system decomposition, the distribution of the first co-changes between building blocks from different decomposition elements. The first co-changes are indicated with the letter **F** in Figure 4.2.

The actual data for the subsystem decomposition is included in Table 4.3, showing the maximum, minimum, average and the standard deviation (given in days) of the first co-changes. As we can see from Table 4.3, all building blocks participating in the analyzed evolutionary cluster were first changed together on the same day. This statement happens to be true for the evolutionary cluster observed; it is most often, however, (in 70% of the evolutionary clusters analyzed in our study) not the case.

Table 4.3: First Co-change Distribution

First Co-change			
MAX	MIN	AVG	STD
15 July 2004	15 July 2004	15 July 2004	0 days

4.4.6 Property 6: Last Co-changes

Similar to Property 5, this property expresses, for a given system decomposition, the distribution of the *last* co-changes of building blocks from different decomposition elements. The last co-changes are indicated with the letter **L** in Figure 4.2. Table 4.4 shows the actual distribution values for the subsystem crossing relationships in case of the evolutionary cluster studied.

Table 4.4: Last Co-change Distribution

Last Co-change			
MAX	MIN	AVG	STD
30 Apr 2009	19 Aug 2008	28 Jan 2009	104 days

4.4.7 Property 7: Co-change Tendencies

Co-changes between two building blocks have a certain distribution over time. Figures 4.2 (a), 4.2 (b) and 4.2 (c) illustrate three such distributions. In each of the three cases thicker vertical line segments indicate co-changes between a pair of building blocks. The horizontal position of those segments is determined by when the co-change happened between the first co-change and NOW (the end of the time period analyzed). As indicated by Figure 4.2, the distribution of co-changes may show different tendencies:

1. Co-changes getting more frequent (Figure 4.2 (a))
2. Co-changes getting less frequent (Figure 4.2 (b))
3. Co-changes having a more or less stable frequency of occurrence over time (Figure 4.2 (c)).

Similar to the Yesterday's Weather approach of Gîrba et al. [2004], we consider the tendency of co-changes to get a better understanding of what may happen in the near future. More specifically, tendency observations may help us understand which unwanted couplings we are more likely to suffer from in the near future. To decide which of the described tendency types can be found in an evolutionary cluster we need to do some further measurements.

We determine the tendency type for two building blocks by mapping the period between the first co-change and NOW on a [-1,1] interval, see Figure 4.2. Then we determine the

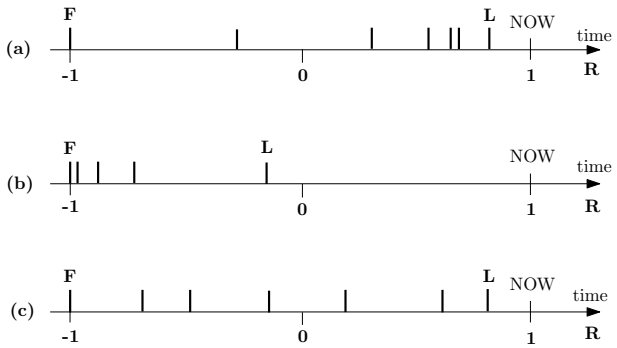

Figure 4.2: Three scenarios for co-change tendency between two building blocks

position of every co-change in this interval resulting in numbers t_1, t_2, ..., t_N where $\forall i \in [1..N]$: $t_i \in [-1, 1]$ and N is the number of co-changes between the two building blocks analyzed. The average of all the resulting numbers ($\frac{t_1+t_2+...+t_N}{N}$) is referred to as the *tendency number*. A tendency number near to -1 tells us that the co-change frequency decreased, a tendency number of near 1 shows that co-changes are getting more frequent over time and a value near to 0 indicates that co-changes are evenly distributed over time. There may be more sophisticated ways to measure the tendency of co-changes than how we do it, but we picked our solution because of its simplicity. This property describes, for a given system decomposition, the distribution of the tendency numbers between building blocks from different decomposition elements. Table 4.5 shows the maximum, minimum, average and the standard deviation for the tendency values for the release group crossing relationships in our example cluster.

Table 4.5: Co-change Tendency Distribution

Co-change Tendency			
MAX	MIN	AVG	STD
-0.16	-0.30	-0.23	0.04

4.4.8 Property 8: Static Relationships

With this property we count, for a given system decomposition, how many building block pairs from different decomposition elements are also coupled in terms of static relationships. The static relationships we considered in our study environment are include and call relations. These relationships can help to identify couplings between building blocks by just having a look at the content of the files in those building blocks. In case of the observed evolutionary cluster we found seven static relations crossing release group borders.

4.4.9 Discussion

In the previous subsections, we characterized one evolutionary cluster along a number of dimensions. The assessment whether this evolutionary cluster denotes an unwanted coupling depends on one's interests:

- The building blocks in the evolutionary cluster were mostly changed relatively recently. This might suggest they will again change in the near future.
- The building blocks all come from the same development group, so from that point of view the coupling is not unwanted.
- The size of the cluster we used as an example in this section (see Figure 4.1) is fairly large (7), compared to other clusters. From the size perspective, this cluster definitely indicates an unwanted coupling in our study environment. Note that in other environments the threshold for large clusters may be different.
- Not only is the size relatively large, also the frequency with which the building blocks from different subsystems changed together is quite high.

4.5 Evolution Anti-Scenarios

The characterization described in Section 4.4 can be applied to all the evolutionary clusters. As a result, we know for every evolutionary cluster exactly what properties it has. For instance, we know which clusters involve more than one subsystem and which of those have a high average support.

When evolutionary clusters indicating unwanted couplings are to be selected, we have to define a query on the properties identified. We can specify, for example, that we are interested in evolutionary clusters (1) crossing the borders of development groups, (2) having a high support distribution for the relationship crossing the borders of development groups, and (3) where the building blocks from different development groups have co-evolved much more frequently in the recent past than earlier in time.

What is considered unwanted depends on the architects or developers and their responsibilities. Once we know which dependencies experts are looking for, we are able to query evolutionary clusters that point to those dependencies. Architects however are not thinking explicitly in terms of evolutionary cluster properties when unwanted couplings are described. In our experience, architects tend to express what they consider unwanted couplings in terms of concrete scenarios. These scenarios are often coming from their past experience and are described in a domain specific way. Using scenarios is a known way to assess properties of software architectures [Bass et al., 2003]. The next paragraph describes one such scenario.

Developer D_A is a member of the development group G_A. G_A owns subsystem *A* and is responsible for introducing modifications to that subsystem. In a similar way, developer D_B from development group G_B owns subsystem *B*. One day, D_A changes a set of files and realizes that, as a consequence, files from subsystem *B* also need to be changed. So he calls

D_B at the other side of the world and asks him to introduce some modifications. As G_B is busy implementing other functionalities, D_B puts the request onto a priority list. After 10 telephone calls from D_A and after one month of time has elapsed, the required files in subsystem *B* are modified. Altogether it took 40 man-hours to introduce the change initiated by D_A. Changes which are similar in complexity but involve only G_A cost usually 4 man-hours.

The above scenario is an example of what we call an *evolution anti-scenario*, analogous to the notion of design anti-patterns [Feng et al., 2004]. Such a scenario describes what architects do *not* want to happen during the evolution of the software system. Evolution anti-scenarios are usually kept implicit in the heads of the architects. Interviews can be used to elicit the implicit knowledge of anti-scenarios. During these interviews, documents describing which unwanted couplings an architect resolved or plans to resolve may help to get (or derive) the explicitly described anti-scenarios. The extracted anti-scenarios can then be used to understand which type of coupling the architects and developers consider to be unwanted. Understanding which couplings are unwanted is the result of an iterative approach rather than the result of a one shot activity.

4.6 Property Justification

Having the evolution anti-scenarios collected we need to translate them to one or more queries on the collection of evolutionary clusters. This puts an additional requirement on the characterization. The properties (size, borders crossed, etc.) we use to characterize clusters should allow us to query evolutionary clusters and identify the types of unwanted couplings described by the scenarios. In this section, we argue for every property described in Section 4.4, that they should be included in the characterization. These properties need not form an exhaustive set. In this section, we use real-life evolution anti-scenarios from the study environment to support our arguments.

4.6.1 Property 1: Cluster Size

Let us consider the following scenario. In a large software development company it is quite common that experienced developers are leaving, while others with little or no domain knowledge are entering the organization. Developer *D* is such a newly employed developer. *D* gets involved with his new project and his project leader assigns *D* the task to modify the database schema. *D* modifies the database schema and some building blocks which he thinks are affected by the change. Being new and having a lack of domain knowledge, however, *D* forgets to modify another ten building blocks from different decomposition elements. As a consequence, the test of the software system fails. Such a scenario costs the company a lot of effort (in time and money).

The lesson we can learn from the above evolution anti-scenario is that the more building blocks are involved, the more difficult it is to maintain consistent changes. If the project leader

Even though the subsystems are owned by different development groups, dependencies may arise – and show up in change sets – for instance through the common use of interface files.

knows the size of evolutionary clusters, he may assign a task to *D* which involves smaller evolutionary clusters and is therefore less complex.

Huge evolutionary clusters, however, are problematic even for experienced developers. Experts could therefore select huge evolutionary clusters and try to reduce their size by making the involved building blocks less dependent, for instance by moving building blocks to another decomposition element.

On the other hand, in some cases small evolutionary clusters are the ones which are important. For instance, if the goal is to resolve as many unwanted couplings as possible from a fixed budget, then evolutionary clusters which indicate cheap-to-resolve unwanted couplings are probably the target for the resolution activities. The effort to resolve an unwanted coupling is likely to be influenced by the size of the evolutionary cluster. The bigger the evolutionary cluster is, the more building blocks are related and the more effort it may take to resolve the unwanted coupling related.

4.6.2 Property 2: Borders Crossed

Let us reconsider the evolution anti-scenario described in Section 4.5. In that scenario, building blocks from different development groups and development sites had to be changed together. Compared to the changes which are local to development groups, these types of changes tend to be more costly. The increase in costs is caused by the additional communication costs and the costs caused by the fact that one development group has to wait for another one.

Certain architects may be responsible for reducing the above costs by carefully assigning building blocks to development groups. In order to assess and to improve the current structure, these architects have to know which building blocks from different development groups changed together. In this case, the architects in charge are less interested in evolutionary clusters not crossing the borders of these groups. If an evolutionary cluster is not crossing a border of development groups it may still cross the borders of independent release groups, for instance. Those evolutionary clusters in turn may be relevant for architects responsible for the release group structure.

Typically, all of these various border-crossing relationships are important. It is a multi-dimensional separation of concerns and a careful weighing of concerns is required to resolve the associated unwanted couplings.

4.6.3 Property 3: Support Distribution

A description of an evolution anti-scenario might be as follows. In a software company new projects were started last year, to develop new releases of their software product with improved and new features. During this period certain building blocks were modified much more frequently than others. We observed that one set of building blocks X from subsystem S_X changed together with a set of building blocks Y from subsystem S_Y 200 times during the last year. With this number of co-changes, X and Y are the building block sets which changed together most often in the last year.

In the above scenario the co-change of X and Y may indicate an unwanted coupling. Even if changing those sets of building blocks together is not the most costly operation, changing them together 200 times makes the underlying coupling the most costly one during the last year of development. In case development activities will touch upon the same subsystems the next year, architects may want to resolve unwanted couplings similar to the one indicated by the co-change of X and Y. Usually, architects are interested in the outliers when considering the support distribution.

4.6.4 Property 4: Jaccard similarity Distribution

Let us consider the following scenario. The architect decides that subsystem S needs to be outsourced, and his decision is approved by management. The decision is taken in order to reduce development costs. Company C_O takes over the development of S. Although S is outsourced, it is still actively connected to the rest of the system: very often, a change in S requires a change in the rest of the system and vice versa. After a year of co-operation it turns out that the communication and the increased management costs due to the poor isolation of S exceeded the original development costs. Consequently, a decision is taken to in-source S again.

When looking for unwanted couplings between the outsourced subsystem and the rest, the relative distribution of single-site changes as opposed to site-crossing changes, i.e. the Jaccard similarity distribution, may play a role. If, relatively speaking, the number of local changes is large (i.e. the Jaccard similarity is low), this may warrant keeping the subsystem outsourced, for financial reasons, or because of local expertise. If, on the other hand, the Jaccard similarity is high, communication and collaboration cost may become a bottleneck, and one may decide to resolve this dependency.

4.6.5 Property 5: First Co-changes

The first co-change between two building blocks determines since when they were changed together. Building blocks changing together since last month may indicate a less severe dependency than building blocks which evolved together during the last four years. Four years of continuous co-change tells us that the reason why they are changing together is still not resolved. On the other hand, building blocks which only changed together recently may have done so, for instance, because of the introduction of a new feature. In the software system we observed it happens frequently that, after a new feature introduction, it takes a while before the concerns are well separated. Co-changes during that initial period need not indicate an unwanted coupling.

4.6.6 Property 6: Last Co-changes

During the development and maintenance of a software system unwanted couplings are continuously resolved. As a result we often see from the development history that building blocks

changed together a long time ago but, after a while, stop doing so. Therefore, evolutionary clusters containing building blocks where the last co-changes happened a long time ago probably indicate (1) an already resolved dependency or (2) stable couplings between building blocks with respect to evolution. Therefore, those type of evolutionary clusters are not interesting to the architects and need to be filtered out.

4.6.7 Property 7: Co-change Tendencies

Building blocks which changed together a lot but the frequency of co-change has become very low may not be interesting to the architects because at present those building blocks are less likely to change together. On the other hand, building blocks which have been changed together periodically, let's say every month, will most probably change together in the next month if we do not do anything against it. Therefore it is important to measure the tendencies of co-change frequencies between building blocks.

4.6.8 Property 8: Static Relationships

One evolution anti-scenario runs as follows. It becomes important to separate the evolution of subsystem S_1 from the evolution of subsystem S_2. To reach that goal, evolution-type dependencies between S_1 and S_2 are identified using static relationships only. The reason is that static relationships are easy to determine from the source code. Furthermore, static relationships are also the kind of relationships which software developers tend to recall, and resolve, most easily. After resolving those issues, S_1 and S_2 still cannot be evolved as independent as the architects would desire it, mainly because the testing and communication costs involving both subsystems are still too high. After a long and costly investigation the non-static relationships were found to cause the two subsystems to change together.

The evolution anti-scenario just described teaches us that there are situations where evolutionary clusters that exhibit a low number of static relationships crossing the borders of decomposition elements are still important. One of the reasons is that an evolutionary cluster where the related unwanted coupling is caused by static relationships is likely to be already known and therefore its investigation may bring no additional information. Another reason is about the costs of unwanted couplings caused by non-static relationships. The experts we talked to agreed that around 10% of the unwanted couplings are not caused by static relationships. Those are the unwanted couplings which remain unknown for a longer time because of the difficulty to identify them and therefore they are considered to be more harmful, i.e. costly, than other unwanted couplings.

4.7 Querying Evolutionary Clusters

In order to find evolutionary clusters which point to unwanted couplings we need to create and execute a query on the set of evolutionary clusters. Such a query has to result in evolutionary clusters where the corresponding dependencies are considered unwanted by a specific architect

or developer. To create the query, we use as input the evolution anti-scenarios, see Section 4.5, elicited from the architect or developer we want to support. We translate the anti-scenarios to queries together with the software architect during formal meetings. During such a meeting an architect needs to get familiar with the properties used for the characterization and the effects of using those properties in the query. In our case study, it took us approximately one hour to agree on the precise formulation of the query. In the next subsections, we describe three real-life cases when the elicited evolutionary scenario(s) were translated to queries. In sections 4.7.4 and 4.7.5, we discuss the gains of using anti-scenarios and the corresponding queries to filter out unwanted couplings in these cases.

4.7.1 Case 1

In the first case, the architect had the task to make sure that the development groups depend as little as possible on each other. To identify the evolution anti-scenarios related to this case we organized a meeting with the architect. We started that meeting with asking the architect to recall which were so far the most severe development issues he had to face caused by unwanted couplings. The evolution anti-scenarios which we elicited from the architect in this case describe (1) a large amount of communication between development groups and (2) delay of work of some development groups because they have to wait for other development groups. One such scenario from the architect is the following:

> *"At some point in time we decided to outsource subsystem [A] to development site [DS]. We believed that the couplings between subsystem [A] and the rest of the system were low enough to make the outsourcing successful. However, the amount of communication with development site [DS] increased drastically, overruling the benefits of outsourcing. This was a costly lesson to learn."*.

In a second meeting with the software architect we created a query on the evolutionary clusters. Since this was the first such exercise for the architect, we started by explaining the properties we used for characterizing the evolutionary clusters, see Section 4.4. During the one-by-one explanation of the properties we made it possible for the architect to ask questions about the properties. This way we tried to ensure that architect really understood what the properties were about. The architect asked questions like: *"What input data do you use to identify co-changes?"*. Yet another time (after the architect had a rather good knowledge about the properties) he started to suggest improvements, like: *"You could think about grouping co-changes together which happened relatively close in time in order to have a better way of expressing change tendency"*. Next, we discussed with the architect which of the characterization properties had to be used in the query, and which thresholds to use.

First we discussed the cluster size property. If more than one development group is involved in a change, then the number of building blocks involved, and therefore the complexity of the change, does have an influence on how much communication effort is spent. Furthermore, more building blocks typically take more time to modify and in such cases development groups

For proprietary reasons, the actual subsystem and development group names are replaced in the quote.

may delay other development groups for a longer time. Therefore we chose to include the size of the evolutionary clusters in the query.

Next we looked at the border crossing properties of evolutionary clusters. It was straightforward to decide to select the evolutionary clusters crossing the borders of development groups, since the architect was interested in the unwanted couplings of the proposed development group structure and not in the unwanted couplings of other structures.

We continued with a discussion of the support property. The support property was included in the query because we wanted to identify unwanted couplings that indicate frequent communication between development groups. The more frequent development groups need to communicate, the more effort on communication needs to be spent.

The Jaccard similarity was the next property we discussed. Evolutionary clusters with a low average Jaccard similarity are a consequence of many changes that are local to a development group, and few changes that involve more than one development groups. Such evolutionary clusters therefore denote the correctness of the development group decomposition rather than an unwanted coupling. Consequently, we decided to filter out evolutionary clusters having a low Jaccard similarity average.

Next we looked at the first and last co-change and co-change tendency properties. We explained the architect that if building blocks last changed together a long time ago, e.g., the last co-change between those building blocks happened 5 years ago, then this probably indicates an already solved issue. According to the architect this was a safe assumption, since normally parts of the software system never stay unmodified for such a long time.

We also explained to the architect that even if the first co-changes of building blocks happened relatively recently (a few weeks ago), the frequent co-changes of those building blocks would not provide us with enough historical evidence to say that we are facing an unwanted coupling. The architect agreed that using the first co-change property in the query is indeed helpful. According to his line of reasoning, an initial feature development often requires more than one development group to create and change building blocks. Later on, the responsibilities of those development groups with respect to that feature become more stable and they can work relatively independently from then on. Such a work pattern results in many co-changes only for a short period of time. If the first co-evolutions happened recently we do not know (yet) if we are looking at such a development pattern or not.

The architect was very positive about the idea of using the co-change tendency property. According to him, knowing if the co-changes are getting stronger or weaker is important to decide which unwanted couplings are more severe than others. When explaining the tendency property to the architect we showed him a few examples of building blocks changing together over time. After looking at those examples the architect expressed that he is most interested in those unwanted couplings where the co-changes of the participating building blocks occur periodically. This translates to a tendency number close to zero. The architect also argued that those cases where the co-changes got more frequent, i.e. the tendency number is positive, are less relevant since the frequent co-changes are only recently observed.

After having discussed all properties, and having decided which ones to include, we set the threshold values for the properties selected. This again involved several iterations. We started

Table 4.6: Selection Criteria for Case 1

PROPERTY	CRITERION
Cluster Size	≥ 3
Borders Crossed	="Development Groups"
Support AVG	> 50
Jaccard Similarity AVG	$> 20\%$
First Co-change MIN	more than 2 months ago
Last Co-change MAX	less than 2 years ago
Co-change Tendency AVG	$>$ -0.15
Static Relations	–

with an excel sheet containing all clusters that had a Jaccard similarity level > 0.05. This excel sheet contained 178 of the 595 evolutionary clusters. The columns of the sheet contained the minimum, maximum, average and standard deviation for all properties. Browsing this sheet gave the architect a feeling for the distribution of the various properties. For example, the size of the clusters ranged from 2 to 42. Based on these ranges, the architect next decided upon the thresholds to be used. The order in which the thresholds were decided upon reflect their relative importance to the architect. For instance, he considered the co-change tendency more important than the first co-change numbers. In the case of co-change tendency the architect first opted for the value 0.2. This is a relatively high tendency number which also expresses his need: to find unwanted couplings where the co-changes are occurring periodically. After executing the query with that tendency number threshold and analyzing the sheet with the resulting evolutionary clusters, we found however that the threshold set was too strict: the architect also wanted to find unwanted couplings with a somewhat lower threshold. Therefore we modified the threshold to -0.15 and reran the query. The resulting sheet contained some additional evolutionary clusters pointing to couplings the architect deemed unwanted. This example shows that multiple runs of the query are needed to get a good feeling about which thresholds to use. The final query on the evolutionary clusters is shown in Table 4.6.

As we can see, there was no filtering criterion set on the static relations. The reason for that is that the architect wanted to have an overview of the unwanted couplings independent from whether they are easy to identify from the static relationships or not.

4.7.2 Case 2

In the second case we consider an architect who had the task to make release groups more independent. The final goal was to release new versions of every release group on their own, without mutual dependencies. As a first step, static relationships were analyzed to know at which points release groups are coupled. This step was thought to be relatively cheap and effective, but looking only at static relationships did not allow the architect to identify all the

Table 4.7: Selection Criteria for Case 2

PROPERTY	CRITERION
Cluster Size	–
Borders Crossed	=”Release Groups”
Support AVG	> 20
Jaccard similarity AVG	$> 20\%$
First Co-change MIN	more than 2 months ago
Last Co-change MAX	less than 2 years ago
Co-change Tendency AVG	$>$ -0.15
Static Relations	$= 0$

unwanted couplings. An additional argument was that co-changes without static dependencies are much more costly than the ones caused by static dependencies. Therefore the architect was interested in couplings where the unwanted co-changes were *not* caused by direct static dependencies. This can be the case for instance if two building blocks implement one and the same data model. This then introduces a semantic dependency between those building blocks which is likely to show up in changes involving that data model. The evolution anti-scenario in this case describes that building blocks from different release groups cannot be released independently although static dependencies are removed. Based on this scenario we created the query on the evolutionary clusters with the criteria described in Table 4.7. Since we have created the query with the same architect involved in Case 1, we did not have to explain the characterization properties again.

We determined the thresholds for the characterization properties very similar to how we have done it in the previous case. First, we considered only the evolutionary clusters where there were no static relationships between the building blocks included. Second, the release groups were of interest rather than the development groups. Third, we defined no restriction for the cluster size and we specified a lower threshold for the support than in the previous case. The reason for these lower thresholds is that the developers working on the system analyzed know that there are far fewer unwanted couplings caused by non-static dependencies. It is important to the architect to know of evolutionary clusters even if the building blocks changed less frequently together or the cluster contains fewer building blocks.

4.7.3 Case 3

In the previous two cases the evolutionary clusters characterized were queried to *identify* unwanted couplings. Knowing the unwanted couplings, however, is only the starting point for architects to improve the structure of the software system. After identifying an unwanted coupling an architect needs to

- understand why the unwanted coupling exists

- identify the alternatives to solve or mitigate the unwanted coupling
- decide upon a solution to be implemented
- carry out the structural changes suggested by the solution chosen

Querying the evolutionary clusters characterized can be used beyond the identification of unwanted couplings. It may also help in the subsequent process when software architects have to decide upon an alternative solution. This is an important lesson we learned when we, together with the software architect and several developers, applied a what-if type of analysis on one unwanted coupling that resulted from executing the query from Case 1 (Section 4.7.1).

The what-if type of analysis we applied can be executed if a solution alternative proposed by the experts is about moving some software entities, i.e. files, building blocks, or subsystems, from one decomposition element to another. In our case, such alternative solutions were suggested related to half of the evolutionary clusters analyzed with the experts. The analysis is necessary if the software architect is not sure about the consequences of implementing an alternative solution for an unwanted coupling. Our what-if type of analysis consists of the following steps:

- Altering the paths of the software entities which were to be relocated. The paths have to be changed in the versioning meta-data used to create the evolutionary clusters. As a result we have a historical data set which mimics the situation where the software entities to be relocated reside in their new decomposition element.
- Re-running the approach used to identify evolutionary clusters on the data set resulting from the previous step.
- Applying the same query on the newly identified evolutionary clusters.
- Analyzing the difference between the original and newly queried evolutionary clusters. During such a comparison we are especially interested in knowing whether the software entities moved are participating in any of the newly identified evolutionary clusters (new unwanted couplings would be introduced). Also we need to know if the original evolutionary cluster analyzed is still present in some form in the new set of evolutionary clusters (i.e., the coupling is not (completely) solved).
- Presenting the results to the architect such that he can take them into account when deciding about the solution alternative in discussion.

In the rest of this section we elaborate further on the actual instance of the what-if type of analysis which we executed in our study environment for Case 1, see Section 4.7.1.

The unwanted coupling investigated involves C source files from one subsystem and C header files from another subsystem. The file level analysis revealed that source files had been changed together with header files very frequently (more than 20 times). As those subsystems were supposed to be maintained by different development groups, the frequent co-changes between the source and header files pointed out a costly and, therefore unwanted, coupling.

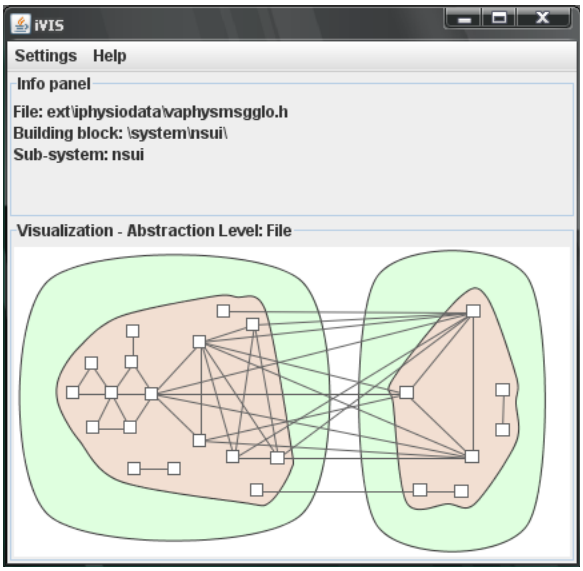

Figure 4.3: An unwanted coupling visualized

For the file level analysis of the evolutionary cluster we used our interactive visualization tool iVis, see also Chapter 5. Figure 4.3 shows a snapshot of the tool while it is visualizing the evolutionary cluster under discussion. The tool presents software entities (subsystems, building blocks, files) and the co-changes between those entities. The outer blobs in Figure 4.3 are representing the two subsystems the C source and header files are coming from. The tool was configured such that it only shows a line between files (represented by white squares) if they were changed together at least 20 times. As we can see, there are many strong relationships crossing the borders of subsystems and therefore contributing to the unwanted coupling. Further information on the tool can be found in Chapter 5.

Meetings organized with the architect and some of the developers involved in the development of the C files under discussion helped us identify the underlying reason for the unwanted coupling. Originally, both the source and the header files were designed to be in the same subsystem but a recent restructuring activity had put them apart. It has been acknowledged by the developers to be a mistake made during the restructuring.

The developers proposed a solution to the architect to solve the unwanted coupling. They suggested moving both the source and the header files under discussion to a third subsystem. The architect had doubts about the adequacy of the solution proposed. He expected that moving the source and header files involved in the unwanted coupling would give rise to new unwanted couplings related to those files. Therefore, the architect wanted to know if moving the files to their proposed location would indeed introduce new unwanted couplings and if the coupling discussed would be resolved. Knowing the answers to these questions was a key issue for the

architect to decide upon the acceptance or rejection of the solution alternative proposed by the developers.

To help answer the questions of the architect we redefined the location of the source and header files under discussion as if they had been moved according to the suggestion of the developers. We have done that by changing the paths of the relevant files in our database containing the historical meta-data from ClearCase. Using the modified input we identified, characterized and queried the evolutionary clusters the same way we had done before in Case 1, see Section 4.7.1. By comparing the newly selected evolutionary clusters to the originally selected ones we made the following observations:

- None of the source and header files under discussion are part of the newly selected evolutionary clusters.
- None of the building blocks originally containing the discussed source and header files are part of the newly selected evolutionary clusters
- The set of newly selected evolutionary clusters is the same as the one before the relocation of the source and header files except for the evolutionary cluster with the source and header files.

Our first observation indicates that moving the source and header files would not result in new unwanted couplings related to the files moved. The second observation points out that following the suggestion of the developers would indeed solve the unwanted coupling between the original building blocks. Our third observation reflects on the stability of the approach we use to identify evolutionary clusters. Knowing these results the architect accepted the proposal of the developers and as a result a problem report (PR) was issued to implement the necessary changes.

4.7.4 Comparing Selection Approaches

How much did we actually gain by using a more sophisticated selection of evolutionary clusters, as compared to the selection mechanism suggested by previous work? To answer this question we measure how many evolutionary clusters previous approaches and our approach sort out from the complete set of evolutionary clusters identified. The better we can retrieve the evolutionary clusters which the supported architect acknowledges to point to unwanted couplings, the better our approach is. We use Case 1 described in Section 4.7.1 during the comparison.

Altogether, we identified 595 evolutionary clusters. A vast majority of the related work dealing with the identification of unwanted couplings, see [Fischer and Gall, 2006, D'Ambros et al., 2009, Ratzinger et al., 2005a, Beyer and Hassan, 2006], suggests that software entities (like building blocks in our case) from different subsystems may point to an unwanted coupling if those entities changed together frequently. Using the terms of our characterization it means that the interesting evolutionary clusters are crossing the borders of subsystems (or modules) and that the co-change relationships between entities from different subsystems have a high

support. We denote those commonly used criteria (support and borders crossed) as *Default*. If the architect had specified only these commonly used filtering criteria, 68 from a total of 595 evolutionary clusters would have been selected.

Our experience shows that analyzing whether an evolutionary cluster is an unwanted coupling or not takes a considerable amount of time (a couple of hours at least), even if we reject the cluster at the end. Our experience is also that an architect has a very limited time and cannot spend weeks on a single task. Analyzing 68 evolutionary clusters would have required too much time from the architect.

Some earlier work extends the common way to find entities participating in unwanted couplings. Next to the subsystem crossing and support properties, German [2006] and Zimmerman et al. [2003] consider the confidence between entities as well, very similar to what we do with the Jaccard measure. Recall that the Jaccard measure (Section 4.4.4) is a combination of confidence measures. We refer to the criteria used here as *Default + Jaccard*. Applying this additional filter results in 25 evolutionary clusters. Antoniol et al. [2005] are using the size of the evolutionary clusters together with the support and subsystem crossing properties to filter entities changing together and find unwanted couplings. The criteria used by Antoniol et al. is further referenced as *Default + Size*. In Case 1, querying evolutionary clusters based on these three properties results in 36 evolutionary clusters.

In Case 1 we needed seven selection criteria to express which evolutionary clusters the architect was interested in (see Table 4.6). Using these seven criteria in the query results in 12 evolutionary clusters. We used the interactive visualization approach to further analyze with the architect and developers the topmost 10 of those 12 evolutionary clusters selected, see also [Vanya et al., 2010]. Each of these clusters pointed to an unwanted coupling as acknowledged by the developers. As compared to the common way to find unwanted couplings, we have eliminated 56 evolutionary clusters (82 %). The architect was not interested in these clusters because, for instance, they were likely to point to an already solved dependency (when, e.g., the last co-change happened more than 2 years ago) or too little evidence has been gathered about the files changing together (when, e.g., the first co-change happened less than 2 months ago). Similar to Case 1, we executed the query in Case 2 as well. That query also results in a small number of evolutionary clusters, namely ten.

By querying evolutionary clusters we identify a subset of them. Analyzing the selected evolutionary clusters, however, is typically a *sequential* process where those clusters are analyzed one after the other. Furthermore, an architect may have a preference to analyze some evolutionary clusters sooner than others. This implies that evolutionary clusters need to be prioritized when we present them to software architects. Many of the related work suggest to base this prioritization on the number of co-changes, i.e. on the support. Also, the architect we worked with wanted to analyze the evolutionary clusters in decreasing order of their support. Such a sorting makes it possible to compare approaches selecting evolutionary clusters.

The criteria we used in Case 1 and Case 2 to query evolutionary clusters form a superset of the criteria used by previous work. Therefore, the evolutionary clusters we selected in Case 1 and Case 2 form a subset of the evolutionary clusters selected with the criteria of the previous work. We can then observe where the evolutionary clusters selected in Case 1 and

Table 4.8: Priority mapping for Case 1

Case 1	Default	Default + Size	Default + Jaccard
1	1	1	1
2	2	2	2
3	9	5	7
4	34	17	12
5	36	19	13
6	37	20	14
⋮	⋮	⋮	⋮

Table 4.9: Priority mapping for Case 2

Case 2	Default	Default + Size	Default + Jaccard
1	4	2	3
2	9	5	4
3	19	9	7
4	22	10	10
5	29	16	13
6	37	18	15
⋮	⋮	⋮	⋮

Case 2 end up in the lists of evolutionary clusters we get by querying the clusters using the selection criteria of the previous work and sorting them by support. Tables 4.8 and 4.9 show the results of the comparisons. During the comparison we used the three groups of related works mentioned, where the criteria to filter evolutionary clusters are: *Default*, *Default + Size* and *Default + Jaccard*.

From Table 4.8 we can see, for instance, that the third evolutionary cluster which the architect analyzed in Case 1 would have been the ninth cluster analyzed using the default criteria for querying the clusters. Remarkable is that the architect who was interested in the evolutionary clusters queried in Case 1 should have analyzed 34 - 4 = 30 clusters, queried using the default criteria, before he would have found the fourth cluster he was actually interested in. We can also see that using the size and Jaccard similarity properties in the query further helps to focus on relevant clusters but the number of evolutionary clusters rejected before a next relevant one is found can still be as large as 17 - 5 + 1 = 11 (with Size) or 12 - 7 + 1 = 4 (with Jaccard

similarity). Similar results hold for Case 2, see Table 4.9, as well.

4.7.5 Discussion

Again, we want to emphasize that we do not claim to find *the* unwanted couplings exclusively. We claim that by using our approach architects can better handle/reduce the huge amount of clusters which previous work generates. Of course, software architects may need more information (deeper analysis) than just the result of the query to make a final decision on the severity of an evolutionary cluster.

We expect that in most cases one will use the MIN value for the first co-change, MAX value for the last co-change, and the AVG values for the other properties. However, knowing the distribution of values allows us to detect, and possibly remove, outliers. Showing the distributions of the characterization property values to the architect is a fundamental step in the process we applied for setting the property thresholds in the query on evolutionary clusters. This process presumes that the architect has a good knowledge about the characterization properties. To make sure this is the case, we explained the architect the meaning of those properties, in advance. Based on this knowledge, the architect executed iterations in the process. In every iteration, the architect first observed the distribution of the values for the next most important characterization property. The architect determined the importance of a property based on, besides others, the evolution anti-scenarios extracted. Knowing the distribution of the observed property, the architect set the initial threshold for that property. When the query with this threshold resulted in too few or too many evolutionary clusters, the architect decided to change the threshold again and rerun the query. The process continued with setting the threshold for the next most important characterization property. The process finished when the architect set the threshold for all the characterization properties he believed to be relevant to query evolutionary clusters pointing to unwanted couplings. We need to do further experiments to obtain better guidance on how to obtain the proper selection criteria to use.

It is not guaranteed that the initial set of properties as defined in Section 4.4 is complete enough to express what architects consider to be an evolutionary cluster pointing to an unwanted coupling. Therefore, it may be necessary to extend the set of properties used for characterization and/or to have a deeper analysis of the evolutionary clusters selected.

4.8 Generalization

We have executed the characterization of evolutionary clusters in our study environment. Furthermore, we have characterized and queried the evolutionary clusters of the open source project ArgoUML to gain further insight into the generalizability of our characterization approach. We have assumed relatively little about the study environment and therefore our approach can be re-applied in many other environments as well. In this section we elaborate for every step of our approach which are the major points of concern one may need to think about when executing our approach in a different environment.

Our approach works on multiple input sources. These are the following:

1. the meta-data on changes extracted from the version management system
2. the evolutionary clusters containing software entities which frequently changed together
3. the decompositions of the software system to be evaluated and improved
4. the abstraction level of interest for the software entities in the evolutionary clusters.

The change meta-data we used from a version managements system about who changed what and when is widely available. This is even true in those environments where early version management systems, like CVS, are still used. Based on this generally available input the evolutionary clusters can be constructed, for instance, with the approach we described in our earlier work, see [Vanya et al., 2008]. In case of ArgoUML, which is relying on the Subversion version management system, change meta-data were captured explicitly, so no approximations were needed.

We extracted the development history of the first 8 years of ArgoUML (the same period covered in [Beyer and Hassan, 2006]). We only considered the Java classes, and identified major subsystems such as the `Explorer`, `Model`, and `Diagram`. ArgoUML is a much smaller system than the Healthcare system discussed in this chapter. The three considered subsystems have only about 700 Java source files. No intermediate layer such as the building block layer was needed in this case. We applied the process sketched in section 3.4.3 to the version data of ArgoUML.

Our approach is used to identify unwanted couplings in one or more decompositions of the software system. The approach assumes that the decompositions to be assessed are known. Our approach only assumes that the decompositions partition the software entities of the system into decomposition elements. In that sense our approach works with any kind of system decomposition. When working on the ArgoUML data we used the `Model`, `Explorer` and `Diagram` subsystems as a system decomposition.

The abstraction level of interest for the software entities in the evolutionary clusters, e.g. building blocks in our study environment or files in the case of ArgoUML, can be anything as long as we can observe when they changed together. Selecting a low abstraction level only poses computational challenges on the evolutionary cluster identification because of the typically huge number of software entities.

After acquiring the input data the next step is to characterize the evolutionary clusters. This step can be done in an automatic way, using an algorithm. Therefore, executing this step is straightforward and applicable in any environment where the previously described input data is available.

Querying evolutionary clusters is the last step to characterize the evolutionary clusters. How this step is executed may vary depending on the environment we are considering. The major point here is that we need to contact the architect (or someone else in the organization) knowledgeable about the evolution anti-scenarios and with whom we can translate those scenarios to queries on the evolutionary clusters.

For the ArgoUML data we identified 675 evolutionary clusters. We have characterized those evolutionary clusters the same way as the ones we have identified in our study environ-

Table 4.10: Priority mapping for the ArgoUML case

Case 2	Default	Default + Size	Default + Jaccard
1	8	6	3
2	18	11	10
3	21	19	12
4	26	22	16
5	36	29	21
6	44	32	25
7	55	37	28
8	61	40	32

ment. Furthermore, we used the same scenarios as in Case 1 (see Section 4.7.1) to query the evolutionary clusters. In case of ArgoUML, the scenarios applied remain hypothetical since they are not based on discussions with software architects of the system. However, being able to apply those scenarios also to the ArgoUML data reinforces that our approach is generalizable to other environments as well.

Querying the ArgoUML evolutionary clusters this way resulted in 8 evolutionary clusters. Similar to Table 4.8 and Table 4.9, Table 4.10 indicates where the evolutionary clusters we queried based on our characterization would end up in the ordered list of evolutionary clusters previous approaches would propose to investigate.

We have not further analysed the evolutionary clusters queried. However, Table 4.10 suggests that, similar to Case 1 (see Section 4.7.1) and Case 2 (see Section 4.7.2), our characterization approach is more efficient in identifying unwanted couplings than previous approaches.

4.9 Threats to Validity

In this section, we list possible limitations to our experience report by discussing the internal and external validity, following [Kitchenham et al., 2002] and [Perry et al., 2000]. Internal validity relates to the extent to which the results of our case studies may have been biased by confounding variables and other sources of bias. External validity relates to the extent to which any conclusions can be generalized to settings outside that of the current study.

With respect to internal validity, we discuss the following threats:

- we had no complete freedom in the selection of cases to explore,
- we used the opinion of one architect only,
- the results may be sensitive to the threshold values used, and

- the accuracy of the evolutionary clusters identified depends on the accuracy of the identified co-changes

Our study is not a controlled experiment. In particular, we were not able to pick a random set of anti-scenarios and corresponding queries, and evaluate the resulting collection of unwanted couplings. We had to use the anti-scenarios that had top priority for the architect. Only then could we involve him in our study. In [Harel, 2009], David Harel reflects on his early work on Statecharts, and observed: "One of the most interesting aspects of this story is the fact that the work was not done in an academic tower, inventing something and trying to push it down the throats of real-world engineers. It was done by going into the lion's den, working with the people in industry. [...] If what you come up with does not jibe with how they think, they will not use it. It's that simple." Our approach is similar. We worked together with a real architects, on real issues he is concerned about, and describe our experiences therewith.

The architect we worked with was confident that the evolutionary clusters filtered out are not pointing to unwanted couplings. It may however be the case that in spite of the belief of the architect some of the evolutionary clusters filtered out still point to unwanted couplings. That is, we do not know whether there are any false negatives. Checking for false negatives was not possible for us because, for instance, analyzing all the 68 evolutionary clusters in Case 1 would have taken too much effort from the architect. On the other hand, this observation also underlines the need for a sophisticated characterization, like the one we described in this article. The fact that our characterizations did not include any false positives helped to convince the architect of the value of our approach.

Based on the distributions presented, the architect determined the property thresholds we applied during the query of the evolutionary clusters. Adjusting the properties slightly may result in a different set of evolutionary clusters being selected. We need to do further studies to learn about the sensitivity of the different property thresholds. So far we have not executed such a study; it is part of our future work.

The accuracy of identifying co-changes between software entities may have an effect on the measured evolutionary cluster properties. In an extreme case, a single wrongly identified co-change may have a direct effect on the MAX and MIN measures. To maximize the accuracy of the evolutionary cluster characterization, we have to make sure that the co-changes are identified as accurately as possible. The latter we studied separately [Vanya et al., 2011].

With respect to external validity, we discuss the following threats:

- we worked with one architect only, and
- we applied the approach in one environment only.

In order to create the queries on the evolutionary clusters characterized we had meetings with one architect only. During those meetings we have also learned that other architects may consider the severity of couplings differently, because of their different responsibilities and/or actual interests. This is also reflected in the differences between Case 1 (Section 4.7.1) and Case 2 (Section 4.7.2). In order to present more solid benefits of our characterization approach we should contact more architects. However, the architect we worked with did have a complete

overview of the software systems' architecture and its issues. This is because the architects of the software system analyzed closely work together and share their concerns.

We have executed our complete approach in one study environment only. Our results from that highly competitive, industrial environment are promising. It was relatively straightforward to also characterize and query the evolutionary clusters of ArgoUML. However, we did not have the possibility to execute the whole approach and compare the results from different environments. The generalizability of our approach has been discussed in Section 4.8. To be able to better generalize our results, further experiments with our approach, in different environments, are needed.

4.10 Related Work

Finding unwanted couplings in a software system is a goal many studies have addressed [Vanya et al., 2008, Ratzinger et al., 2005a, Lungu and Lanza, 2007, Beyer and Hassan, 2006, Kuhn et al., 2005]. The structure of a software system has to fulfill a set of requirements in order to support the development and maintenance activities. For instance, semantically related files should be in the same subsystem to help developers achieve consistent modifications. Another requirement often posed on the structure of the software system is that static relations should exist between files from the same subsystem rather than between files coming from different subsystems. Depending on which requirement posed on the system's structure is addressed, related work looking for unwanted couplings can be categorized. For our current work those studies are related, and therefore relevant, which assess the independent evolution of decomposition elements, like subsystems. Such studies identify which software entities changed together and to which extent. We used their results as an inspiration to characterize and query evolutionary clusters.

Antoniol et al. [2005] identify groups of files changing together in order to find non-trivial dependencies between decomposition elements. Such dependencies are expected to denote unwanted couplings in the software system. When characterizing the groups of files changing together, Antoniol et al. consider (1) the size of the groups (i.e. the number of files included), (2) how many times files in the groups changed together and (3) if the files from the same group are coming from different decomposition elements of the software system.

Gall et al. [1998b, 2003] detect logical couplings between software entities, i.e. subsystems, modules and programs, by comparing in which sequences of releases they have been changed. The extent of the logical coupling between two entities is then measured by the number of releases where both entities got changed. The logical coupling between subsystems should be low to achieve a good modularization. The more two subsystems are logically coupled the higher the chance is that we are facing an unwanted coupling.

Zimmerman et al. [2003] define coupling and cohesion measures to assess the modularity of software systems. To achieve their goal, they measure between pairs of lower level program entities, like functions, methods and attributes, (1) how many times they were changed (checked-in) by the same developer, (2) what the ratio is of the number of common changes and the total number of changes, i.e. they measure confidence values. If a pair of software

entities coming from different decomposition elements of the system were changed frequently together with high confidence values then it negatively impacts the measured status of modularity. The Jaccard similarity value which we measure during the characterization is in fact a combination of the confidence values Zimmerman et al. measure.

German [2006] characterizes different types of modification requests (MRs), which are approximated change sets. Our work takes the characterization further to the level of evolutionary clusters which are identified based on the change sets. Furthermore, German uses MRs containing source code files to derive modification coupling between pairs of files. For each file pair they measure (1) in how many MRs both files were present, i.e. co-changed, and (2) confidence values, which is the likelihood that those files will be changed given that one gets changed.

Breu and Zimmermann [2006] mine aspects from version histories. The authors define the concepts of simple and complex aspect candidates showing some similarity to what we call change sets and evolutionary clusters respectively. To identify relevant aspects Breu and Zimmermann use size, support and compactness measures as an input for aspect candidate prioritization.

Mulder and Zaidman [2010] mine frequent itemsets from software repositories to identify cross cutting concerns. Frequent itemsets are sets of items (in our case building blocks) that frequently change together. An itemset thus is a subset of a change set. Determining frequent itemsets does not involve the type of hierarchical clustering we use.

Using visualizations is an alternative way to identify unwanted couplings. When utilizing a visual approach, it is often up to the intuition of the user of the visualization to find and scope the evolutionary clusters. In that case, identifying evolutionary clusters is somewhat subjective.

The visualization of Ratzinger et al. [2005a] helps the user find unwanted couplings by creating a visual overview on the classes which frequently changed together from different modules. D'Ambros [2009] et al. define a layout of files to see which one of them changed together with a selected module of interest. Typically, one looks for unwanted couplings by identifying patterns in the visualization. Which pattern instance is selected to be further analyzed depends on one's intuition. Pinzger et al. [2005] help the formulation of such an intuition by visualizing multiple evolution metrics for several releases of the software system.

The visualization of Beyer and Hassan [2006] provides a layout for software entities such that software entities which change together more frequently in the past are placed closer together. When those software entities are colored according to the decomposition element they are coming from then it is possible to observe the differences between the decomposition of the software and the software entities frequently changing together. Such differences may point to unwanted couplings. Furthermore, to show how unwanted couplings evolved over time, [Beyer and Hassan, 2006] propose a storyboard-like animated visualization.

Fisher and Gall [2006] also identify unwanted couplings using a visualization. But before visualizing files and their relationships they first filter out the "interesting" file pairs, i.e. files from different decomposition elements which changed frequently together. These interesting file pairs form the input for a visualization to detect anti-patterns, like god-classes. Although

the visualization indicates structural anti-patterns, the approach is not addressing explicitly the prioritization of unwanted couplings.

Treude and Storey [2009] describe an interactive tool to visualize how the relevance of concerns in the software system changed over time. Measuring how the relevance of different concerns changed over time is used to identify which concerns did co-occur in a specified time-frame. The tool of Treude and Storey allows its users to define filters on the cluster of concerns identified, somewhat similar to our filtering of evolutionary clusters using the characterization.

By reviewing previous work we can observe that software entities changing together are characterized in a rather simplistic way. The characterization is typically based on the number of common changes (support) and on the observation whether the co-changed entities are coming from the same decomposition element. Furthermore, visualizations of co-changes between software entities are also mainly based on those two properties. As a consequence, the users of such visualizations have to find and prioritize unwanted couplings based on those few properties only. In Section 4.6 we described that it is important and useful to consider a broader and therefore more careful characterization.

4.11 Conclusion and Future Work

In this chapter we have described how to characterize evolutionary clusters and how this characterization may help architects and developers define what constitutes an unwanted coupling in the structure of a software system, and next find those couplings. After a remedy to an unwanted coupling is proposed, we can redo the characterization to a decomposition that is expected to solve the dependency, and compare the outcomes. This 'what-if' type of analysis allows for a quick assessment of changes to the software archive. A similar type of analysis can be done when the organization changes, for example when a group of people moves to another site, or development groups get reorganized. The 'what-if' analysis will then identify possible unwanted couplings because of that organizational change.

We showed that a well prepared characterization is important to support software architects and developers. We described which properties of evolutionary clusters are to be measured for the characterization. We found that, in practice, evolution anti-scenarios are an important source of information on what is considered to be an unwanted coupling. Furthermore, for every property we have argued its relevance for the characterization. Finally, we argued that the characterization itself may evolve over time and that therefore there is a need to retain it.

The properties we identified are very general, and are also found in other systems. Technically speaking, our method is applicable to other systems as well. Whether the properties we distinguish reflect the main cost drivers for the evolution of other systems remains to be investigated.

Characterizing evolutionary clusters and querying them are only the first steps towards the identification of unwanted couplings. A further pruning of the set of evolutionary clusters identified may be necessary, since some of them may still point to couplings which do not turn out to be unwanted after all. In order to decide whether this is the case we need to analyze the identified clusters at a deeper level of detail. This means that there is a need to explore

the clusters to see (1) which are the actual files or methods which changed together, (2) what is the distribution of the properties measured between those software entities (for instance how many times file pairs change together), (3) what is the reason that those software entities changed together. A good visualization supporting the exploration of evolutionary clusters is essential not only to further analyze severity but also to investigate what could cause an unwanted coupling. We address these issues in [Vanya et al., 2010].

The lead architect we contacted frequently during the characterization was positive about the usefulness of the approach described in this chapter. The architect expressed several times that he especially liked the possibility to query evolutionary clusters based on the co-change tendency property. For him, resolving those unwanted couplings which repetitively caused problems in the past are of first priority. Furthermore, thinking about characterization explicitly helped refining the abstract requirement of independent evolution and to express sharper what an unwanted coupling is.

It is a recurring task of software architects to modify the decomposition of the software system. In those cases unwanted couplings do not show up immediately and are not reported back from the developers. Architects acknowledged that finding unwanted couplings faster in the newly defined structure is one of the major values of using characterized evolutionary clusters. For instance, now it is easier for those architects to assess a new development group decomposition.