Supporting Architecture Evolution by Mining Software Repositories

Vanya, A.

2012

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

Resolving Unwanted Couplings Through Interactive Exploration of Co-evolving Software Entities

5

Frequent changes to groups of software entities belonging to different parts of the system may indicate unwanted couplings between those parts. Visualizations of co-changing software entities have been proposed to help developers identify unwanted couplings. Identifying unwanted couplings, however, is only the first step towards an important goal of a software architect: to improve the decomposition of the software system. An in-depth analysis of co-changing entities is needed to understand the underlying reasons for co-changes, and also determine how to resolve the issues. In this chapter we discuss how interactive visualizations can support the process of analyzing the identified unwanted couplings. We applied a tool that interactively visualizes software evolution in ten working sessions with architects and developers of a large embedded software system having a development history of more than a decade. By doing so, we executed Step 5 of the process described in Section 1.7 to help the software architect. The participants of the working sessions were overall very positive about their experiences with the interactive visualizations. In 70% of the cases investigated, a decision could be taken on how to resolve the unwanted couplings. Our experience suggests that interactive visualization not only helps to identify unwanted couplings but it also helps experts to reason about and resolve them.

5.1 Introduction

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

Parts of this chapter has been published as:

- Vanya, A., Premraj, R., and van Vliet, H. Resolving Unwanted Couplings Through Interactive Exploration of Co-evolving Software Entities – An Experience Report. Journal of Information and Software Technology (2011) – to be published
- Vanya, A., Premraj, R., and van Vliet, H. Interactive Exploration of Co-evolving Software Entities. In 14^{th} *European Conference on Software Maintenance and Reengineering* (CSMR '10), pages 269–272, IEEE Computer Society, 2010.

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

In a large system, with many entities, not all of these couplings between entities need to be judged equally critical. For instance, a coupling between two entities that are developed and maintained at two different sites may be judged more critical than a coupling between two entities that are developed and maintained at the same site, since communication between developers at different sites is often more problematic than communication between developers at the same site. In an embedded system, a coupling between entities that reside on different pieces of hardware may be judged more critical than a coupling between entities that reside on the same piece of hardware, since the former makes hardware replacement more difficult. Either type of coupling, or both, may be deemed unwanted. This is an instance of the well-known multidimensional separation of concerns [Tarr et al., 1999].

One of the jobs of software architects is to identify and investigate unwanted couplings. This task is non-trivial for a variety of reasons. In order to resolve unwanted couplings in the software system, architects need to identify their location, understand their cause, evaluate alternatives (changes) to fix them, and assess the cost and implications of the changes. This problem is further compounded because of the sheer number of software entities in large systems, the number of (often distributed) developers, concurrent business goals that require trade-offs to be made, and the like.

Previous work in this area has predominantly focused on helping architects *identify* unwanted couplings in the system. By leveraging data from version control systems, researchers have proposed several visualizations of the software system, emphasizing software entities that have been changed together, i.e., *co-evolved* (or *co-changed*) [Ball et al., 1997, Gall et al., 1998a, Vanya et al., 2008] (see also Section 5.2). However, to confirm that these couplings pose real threats to the system's evolution, architects need to understand the reasons behind the co-evolution of the software entities before identifying solutions to fix them.

In this chapter, our objective is to test the following research hypothesis: *Interactive visualizations of co-changed software entities can be used beyond the identification of unwanted couplings, to reason about and resolve unwanted couplings*. We consider the usage of an interactive visualization successful if it results in identifying the cause of the unwanted coupling and a proposal on how to resolve it (the advice can also be to do nothing about it).

This is not a piece of work about yet another visualization tool. Though we built our own (prototype) tool to suit our purposes, many existing tools offer similar features. It is the *usage* of the visualization that we focus on. In particular, we make the following specific contributions:

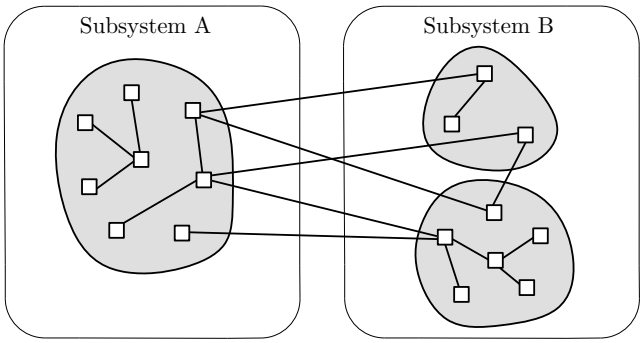


Figure 5.1: A visualization of co-changed entities

- We present a range of possible interactions that can be performed with the data extracted from the version control system of software products (Section 5.3.2).
- We present a case study illustrating the benefits of the use of interactive visualizations in a large scale software system to identify and reason about unwanted couplings (Sections 5.3.1, 5.3.3 and 5.4). We investigated 10 unwanted couplings, 7 of which resulted in a proposal on how to resolve them.

We conclude this chapter with key lessons learned from our experience (Section 5.5), threats to validity (Section 5.6), and conclusions (Section 5.7). In an earlier version [Vanya et al., 2010] we described the essentials of the interactive vizualization tool and discussed one of the three cases discussed here.

5.2 Related Work

When performing the literature study related to this research, we mainly looked at how and why interactivity is used when visualizing co-changing software entities. We restrict ourselves to visualizations of co-changing software entities, as opposed to visualizations of the structure of software systems, such as Rigi [Müller and Klashinsky, 1988], DA4Java [Pinzger et al., 2008], and many others. We also researched how the reasons of co-changes are identified. The following paragraphs summarize the results of our literature study.

5.2.1 Visualizing co-changed entities

Figure 5.1 illustrates the visualization that gives an overview of co-changed software entities. In the figure, two example subsystems (A and B) are depicted using rounded rectangles. Each subsystem comprises one or more components which are illustrated using gray colored blobs. Inside each component, files are depicted using white colored squares. Files that co-changed are shown by drawing an edge connecting the two.

Variants of Figure 5.1 have been used by others to visualize software evolution. Fischer and Gall [2006] used the symbol size (in their case, a square too) to indicate the change frequency of a file. Antoniol et al. [2005] used a graph layout algorithm that places two files closer to each other if they were co-changed more frequently. No lines between files were drawn. A similar approach is taken in [Ball et al., 1997, Beyer and Hassan, 2006, Beyer and Noack, 2005]. Hindle et al. [2007] use different widths for the lines connecting files to reflect the strength of their co-evolution.

Although such visualizations provide ample information about co-changed entities, only limited attention has been paid so far on how interaction and dynamic updates can support decision making. Examples of currently supported interactions include zooming in/out and filtering files to display only a subset of the software entities. However, such interaction may not be of much help to understand the underlying rationale for the co-evolution. We are especially interested in recovering such rationales related to design decisions to help architects resolve issues.

5.2.2 Interactions with visualizations

Some previous work uses a purely static visualization of co-changing software entities. Those visualizations show software entities and the co-evolution type relationships between them, but they do not allow the user to initiate any interaction, see for example the work of Gall et al. [2003].

Other works allow some interaction with the visualization. The type of interactions supported by these works varies somewhat. A frequently supported interaction type is that the user can select from which time period the version management data should be used to create the visualization. This allows one to study how the evolution-type relationships between software entities evolved over time. Several works support this type of interaction [D'Ambros et al., 2009, Hanakawa, 2007, Beyer and Hassan, 2006, Ratzinger et al., 2005b]. Hindle et al. in [2007] take the idea even further by creating a framework for film-like demonstrations.

Panning and zooming are interaction-related features that help mitigate scalability issues. They are implemented, for instance in the EvoLens tool [Ratzinger et al., 2005b] and in the visualization tool of Beyer and Noack [2005].

Some previous work allows the user to select software entities and/or the couplings between them to get additional information about the entities and couplings selected. In case a file is selected, the Evolution Radar [D'Ambros et al., 2009] can show the content of the file or indicate the related version management meta-data, like check-in comment, timestamp and developer id. Selection of software entities can also be used to change the focus to the entity selected [D'Ambros et al., 2009, Ratzinger et al., 2005b], track the evolution of the entity selected [D'Ambros et al., 2009], fold and unfold modules [Ratzinger et al., 2005b].

Another type of interaction described in the literature is that users of the visualization can specify which entities to hide or show. In [D'Ambros et al., 2009] it is possible to specify, for instance, whether all files or only source code files should be included in the visualization.

Subsystems, files, methods are software entities at different levels of abstraction. The lower

the abstraction level, the more and finer grained software entities we are talking about. One of the interactions Ratzinger et al. [2005b] implemented in their tool is to allow the user to set the abstraction level of software entities. This allows the user to get an overview (high abstraction level) or get more details (low abstraction level) of the co-changed software entities.

The number of times software entities changed together (called 'support') is used to indicate which co-changes are more problematic. Therefore, some of the previous work [Ratzinger et al., 2005b] implements interactive visualizations where the user can set the threshold for the number of co-changes. Then only those relationships between software entities are visualized where the support value exceeds the predefined threshold.

An important aspect of visualizations is how the layout of the entities to be visualized is set. Depending on the layout, the user of the visualization may look at the entities visualized in multiple ways. [D'Ambros et al., 2009, Ratzinger et al., 2005b] define different layouts and let the user choose which layout best suits their investigations.

Except for the option to drag entities around (Section 5.3.2), and to hide/show relations between entities from the same part of the software system (Section 5.3.2), the interaction features our tool offers do not differ much from those offered by existing tools. It is the *usage* of our interactive visualization tool that differs. Previously, interactive tools visualizing co-changes of software entities were used mainly to *identify* unwanted couplings. We evaluated if and how interactive visualizations can also help to *reason about and resolve* those unwanted couplings identified.

Table 5.1 (see next page) indicates which interactions different visualizations are reported to implement. Note that visualizations which support the same type of interactions may implement that interaction type in different ways.

In Section 5.3.2 we further elaborate on four of the above interaction types which we identified as being relevant to identify reasons for co-changes and find solutions to unwanted couplings: (1) Set abstraction level, (2) Set support threshold, (3) Hide internal relationships, and (4) Drag entities.

5.2.3 Reasons for interaction

We reviewed articles on the visualization of software evolution to understand why interaction-related functionalities were implemented. The reasons found are summarized in the following list:

- to identify design erosion [Ratzinger et al., 2005b]
- to identify structural decay and the spread of cross-cutting concerns [Beyer and Hassan, 2006]
- to identify change smells [Ratzinger et al., 2005a]
- to explore the effect of changes that occur within a software system over time [Hindle et al., 2007]

Table 5.1: Interactions supported

	Set input time period (IT)	Pan and Zoom (PZ)	Select entities (SE)	Show / Hide entities (HE)	Change layout (CL)	Set abstraction level (AL)	Set support threshold (ST)	Hide internal relations (IR)	Drag entities (DE)
Gall et al. [2003]									
Ratzinger et al. [2005a]									
D'Ambros et al. [2009]	✓		✓	✓	✓				
Hanakawa [2007]	✓								
Beyer and Hassan [2006]	✓								
Ratzinger et al. [2005b]	✓	✓	✓		✓	✓	✓		
Hindle et al. [2007]	✓								
Beyer and Noack [2005]	✓	✓							
iVIS (our tool)		✓	✓		✓	✓	✓	✓	✓

- to assess the complexity of the software [Hanakawa, 2007]
- to uncover hidden dependencies [D'Ambros et al., 2009]

This list shows that interactions are mainly used to identify *what* evolution type issues there are in the system. Whether interactions can also help investigate *why* those issues are there and *how* to resolve them is the question we address in this chapter.

5.2.4 Identifying reasons for co-changes

When we want to understand why a structural issue exists then we may look at the reasons for co-changes between each pair of files involved in the issue. Mockus and Votta identified reasons for co-changes by looking at the textual description fields of modification requests (MRs) [Mockus and Votta, 2000]. Check-in comments from version management systems are also used for the same purpose [Kim et al., 2006, 2008]. Yet another alternative is to consider the reason field in postlist files [van de Laar, 2009], which developers create to describe which source files they modified related to one or more development tasks. Finally, one may inspect the source code, as is for instance done in [D'Ambros et al., 2009].

5.3 Experimental Design

We tested our research hypothesis by observing how the software architect and developers in our study environment use the interactivity of the visualization tool we provided them to reason about and resolve unwanted couplings. To reach that point we had to

- identify the unwanted couplings which the architect and developers wanted to analyze
- define and implement the interactions to be tested
- organize working sessions with the architect and developers to analyze the unwanted couplings

These steps are described in the following subsections.

5.3.1 Unwanted couplings to be analyzed

Co-changed software entities are entities that were changed and committed together in the version control system. Identifying entities that were changed together is central to our work because it is the data that is eventually visualized and presented to the architects in order to support them in decision making. Co-changed entities may point to unwanted couplings in the system, for instance if these entities belong to different parts of the software system. In this section, we present the procedure we used to mine co-changed entities and to identify unwanted couplings.

Raw data extraction

ClearCase provides the functionality to extract file check-in related meta-data via a command line interface (similar to the `log` command in CVS and SVN). This meta-data typically includes information such as the id of the developer who modified a file, the name of the file which was modified in the software system, and when the modification took place. Furthermore, like many other version management systems, ClearCase can store the reason why a file was changed given that a developer specifies such a reason text.

We collected this meta-data from all available projects and branches of the software system for the last nine years of development history. This meta-data also allowed the determination of the containing building blocks and subsystems for each file in the system by using its absolute pathname. Building blocks are directories in the code structure representing the next level of abstraction above files, see also [van der Linden and Müller, 1995].

Identifying co-changed software entities

ClearCase is similar to CVS—check-in related meta-data is stored at the file level. This makes it impossible to determine which files were changed together (to resolve a specific issue) and checked-in to the version control system at the same time. We identified co-changed files

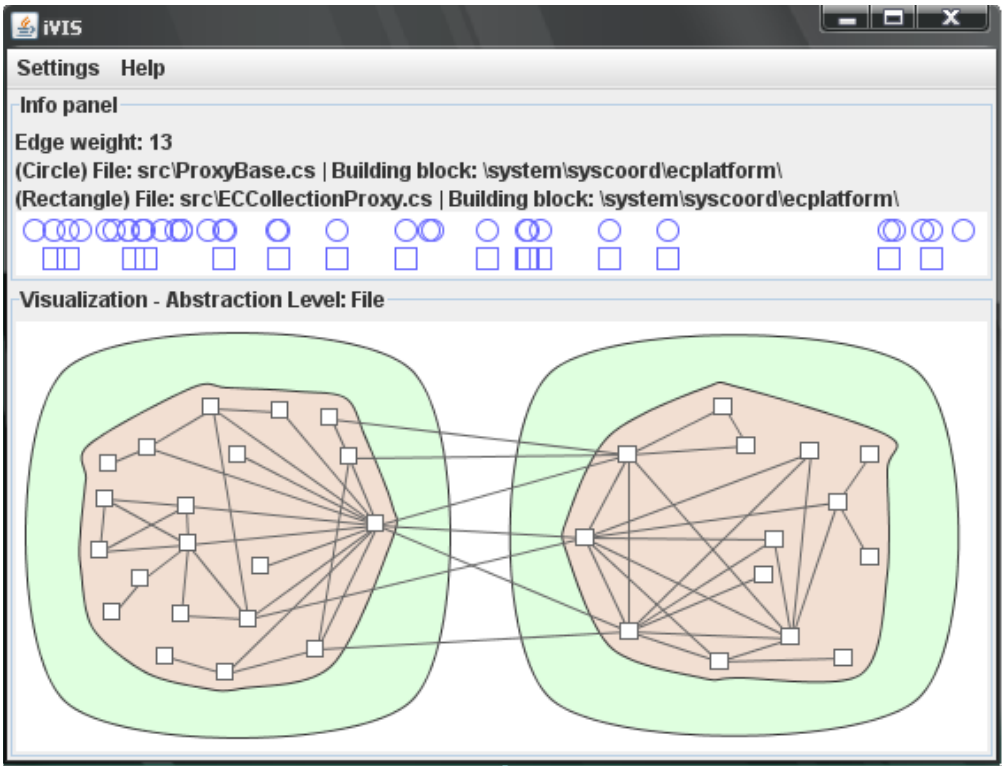


Figure 5.2: iVIS main screen

from the meta-data using the sliding time window algorithm borrowed from Zimmermann and Weißgerber [2004] such as leveraging author names and commit log messages to infer groups of files that were changed together. These heuristics were adapted to fit our study environment and then used to approximate which files were changed and checked-in together in the system.

Identifying Unwanted Couplings

To identify unwanted couplings we first use the data about co-changed entities and compute how often pairs of software entities changed together, given at least one of them was changed, using the Jaccard similarity coefficient [Abreu et al., 2006]. Next, the similarity coefficients for all pairs of co-changed entities are fed into an Agglomerative Hierarchical Clustering Algorithm [Tryan, 1939], which results in a dendrogram—leaf nodes of the dendrogram indicate groups of co-changed entities [Vanya et al., 2008].

we only use the developer id, not any commit messages, since the latter are usually not stored in the environment we studied

Not all groups of co-changed entities point to unwanted couplings. Therefore, we characterize and filter them on the basis of, for instance, whether their constituent entities belong to different enclosing entities, how many times the entities co-changed, and how recently the entities co-changed. During this process, we intensively interact with an architect, to elicit his interests and express it in a scenario, such as "I am interested in unwanted couplings involving multiple development sites and where the co-changes happened recently.". The resulting scenario is next used to create a query to filter and sort the groups of co-changed entities. For more details, we refer the reader to our previous work on this topic [Vanya et al., 2009].

For one such scenario, the top 10 groups of co-changed entities that resulted from executing the query were selected and analyzed further using the interactive visualization, as described in the following section.

5.3.2 Interacting with co-changed entities

The ability to interact with a visualization in real-time for the purpose of data exploration opens up a wealth of possibilities. Take for instance Google Maps—a free and comprehensive geographical mapping engine covering most countries on the globe. Some of the key benefits of using Google Maps are direct results of its API that allows users to interact with the map: zooming in/out, changing from map to satellite view, toggling terrain view on/off, viewing user created overlays for more information, etc.

We believe that transforming visualizations such as those in Figure 5.1 into interactive visualizations opens several new possibilities to support architects too. We implemented a tool (iVIS, see screenshot in Figure 5.2) to facilitate such interaction with our data on co-changed entities.

Similar to Figure 5.1, the screenshot displays two subsystems, their constituent building blocks and files, and edges that connect files that are known to have changed together in the past. By clicking on an edge, the tool presents the names of the two files, their parent building blocks, and the frequency of co-changes (*edge weight*) in the *Info Panel*. In addition, the tool then presents a plot showing when the two files were changed. The circles below the file names in Figure 5.2 indicate individual changes of the first file. The squares denote the changes to the second file. The scale of the plot is normalized by the time difference between the time of creation of the older of the two files and the time of the most recent change made to either file. The squares or circles on the timeline overlap when two check-ins of the same file happened close in time. Figures 5.4-5.6 and 5.8 essentially are screenshots as well; however, we only depict the graphical part, not the info panel, and added some labels to enhance readibility.

We implemented the tool in Java. We reuse the interactive graph drawing tool from the prefuse visualization toolkit [pre, 2009].

The tool is available upon request from adam.vanya@gmail.com.

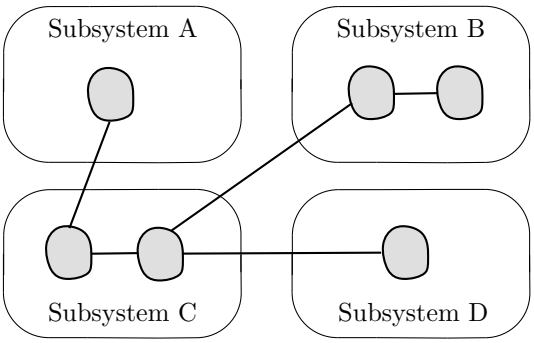


Figure 5.3: Abstraction level increased

Set abstraction level

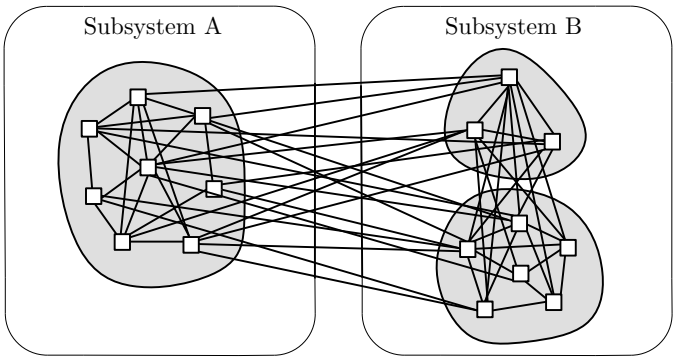
Figure 5.1 visualizes software entities at three different levels of granularity: subsystems, building blocks, and files. Such visualizations involving multiple files from different building blocks and subsystems can quickly become cluttered and lose their purpose. This undesirable effect can be avoided by allowing the data to be plotted at different levels of granularity. For instance, Figure 5.3 shows the co-changes in four different subsystems and the constituent building blocks, while hiding the files that were co-changed. With such a graph, architects might more conveniently identify candidate unwanted couplings in the system and further investigate them in depth. In iVIS, a user can change the abstraction level through a simple menu.

Set support threshold

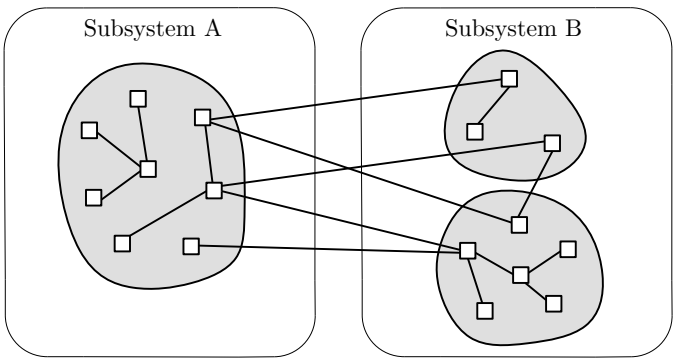
The frequency with which software entities co-change is one measure of the strength of interdependency between those entities. Not all such changes may be of interest to the architects. For example, architects may be aware that one-shot changes in files from different subsystems may be made in a highly controlled manner and pose no risk to the system; these architects may hence be more interested in frequently occurring co-changes.

Adjusting the threshold level for the number of co-changes between files on the fly can help architects focus on more serious issues. The optimal value for the threshold may vary from one system to another and depend upon the investigation. In some cases, architects may simply be interested in learning the overall pattern of co-evolution between the entities; in other cases, architects may be interested specifically in strongly co-changed entities across different subsystems [Ratzinger et al., 2005a]. Furthermore, architects may wish to validate their speculations about relationships between files by varying the threshold value. For example, an architect may expect files A and B to co-evolve with file C. However, this pattern may only reveal itself by lowering the threshold value.

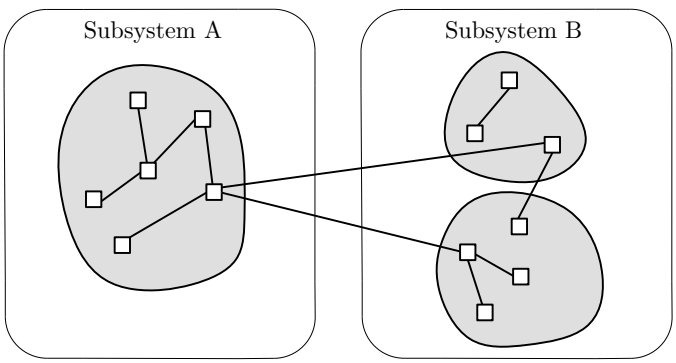
As an added advantage, raising the threshold reduces clutter in the visuals, thus emphasiz-



(a) Low support threshold

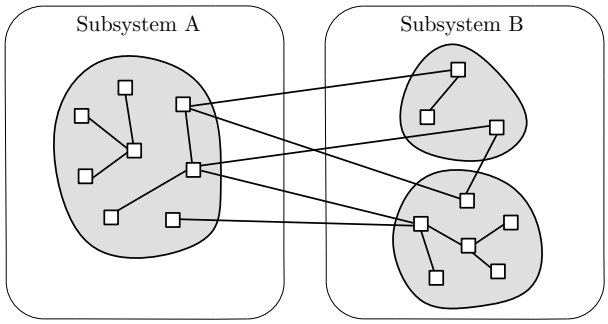


(b) Medium support threshold

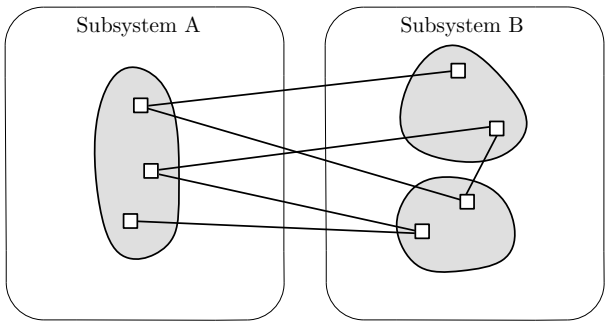


(c) High support threshold

Figure 5.4: Support threshold



(a) Internal Relations Shown



(b) Internal Relations Hidden

Figure 5.5: Internal relations

ing the more serious issues in the system. See for example Figure 5.4 that shows the evolution of software entities at three different threshold levels of co-evolution. In Figure 5.4 (a), a low threshold value for co-evolution has been set which shows most co-changes between files. Increasing the thresholds as in Figures 5.4 (b) and 5.4 (c) reduces the number of files and edges in the graph, and shows only the more frequently co-changed files. Similar to adjusting the abstraction level, the threshold level can be changed in iVIS through a menu.

Hide internal changes

Changes where the files involved are located in the same software entity, be it a building block or subsystem, are internal changes. External changes, on the other hand, are changes made to entities that belong to different enclosing entities.

While external changes are relevant to architects in nearly all cases, both hiding and showing internal changes has advantages when analyzing the cause of the evolution of the system. Hiding internal changes makes visualization less cluttered and is therefore easier to study. Also, changes across the borders at higher levels of granularity are more indicative of un-

wanted couplings and, thus, of more interest to the architects.

On the other hand, showing internal changes may help recall the role of a software entity or identify a co-change pattern. Knowing the internal relations also allows one to conduct an impact analysis ahead of making any changes. For instance, architects may wish to evaluate the consequences of a file being moved to another subsystem. By examining the internal relations, they could immediately see which relations would become external. Architects then see whether moving the file in question would mitigate or intensify the unwanted coupling being analyzed.

We allow turning the display of internal changes on or off. The effects of this interaction are illustrated in Figures 5.5 (a) and (b). In Figure (a) the former figure, one can see all co-changed files, whether internal or across enclosing entities. Figure (b) allows the architect to concentrate on changes across boundaries since the internal co-changes are hidden.

Drag entities

This interaction allows the users to select a software entity (file, building block, or subsystem) and drag it to a new position in the visualization. The resulting graph is isomorphic with the original. Figure 5.6 is an example of how one may re-arrange the layout of the visualization by dragging files to a new position.

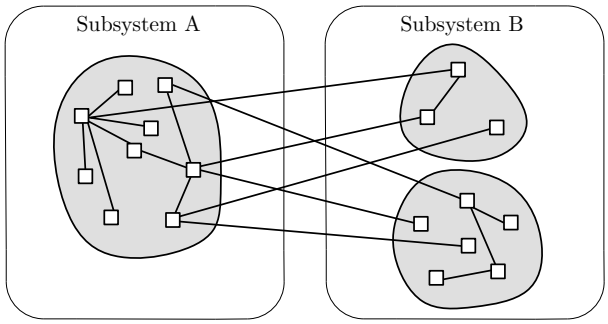
There are many ways in which such an interaction can help architects. First, the ability to drag an entity allows visualizing the entity itself and the edges related to that entity in a more pronounced way. This may help understand the relation of the entity to the rest of the system, and can also be useful when problematic entities or relationships related to unwanted couplings have to be recalled.

Next, developers can drag software entities to group them according to the system's design—for instance, some software entities contribute to more than one functionality and, as a result, are co-changed with two or more distinct other sets of entities. Repositioning such entities in the visualization could help making such a relation appear clearly.

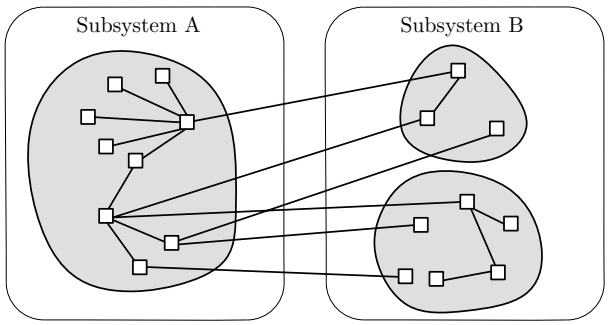
Lastly, the user can drag software entities to express a mental picture about the part of the software system being analyzed. For instance, building blocks can be placed above each other to express that the intention is to have a layered architecture.

5.3.3 Analysis of unwanted couplings

We validated the usefulness of using interactive visualizations by inviting architects and selected developers of the software system to working sessions in which the candidate unwanted couplings identified (see Section 5.3.1) were discussed. During the sessions, we observed how interactive visualizations lend themselves to help recall the reasons behind the underlying changes that contribute to the identified couplings, and confirm that the identified couplings are indeed serious and require action to counter their effects on the system. Ten such working sessions were organized; in each session, we explored one coupling together with core developers of the entities involved.



(a) Before Dragging



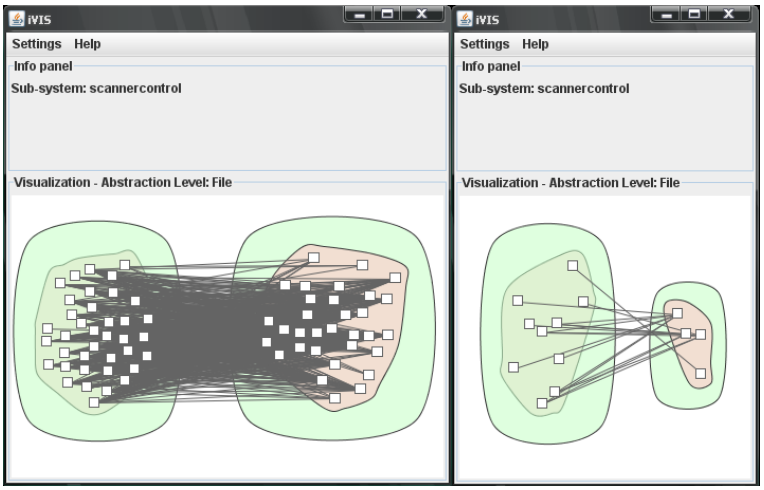
(b) After Dragging

Figure 5.6: Re-arranging entities

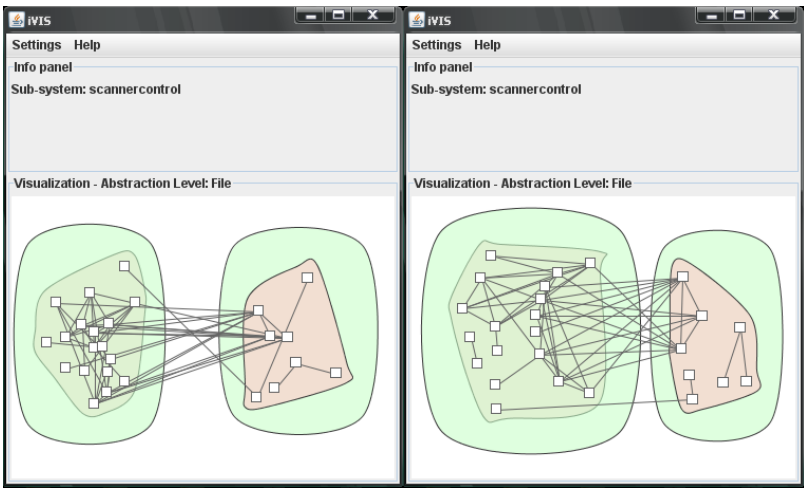
Prior to these sessions, we set iVIS up to present an initial visualization of the identified unwanted coupling candidates so as to begin the group discussion. This preparation phase involved determining the level of abstraction to visualize and a good threshold value for co-evolved entities, deciding whether internal links should be visualized or not, and lastly the layout of the nodes on the graph such that the unwanted couplings are made to surface. These preparations on the visualization are illustrated in Figure 5.7 for a working session which we later describe in detail in Section 5.4.3.

Such preparations were conducted in consultation with the architect for all but 3 meetings when the architect was unavailable. In the latter cases, the author of this thesis used his own experience and judgement to arrive at an initial layout. The architect could often help creating the initial layout, and therefore limit the time needed from the developers, since he had a hunch about the entities involved.

Creating an initial layout for the visualization was important to minimize the amount of time needed from the developers we invited to the sessions. Also, very often the architect had an idea what the identified unwanted coupling could be. In those cases we created the initial layout such that the software entities playing a major role in the unwanted coupling, according



(a) Step 1: Initial visualization. (b) Step 2: Threshold adjusted.



(c) Step 3: Internal couplings shown. (d) Step 4: Layout re-arranged.

Figure 5.7: Preparation of the visualization for the working session.

to the software architect, were emphasized. This way we could quickly validate with the developers if the assumptions of the architect were correct or not. In case the assumptions of the architect did not hold the layout was further modified during the session.

Each session was commenced with a briefing to the participants on the purpose of the session. They were also given a demonstration of iVIS's functionalities and how they could

be used to facilitate discussing and reasoning in the session. During the session, participants interacted with iVIS to reason and explain to themselves why certain files co-changed.

To support or even confirm their reasoning, developers sometimes used additional sources of information. For one co-changed file pair developers applied the `diff` tool of ClearCase between subsequent revisions of those files to try to identify patterns in the co-changes. In other cases, developers needed to know which developer carried out a change to both of the co-changing files and which software project they ran at that time.

The main role of the architect during the meeting was to listen to the developers' reasoning about why files co-evolved. The architect then abstracted from the domain specific terminology of the developers to understand the design decision(s) that resulted in the co-changes. The architect also discussed possible design alternatives with the developers to come to an agreement on how the system's structure has to be improved.

5.4 Case Study

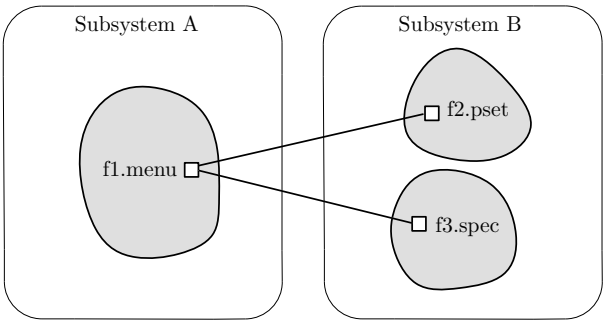
In the following sub-sections, we present specific episodes from the working sessions that demonstrate how the interactive visualization facilitated the reasoning process. These episodes support our viewpoint that interactive visualizations can be very helpful in understanding root causes behind unwanted couplings in large software systems. The three cases we describe were selected such that we can demonstrate the different benefits of an interactive visualization; in Case 1 to identify causes of unwanted couplings, in Case 2 to identify solution alternatives and in Case 3 to evaluate an alternative solution.

5.4.1 Case 1: Recalling reasons for co-changes

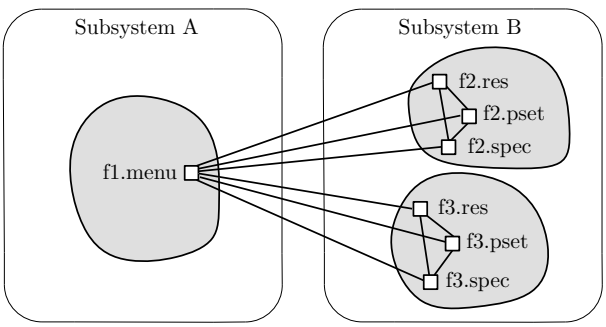
One of the unwanted couplings identified in our analysis in Section 5.3.1 showed how interaction helped the participants recall the reasons behind the visualized co-changes.

The preparation phase for the discussion resulted in a visualization for this issue similar to Figure 5.8 (a). The file names in Figure 5.8 (a) and Figure 5.8 (b) are only used here as annotations. When the tool is used the names of the files are shown when clicking on one of the squares. In Figure 5.8 (a), we can see that file `f1.menu`, contained in subsystem A, has been often co-changed (a high threshold was set) with files `f2.pset` and `f3.spec`. The latter two files belong to different building blocks in subsystem B. The participants were unable to give the reason for this change pattern and chose to interact with the visualization hoping an explanation would surface.

The visualization was updated after lowering the threshold, but this resulted in so many files and edges being plotted on the screen that a careful examination became very difficult. Next, the threshold limit was adjusted several times until a visualization similar to Figure 5.8 (b) emerged that showed that in fact, `f1.menu` was changed together with files `f2.pset`, `f2.res`, `f2.spec`, `f3.pset`, `f3.res`, and `f3.spec`. This triggered the participants to recall the general practice of adding tests for new hardware.



(a) Initial visualization at high threshold.



(b) Updated visualization with adjusted threshold.

Figure 5.8: An example showing how adjusting threshold value revealed relevant co-changes.

The software system under study includes several hardware parts, such as magnetic resonance imaging machines and re-constructors (they manipulate raw images for examination). In order to add or improve functionalities, hardware components and/or their software are updated and tested. Hardware related tests are performed using an application supported by a graphical user interface (GUI). The GUI allows the testers to select and initiate the execution of hardware tests using a menu structure.

Only hardware tests that appear in the menu structure can be executed. Defining a hardware test involves (1) changing the file of test definitions (the `.res` file), (2) modifying the parameters (the `.pset` file), and lastly (3) updating the outcome specification and acceptance criteria (the `.spec` file). Every major hardware component is associated with such test files.

The visualization (similar to Figure 5.8 (b)) helped participants recall and explain that each time a new hardware test is defined, the `.pset`, `.res`, and `.spec` files are modified. With this understanding, it also became immediately clear to the participants that file `f1.menu` was updated to include an item in the menu structure for the newly added test.

The participants in the session considered various solutions to this unwanted coupling. One solution was to automatically generate the menu structure from the tests available, possibly

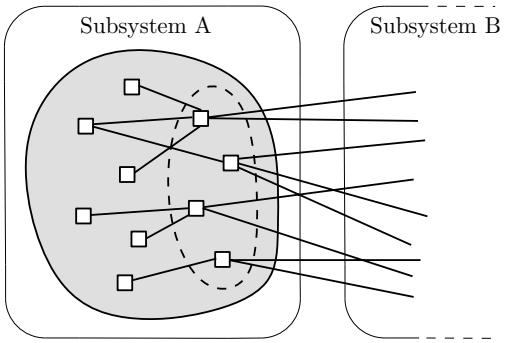


Figure 5.9: An illustration of the unwanted coupling in Case 2.

using a design pattern. Another solution discussed was to revise the software process such that developers would add a new test to the menu file only towards the end of integrating the new hardware into the system. For reasons of practicality, the participants agreed the latter solution to be the most suitable.

5.4.2 Case 2: Seeking solutions to unwanted couplings

In another working session, we considered a candidate unwanted coupling that involved seven building blocks from one subsystem being co-changed with one building block from another subsystem. The files involved dealt with creating templates for examination reports from medical image analyses.

During a medical examination, a series of images is acquired; how each of these images is to be captured is planned in advance. Improvements and new functionalities are continuously considered to reduce examination time. Given that many examinations are similar in nature, one option to speed them up was to create examination templates that could be automated and reused.

The issue analyzed in this case involved seven building blocks related to the examination templates. These building blocks implemented different functionalities, such as the user interface, the database, and data model of the templates. When we observed that these building blocks were changed with another building block that belonged to a different subsystem, developers were quick to explain that creating or modifying image acquiring methods opened new opportunities for report generation. This is why the examination report templates – residing in another subsystem – were changed in parallel.

In this particular case, the main benefit of the interaction is in finding how the interdependency between the two subsystems involved can be reduced. To evaluate possible solutions, participants from the session first interacted with the tool to show internal relations between files. Then they manually changed the layout of the building block implementing the image acquisition methods by dragging files, which resulted in a layout similar to Figure 5.9.

In Figure 5.9, subsystem A contains the image acquisition methods, whereas subsystem B contains the seven examination report template related building blocks. From the visualization similar to the one in Figure 5.9, participants realized that files with external relations in subsystem A were all `.cs` files (files encircled with a dotted line in Figure 5.9), and these were all internally related to `.c` files. Also, the external relations in subsystem A outnumbered the internal relations. For this reason, and because the existing internal relations between `.cs` and `.c` files are easier to maintain, participants agreed to move the `.cs` files into one of the building blocks in subsystem B.

5.4.3 Case 3: Co-change of C implementation and header files

This unwanted coupling involves two subsystems. One of them implements functionalities needed to visualize and manipulate the medical images acquired during the scan of a patient. The other subsystem controls the scan. For instance, using the latter subsystem, the medical worker can specify which intersection of the human body has to be scanned and which type of tissue has to be highlighted on the resulting image.

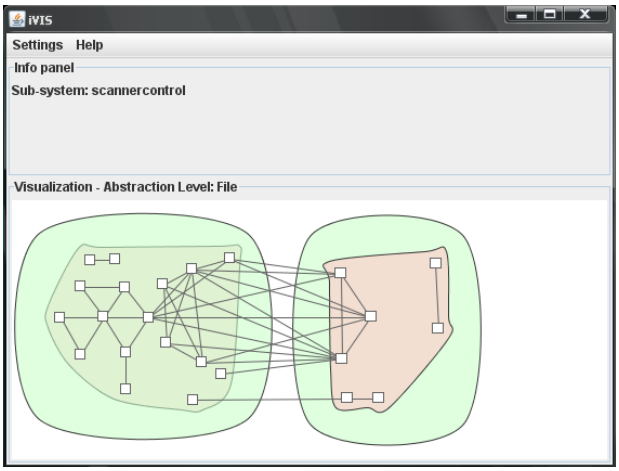
From each of the two subsystems one building block was identified as being related to the unwanted coupling. During the working session with the architect and developers we noticed that there were many co-changes between files from those two building blocks.

The interaction with the tool helped us identify those subsystem crossing relations which were notably stronger than others. The architect and developers assumed that these relations were the ones mainly contributing to the structural issue under discussion and decided to analyze them further.

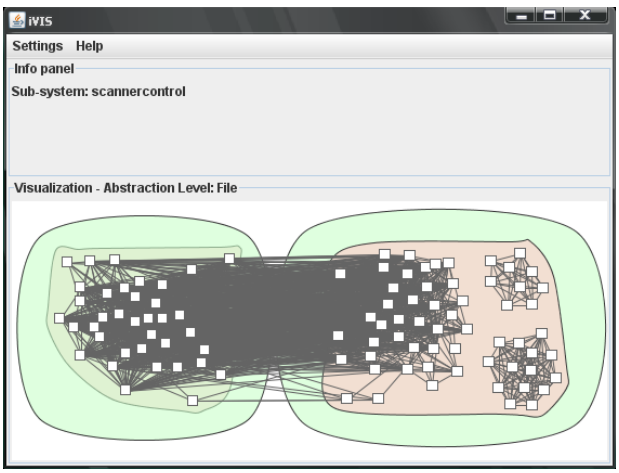
To find out which subsystem crossing relations are strong, i.e. which files changed frequently together, we first set the support threshold to 25 co-changes, so that only the strongest subsystem crossing relations remained. We then gradually decreased the support value. While doing so we noticed that the set of subsystem crossing relations first quickly increases, and then remained stable (between 19 and 10 co-changes). Further decreasing the threshold relatively quickly resulted in visualizing all the subsystem crossing co-changes. Figure 5.10a shows a screenshot where only the strong relations (having more than 19 co-changes) are shown.

The analysis of the frequently co-changing files from different subsystems revealed that they were C header files from one of the subsystems and their corresponding C implementations from the other subsystem. The developers recalled that these files earlier resided in the same subsystem. Recent restructuring activities, however, incorrectly assigned them to different subsystems.

The developers agreed that the solution was to move both the C header and the implementation files to the same subsystem. The C header and implementation files under discussion changed relatively infrequently with other files from the same subsystem. Therefore, the developers suggested the architect to move the C files involved to a third building block. The suggestion of the developers was accepted by the architect and a problem report (PR) was issued to carry out the necessary changes.



(a) Support threshold set



(b) Support threshold unset

Figure 5.10: Co-changes in Case 3

5.5 Discussion

The participants from the working sessions were overall very positive about their experience with the interactive visualizions of co-changing enities. The following is a quote from the chief architect of the software system at Philips Healthcare describing the usefulness of this type of visualization:

Table 5.2: Overview of the working sessions

Duration (minutes)	Attendees	Interactions used							Decision taken
		PZ	SE	CL	AL	ST	IR	DE	
60	**4**		✓	✓		✓		✓	**Decision on the software process**
90	3		✓	✓		✓	✓	✓	Unwanted coupling yet unresolved
60	**2**		✓	✓	✓	✓	✓	✓	**Create a PR: relocate files**
30	2		✓			✓			Create a PR: relocate files
30	3		✓	✓		✓		✓	Unwanted coupling yet unresolved
60	2		✓	✓		✓	✓	✓	Create a PR: relocate files
60	**2**		✓			✓	✓	✓	**Create a PR: relocate files**
60	2		✓			✓			Create a PR: phase out a BB*
90	3		✓	✓	✓	✓		✓	Unwanted coupling yet unresolved
60	2		✓			✓	✓		Do not change anything

*BB stands for building block

> *"The tool helps to quickly zoom in from the level of BB [building block] co-changes to files. It reveals patterns of file-level co-evolution relations that characterize the hot spot [unwanted coupling]. In many cases, this view added with top-level domain knowledge yields a hypothesis of the underlying structural issue. In a majority of cases, our hypotheses turned out to be correct."*

The sessions resulted in a better understanding of why software entities co-changed. As a result, the architect initiated steps to resolve confirmed unwanted couplings, while considering how other system requirements besides software evolvability would be affected by these steps. In many cases, problem reports (PRs) were created as a direct result of the working sessions.

Table 5.2 gives a summary of the 10 working sessions. For every working session Table 5.2 specifies how long it took, how many architects and developers attended the session and what the outcome of the session was. The architect was present at seven sessions (including case 1 discussed above). 14 different developers participated in the sessions; one developer participated in three sessions, three developers participated in two sessions, while nine developers participated in one session only. No developer participated in more than one of the cases described above.

The order of the sessions in Table 5.2 reflects the order in which those sessions were held. The three rows in boldface refer to the three cases we presented in Section 5.4. Interactions are labeled with the abbreviations we already introduced in Section 5.2:

- PZ: Pan and Zoom
- SE: Select Entities
- CL: Change Layout

Table 5.3: Sequences of interactions during the sessions discussed

Case	Chronological sequence of interactions used
1	ST, SE, DE, CL, SE, DE, SE
2	ST, AL, ST, AL, CL, SE, CL, IR, DE, SE, DE, SE
3	SE, ST, SE, DE, IR, SE, DE, SE

- AL: Set Abstraction level
- ST: Set Support threshold
- IR: Hide Internal relations
- DE: Drag Entities

As one can see, selecting entities (SE) and setting the support threshold (ST) were always used. Although we implemented panning and zooming, it was not used at all during the sessions (except during the set up phase of one working session). The number of interactions used varies; in some sessions almost all the interactions implemented were used, and others only used a few.

The specific question to be answered guides the interactions used, and no standard procedure can be defined. However, during the working sessions we did observe some usage patterns. Developers typically start analyzing the co-changed entities by adjusting the support threshold to get a feeling of which software entities mainly contribute to the unwanted coupling. After the support threshold is set, entities are typically further analyzed so that the developers can recall what those entities exactly are. This step is often followed by re-arranging the layout such that it is in line with the mental picture the developers have of the structure of the system. Table 5.3 shows which interactions followed each other over time during the three working sessions described in Section 5.4. These sequences support our observations about the usage patterns described.

The participants also expressed some feature requests for a next version of the tool. The top three requests were:

- Show more information about the co-changes.
- Possibilities to annotate the visualization, and
- The ability to manually remove spurious co-changes.

The participants reasoned that knowing the name(s) of the developer(s) who committed the changes would allow them to know the expertise of the person, and the project he was involved in at that point of time, and it would be helpful if this information could be shown as well. This information would have helped them to more efficiently identify the reason for co-changes and in general the reason for the unwanted coupling analyzed.

Being able to anotate the visualization was another feature request of the participants. They suggested that the tool itself should provide a means to document the findings of the analysis. For example selecting an already analyzed coupling should reveal the result of the analysis. Developers reselected couplings from time to time and wanted to know what the result of the earlier analysis was.

Some co-changes identified were the result of actions, such as merges, that do not really point out couplings between files. Participants wanted to have the ability to manually disable co-changes between those files. By doing so, it is much easier to focus on the real issues.

The current version of the tool only shows the entities that change together. The context of those entities, i.e. all other entities that are part of the same subsystems but which do not change together, are not shown. A straightforward extension of the tool is to extend the info panel with the button to show/not show this context, for example to show the size of a subsystem, or the number of files in it. One reason why this type of information was not needed is that the participants had a lot of expertise in the system. They knew the size of subsystems, and need not be told.

In a similar vein, integrating structural information through the info panel could be a useful extension of the tool. Again, this requirement did not show up during our working sessions because the participants were experts in the system. Such shows, for example, in Case 2 discussed above. Here, the participants hypothesized a set of `.cs` files to be internally related to `.c` files. The visualization next showed this to be true.

In all working sessions, the participating architect and developers could identify why files related to the unwanted coupling presented changed together. In seven cases, decisions could also be made on how to handle the issue. As for the remaining three cases, which include the two longest lasting sessions, the developers identified alternatives to solve the unwanted couplings but together with the architect they could not decide on a solution during the sessions because of the complexity of the issue.

During the sessions, we kept a log of how developers used the interactive visualizations. The following lists the main observations we made:

- Setting the threshold value in the preparation stage such that the visualization shows the first few subsystem crossing co-evolutions worked well to kickstart a discussion amongst the participants. For example, in Case 2 a few cross-subsystem co-changes (shown in the visualization after the preparation phase) sufficed to determine the roles of the files participating in the co-changes and articulate a likely rationale for those co-changes. The further interactions served to verify that initial observation.

  During one of our working sessions we initially set the support threshold very low. This resulted in a visualization where almost all the files were connected. Due to the huge amount of the connections the visualization was not useful any more. When we set the threshold somewhat higher, the visualization cleared up and one of the developers noted that it was a very useful feature of the iVis tool to show only those relations which are stronger than a give threshold, since this way one can easily filter out noise from the presentation and get focus.

- Adjusting the threshold level often helps developers to validate their conjectures about the co-changes of entities. In Case 2 a developer participating in the working session noticed that the co-changes between building blocks were in fact co-changes of .cs files in those building blocks. After pointing to one of the building blocks that developer told the following:

  > *"I expect that internal to this building block all the .cs files were only changed frequently together with .c files. If we could verify this with the tool then an alternative to make the problem analyzed less severe would be to move all the .cs to the other building blocks. It is because I expect that .c and .cs files are changed less frequently than .cs files."*

  Lowering the support threshold from 27 to 4 revealed that the developer was right: only .c files were changed frequently together with .cs files internal to the building block in discussion.

- Adjustments to the threshold value helped participants recall the design decisions behind specific co-changes. Lowering the threshold in Case 1 helped participants recall that adding new hardware tests resulted in modifying the menu files in one subsystem and the corresponding .res, .pset and .spec files in another. The design decision being to clearly separate *what* is tested from *how* it is tested. When the threshold level was initially very high the developers only saw menu files related to one of the .res, .pset, .spec files. One of the developers had a remark on this:

  > *"I do not yet see why the menu file would only change together with the parameter file[.pset file]."*

  After lowering the threshold the developers could see that without any exception the menu file was always changed together with all the three type of files describing how a functionality is tested.

- Dragging entities allows participants to link the visualization to their mental representation of the system. For instance, one of the participants in Case 2 grouped .cs files to see which other files that group was related to. When ready with the grouping of .cs files the developer explained that they represent the examination templates, see Case 2. Just after the grouping happened, the developer started to think about moving the .cs template files to other subsystems. So grouping in this case helped the developer to formulate his ideas at a higher level of abstraction.

  In another case grouping helped to reveal that the originally intended layered architecture was not correctly implemented. In that case the drivers of the software were designed to serve sub-systems including one which implements examination methods. Examination methods are intended to be used by a control type subsystem. During one working session the co-changes between the drives and the control subsystem were analyzed. After grouping the co-changed driers and the co-changed control files it became clear

that they belong together to implement the same functionality (image based shimming) but layers implementing this functionality in the architecture are not well separated.

- Often, an analysis of a pair of co-changed entities resulted in further analysis of the entity with a high number of internal or external relations. This happened for instance in Case 1, where the menu file and its co-changes were further analyzed, rather than the .pset file from the initial visualization. When lowering the threshold in Case 1 the developers in the working session noticed that the menu file have many more co-changes than the .pset file. So one of them remarked:

  > *"This menu file is related to all the other building blocks. It seems that it plays a central role in this issue."*

  As a result, the focus of the developers shifted from the .pset file and they started to think about what could have gone wrong with the menu file.

- Visualizing internal relations of software entities often helped developers understand the reasons behind co-changes. In one case (different from the three discussed above), files describing hardware configurations (.hcf files) in different subsystems were strongly coupled. Only when looking at the internal relations of those building blocks did the reason surface: the .hcf files co-change because they all have to adhere to the hardware model as implemented in another group of files.

- Changing the abstraction level in the visualization was the least used interaction during the sessions. Participants seemed to understand the relationships between building blocks, and between subsystems, without needing to aggregate co-changes to those levels.

- Of all types of interaction supported, adjusting the threshold value to see co-changes of entities at different strengths was most often used.

- Deciding on how to resolve confirmed unwanted couplings was a non-trivial task. Often, many factors had to be taken into consideration, such as the practicality and impact of a fix before it was sanctioned.

From the observations above we conclude the following lessons learned:

- The actual interactions played a key role in understanding the unwanted couplings: setting the support threshold in cases 1 and 3, and changing the layout and hiding/showing the internal relations in case 2. Though some of the interactions provided were not used (notably panning and zooming), the limited experience we have gained does not suffice to decide **which** interactions are needed to reason about and resolve unwanted couplings.

- Interactions help build up, check or reason about the mental model the experts have of (part of) the system. This is paramount in Case 2, where the expert used the interac-

tions to verify his assumptions. Next, when these assumptions turn out to be true, such increases the trust experts have in the proper identification of the unwanted couplings.

5.6 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: our study is not a controlled experiment. In particular, we were not able to pick a random set of unwanted couplings. We had to select the unwanted couplings that had top priority for the architect. Only then could we involve him and his developers 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 real architects and developers, on real issues they are confronted with, and describe our experiences therewith.

For the same reason, we could not evaluate the same set of unwanted couplings using another visualization tool, to verify whether the use of iVis really matters. On the other hand, we are not evaluating yet another visualization tool, but the interactive visualization as such.

There may be a bias in that we evaluate our own tool. The extensive feedback from the architect and developers, reflected in the quotes given, alleviates that to some extent.

Using our interactive visualization tool is not the only factor that contributed to the success of our working sessions. For instance, we created an initial layout with the help of the software architect and developers used the ClearCase diff tool and source code browsing. Furthermore, we invited those developers to the working sessions who worked with the related part of the software system and therefore were knowledgeable about it. This is also reflected in the second lesson learned above: the tool does not serve as a learning environment for novices, but as a means to build and improve the mental model experts have of the system they work with. To know exactly the extent to which the interactive visualization contributed to our success would require further controlled experiments. However, the environment in which we carried out our research is a highly competitive commercial environment, which at this point in time does not enable us to do such experiments.

With respect to *external* validity: our results are derived from ten working sessions on a single large system in a single environment, and further studies are required to validate their generalizability to other settings. We have, however, assumed relatively little about the development processes and the data sources behind the visualizations. To test this hypothesis, we applied our process to the archive of ArgoUML. We extracted the development history of the first 8 years of ArgoUML (the same period covered in [Beyer and Hassan, 2006]). We only

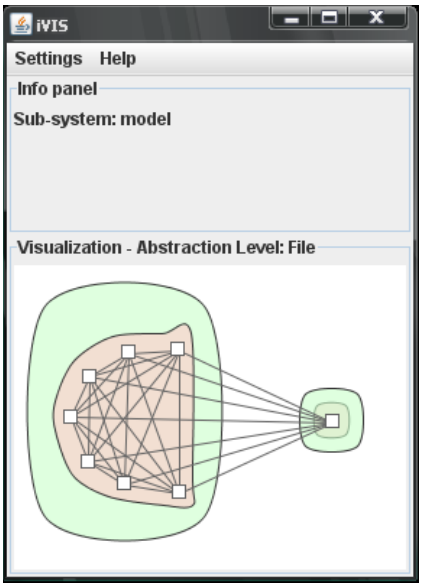


Figure 5.11: Unwanted couplings in ArgoUML

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 discussed in section 5.3.1 to the version data of ArgoUML. The process was straightforward, since the version management system used for ArgoUML allows us to extract which files were changed together.

Figure 5.11 depicts the unwanted cluster that appears at the top after the hierarchical clustering. It shows 10 property panel files from one subsystem, and one file on well-formed rules from another subsystem. This file is called `AbstractWellformednessRule.java`. Interestingly, the comments of this file states:

> *In several build methods in the uml factories these rules are used. Furthermore they are used in the proppanels to veto some changes.*

Note this only shows that our tool can be used to visualize unwanted couplings of another system as well. The role of **interactive** visualization to determine underlying cases and remedies has not been tested, since we lack the possibility to iteratively explore the issues with ArgoUML experts.

Some of the interactions supported seem to be more useful when analyzing really big systems. For instance, setting the abstraction level was not needed when visualing the ArgoUML data, since there are really only two levels: subsystems and files.

5.7 Conclusions

Evolution of a large scale software system is often a challenge for the system's architects. In this chapter, we have illustrated how interactive visualizations can support architects to reason about and discuss potential unwanted couplings in the system and arrive at solutions to resolve them.

We used a tool (iVIS) that allows software developers to visualize co-changed entities from the system and interact with these visualizations. The tool supports adjusting the abstraction level, adjusting the threshold level of the number of co-changes, visualizing internal changes, and repositioning software entities.

To validate the usefulness of interactive visualizations, we invited the architect and developers of a large medical software system at Philips Healthcare to use iVIS to investigate unwanted couplings in the system. We observed that not only did the tool support the participants to reason about unwanted couplings, it also helped them consider various solutions and measure their impact w.r.t. evolutionary coupling. For 7 of the 10 discussed issues, a solution was proposed during the session with the developers and the architect.

Overall, we have illustrated that interactive visualizations can be used to not only check where unwanted couplings are located in the system, but also reason about them and evaluate available alternatives to fix them. The interactions also help users to build up and reason about their mental model of the system. Further experiments on other large-scale industrial systems are needed to further underpin these observations.

The case study also pointed out some areas where the tool could be improved. Most notably, the following requirements came up during our case study:

- show more information about co-changes,
- allow the user to annotate the visualization, and
- allow the user to manually remove spurious co-changes.