Interactive Exploration of Co-evolving Software Entities

Adam Vanya, Rahul Premraj and Hans van Vliet
Computer Science Department
VU University Amsterdam
Amsterdam, The Netherlands
{*vanya,rpremraj,hans*}*@cs.vu.nl*

***Abstract*—If a group of software entities co-evolve frequently, it may indicate a weakness in the system's decomposition. Such a group needs to be analyzed further to understand the underlying reasons for co-evolution, and to determine how to resolve the issue. Static visualizations are one way to support such an analysis, but they help only to a certain extent. In this paper we discuss how interactive visualizations can support the analysis process even better. We implemented a tool that interactively visualizes software evolution and applied it to a large embedded software system having a development history of more than a decade. Our experience in using the tool along with the architects and developers suggests that interactivity adds much value when analyzing groups of co-evolving software entities.**

I. Introduction

Improving the decomposition of a software system is one of the challenges software architects face. Typical reasons why an architect would want to improve the decomposition is that development groups communicate too much, there are delays in the deliverables of those groups and the scope of tests cannot be as limited as they should be. Clearly, these are some symptoms of a software decomposition which does not completely satisfy the requirement of evolvability.

In case an architect has to make the structure of a software system more evolvable, then he needs to follow some steps. Firstly, it may not be evident for the architect immediately where the evolution-type structural issues are in the system. Therefore he has to identify those issues. The method of Antoniol [1] and the dendrogram approach described in [2] are alternative ways to point out those issues. Both of those alternatives build on the observation that software entities which changed together in the past, or with other words *co-evolved*, can help finding structural weaknesses [3]. The approaches under discussion identify co-evolutions by using historical meta-data from version management systems.

Secondly, the architect may need to select those structural weaknesses for further investigations which are more severe than other ones. By using scenario-driven characterization of the co-evolving sets of software entities in our previous work [4] we address the selection of severe structural weaknesses.

Thirdly, the evolutionary clusters selected are essentially only sets of software entities, i.e. files. Those clusters need to be further analyzed to find out why those software entities co-evolved frequently. The reason recovered may be, for instance, a design decision taken in the past.

Forthly, the architect has to think about one or more alternatives to resolve the structural weakness he is selected to resolve. These alternatives can be, for instance, to take new design decisions.

Fifthly, before the architect decides to resolve a structural weakness he has to first understand how the current design, causing the evolution-related structural issue, is related to other requirements, like for instance performance or reliability. These requirements may be affected by the new design the architect wants to apply to improve evolvability. Therefore, some trade-offs may need to be taken. Alternatively, the architect decides not to address the evolution-type issue at all, because by changing the design to better accommodate evolvability, other - potentially more important - requirements are heavily affected, i.e. performance gets noticeably worse. We noticed that in practice it is not rare that these decisions are taken.

Finally, the architect needs to initiate some re-factoring activities. As a result of such an initiation, re-factoring activities get assigned to one or more future projects. For a project to carry out the re-factorings required the architects has to describe which (design) elements of the system are involved, what the current and the desired design is.

In the process described above it is essential that architects can get understand the root cause of a structural weakness. Only then is it possible to find a proper resolution for the weakness and finally improve the structure of the system. Interactive visualizations of co-evolving software entities may help identify the root cause of a a structural weakness and find solution alternatives. So far, however, those visualizations were only applied to support structural weakness identification.

In this paper, with the help of an interactive visualization tool we evaluate how interactive visualization of co-evolving software entities can help the architects and developers to find out why those entities changed together and what the possible solution alternatives are.

In order to observe and capture how the interactive visualization of co-evolving software entities helps architects and developers we organized meetings them. Only those developers were invited to the meetings who were knowl-

edgeable about the modification of the software entities under discussion. To identify who is a "knowledgeable" developer related to the evolutionary cluster under discussion we could use the input of the architect directly or the meta-data found in the version management system.

The remainder of this paper is organized as follows. Section II presents related work. Section III describes the study environment from which we take our examples and experience. Section IV describes how we extracted the data we used. Section V elaborates on what co-evolving software entities are. Section VI describes the interaction types we implemented in our tool. Section VII briefly describes the tool we used. Section VIII elaborates on the case study. Section IX discusses the lessons we learned during the meetings with the architects and developers. Section X describes the threads to validity. Section XI concludes this paper.

II. Related Work

When performing the literature study related to this piece of research we mainly looked for contributions where software entities and the co-evolution type relationships between them are visualized. The following paragraphs summarize the results of our literature study.

A. Interactions with Visualizations

Some previous work suggests to use a purely static visualization of co-evolving 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 the work of Gall et al. [5], Ratzinger et al. [6] and Andrejs et al. [7].

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 way it can also be observed how the evolution-type relationships between software entities evolved over time. Previous work supporting this type of interaction include [8]–[11]. Hindle et al. in [12] take the idea above 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 [11] and in the visualization tool of Dirk and Beyer [13].

In some previous work the selection of software entities and/or the couplings between them is supported. Based on the selection they typically provide additional information about the items selected. In case a file is selected, the Evolution Radar [8] 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 a basis to set the focus on the selected entity [8], [11], track the evolution of the entity selected [8], fold and unfold modules [11], [14] or to specify the input for vertical mining [15].

Another type of interaction described in the literature is that users of the visualization can specify which entities or the relationships between them to hide or show. In [8] it is possible to specify, for instance, whether all the files or only the source code files should be included in the visualization. The EvoLens tool of Ratzinger et al. [11] supports the user to hide couplings between files if the number of co-evolutions between those files does not exceed the pre-definited threshold.

B. 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 [11]
- to identify structural decay and the spread of cross-cutting concerns [10]
- to identify change smells [6]
- to explore the effect of changes that occur within a software system over time [12]
- to assess the complexity of the software [9]
- to uncover hidden dependencies [8]

We can see from the reasons described above that interactions have mainly been used so far 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 addressed in this paper.

III. Study Environment

To address our research question formulated in Section I we have identified and interactively explored co-evolving sets of software entities of a large and complex embedded software system. The system studied contains approximately 8 million lines of code in 34 000 files. Hundreds of developers from three different sites have been developing and maintaining the system for more than a decade. The complexity of the system has increased over time significantly and handling it has become a challenge.

The software system has been developed using mainly the following programming languages: C, C++ and C#. To identify co-evolutions between software entities we have extracted check-in related meta-data from the ClearCase version management system used to manage different revisions of the system. Historical data was available from the last nine years.

When the co-evolving set of software entities were identified we had to choose the level of abstraction for software entities. We decided to observe the co-evolutions of *building blocks* because architects were primarily interested in investigating structural weaknesses at that abstraction

level. Building blocks are the directories representing the next level of abstraction above individual files in the file hierarchy; see also [16].

In the software system studied the architect was interested to know structural weaknesses related to the development group decomposition. That decomposition is an abstractions above the subsystem level. Development groups are only allowed to modify the subsystems they are responsible for.

IV. Data Extraction

From the version management system (ClearCase) we extracted check-in related meta-data using the command line interface (cleartool). This meta-data describe who modified what, when and why in the software system. We collected data from the last nine years of development considering all the available projects and branches. Other than individual information on check-ins we could not extract, since which check-ins belong to the same commit or development task was not captured by the version management system.

We used the data extracted to approximate which check-ins belong to the same transaction [17]. Further, we tried to mitigate the effect of merges. We have done that by disregarding all the transactions which contain more check-ins than 100. This threshold was suggested by developers. Transactions were used to identify co-evolving sets of software entities, see Section V

V. Co-evolving Sets of Software Entities

Software entities, like files or methods, get modified because developers need to work on development tasks. The aim of those tasks can be, for instance, to resolve a problem report (PR) or to develop a new feature. If software entities were modified because of the same development task then they co-evolved once. Co-evolutions between software entities are typically identified using the meta-data available from version management systems. Often, the sources available to identify co-evolutions are incomplete. Therefore, in those cases we need to approximate which modifications belong to the same development task.

Based on the co-evolutions identified we can derive further facts. We can count, for instance, how many times two software entities co-evolved or what the measured probability is that those software entities will co-evolve in the future. These facts are then used in the literature to detect frequently co-evolving sets of software entities. The main purpose of identifying such sets is to use them assessing the structure of the software system. It may be a sign of a structural weakness if a frequently co-evolving set of software entities crosses the borders of high-level decomposition elements, like subsystems.

The detection of frequently co-evolving sets software entities is a mean rather than an ultimate goal. Those sets, together with the meta-data related, need to be presented and analyzed to understand why the software entities included changed together. We may find that a design decision is responsible for the co-evolution of the files. In that case, architects may change their design decision taken so that in the future those files in discussion will co-evolve less frequently. The visualization of a co-evolving file set example is depicted in Figure 1.

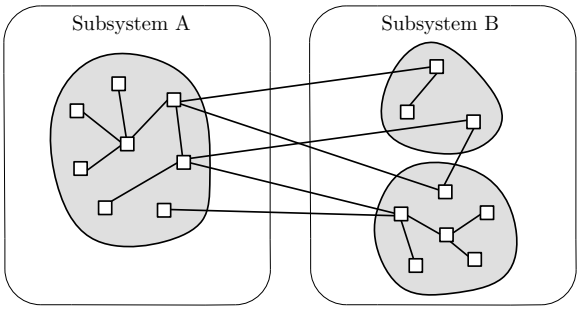


Figure 1: A co-evolving fileset

In Figure 1 we can see the files (indicated by rectangles) and the co-evolution relationship between those files. A line drawn between two rectangles indicate that the corresponding files co-evolved more than a threshold value. Furthermore, Figure 1 shows in which building block (gray blob) and in which subsystem (rounded rectangle) a file is located.

In the literature this visualization technique is altered and/or extended in different ways. In [1] the layout is defined such that files frequently co-evolving are more near in space than the ones which co-evolve less frequently. Therefore, lines between files and are not drawn. Also, the subsystem location of a file is indicated there by using coloring schema. In [15] the size of the squares represent in how many co-evolutions a given file was involved in. In [12] the width of the lines between files is drawn according to the number of co-changes between those files.

Although these visualizations are described to provide enough information for their purposes, interaction with them is either not possible or it is relatively limited. When available, the possible interactions include zooming in/out and selecting a subset of the software entities presented. In this paper we want to understand how different type of interactions with the visualization can help to identify why software entities changed together. We are especially interested in recovering co-change rationale related to design decisions. This way we intend to help the work of software architects. For our purposes, we apply the visualization technique indicated by Figure 1. In the next section we elaborate further on the interaction types we considered.

VI. Interactions with Co-change Groups

Interacting with visualizations opens up new possibilities. Although we know that a picture is worth a hundred words, sometimes we just need more information than provided by

that single picture. We may be triggered by a pattern we identify on one picture and we may need another one to verify what we presume. Also, information from different pictures may complement each other and help us to infer additional conclusions. Furthermore, these different pictures have to be available at request, since we need to remember the content of more pictures at the same time to combine information.

Actually, interacting with visualizations is something we do almost every day and therefore it comes very naturally. It is enough to think about Google Maps where we can zoom in/out, change views, overlap different views and so on. Our idea is to exploit the potentialities of interaction while visualizing co-evolving software entities. One may interact with the co-evolving software entities visualized in different ways. This section describes those types of interactions which we take into account. We do not strive to describe all the possible interaction types here. The selection we provide reflects what we find important to include.

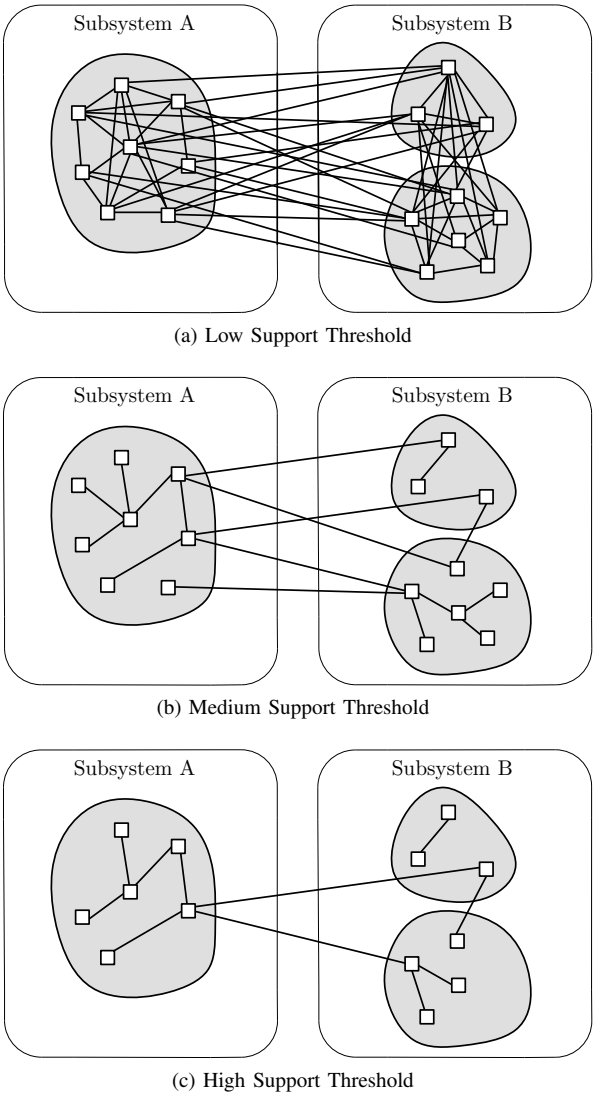
A. Adjusting The Support Threshold

The number of times two software entities co-evolved is known to be the *support* of the relationship between those entities [18]. Often, it is useless to indicate all the relationships where the support is at least one. The reason is that such visualization is usually overly scattered and therefore it is difficult to make any use of it. Figure 2a demonstrates our point. Hence, the user of the visualization needs to be able to set a threshold for the support value. As a result, only those relationships are indicated between software entities where the support is bigger than or equal to the threshold set. It is difficult to say in advance, which threshold value to apply. If the threshold is too high, see Figure 2c, then we may end up only with a couple relationships showing too little information. By interaction the user of the visualization simply needs to experiment which threshold value is optimal, see Figure 2b.

The optimality of the threshold depends on our intentions. Sometimes we just want to know where to start analyze the co-evolving file set. To answer this question, we may want to see the first few strongest relationships crossing the borders of subsystems. In another case we may wish to identify a co-change patterns between software entities, like the ones described in [6]. Furthermore, experts sometimes want to validate their assumptions on relationships having a lower support than the current support. These assumptions often take the form of: *I would expect also files A, B and C to co-evolve with file D*. The validation of these type of assumptions can be done be changing the support threshold.

B. Visibility of Internal Relations

When software entities are located in the same subsystem or building block then the co-change relationship between those entities are internal to the common subsystem or building block respectively. Both hiding and showing the internal relations has its advantages when analyzing co-change clusters.



(a) Low Support Threshold

(b) Medium Support Threshold

(c) High Support Threshold

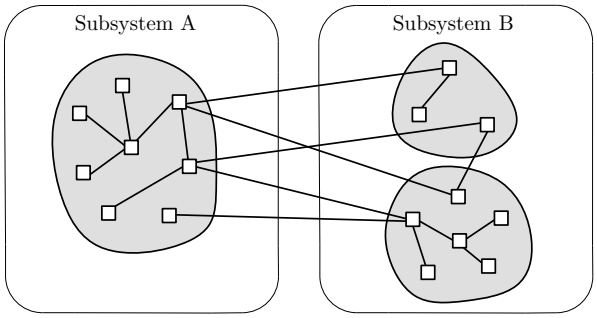
Figure 2: Support Threshold

On the one hand, hiding those relations which are internal makes the visualization less scattered and therefore easier to understand. Also, relations crossing the borders of high-level decomposition elements, like subsystems, are responsible for the structural weakness and those relations are the ones we want to get rid of or at least mitigate.

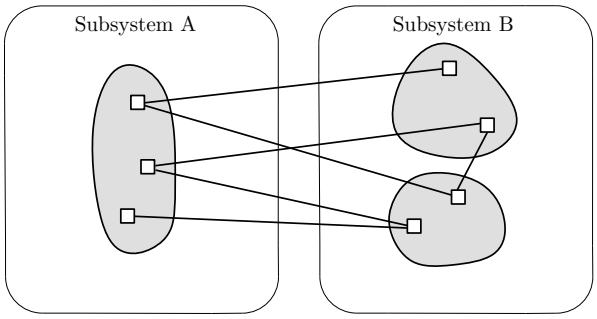
On the other hand, showing internal relationships may help the developers to recall the role of a software entity or to identify a co-change pattern. Knowing the internal relations may also allow the developers and architects to investigate some what-if scenarios. Let us imagine that architects want to evaluate what would happen if a file was

placed into another subsystem. By looking at the internal relations they could see immediately which of them would become external. Consequently, architects could decide if moving the file in discussion would mitigate or intensify the structural issue being analyzed.

Instead of creating a visualization where only one of the options above is implemented we suggest to provide the user the possibility to display or hide internal relations on the fly. The user also has to then specify the scope of internal relations, for instance building block or subsystem. Figure 3 demonstrates how an example visualization could look like with and without the building block internal relations, see Figure 3a and Figure 3b respectively.



(a) Internal Relations Present



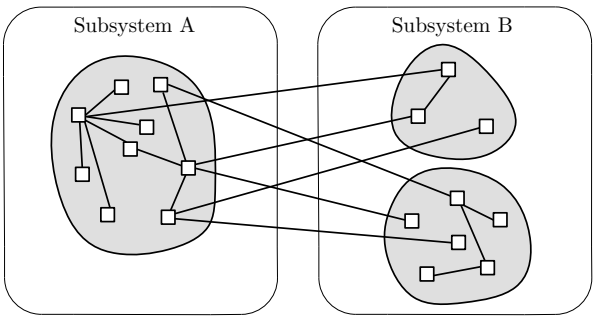
(b) Internal Relations Hidden

Figure 3: Showing and Hiding Internal Relations

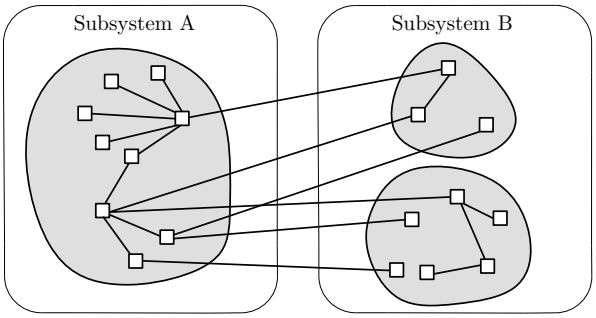
C. Dragging Software Entities

During this type of interaction the user can select a software entity and drag it to a new spot. The resulting graph is isomorphic with the original. Also, the location of the moved entity with respect to the system structure is not affected. We use here the general expression "software entity" on purpose; we are talking about the re-location of files, building blocks, subsystems, etc. Figure 4 shows an example on how one may re-arrange the layout of the visualization by dragging files around.

Interactive modification of the software entity layout may help the users in many ways. Firstly, with entity dragging



(a) Before Dragging



(b) After Dragging

Figure 4: Re-arranging the Entity Layout

developers have the possibility to group files together implementing a given design elements. Later on developers can analyze the co-changes between those design elements. As the same file may implement more than one design element a developer may even want to analyze different groupings.

Secondly, some software entities receive the attention of the developers because those entities have extraordinary properties, like strong relations with many other entities. Those interesting entities can then be placed such that they also visually get more attention. This functionality may be especially useful when the snapshots of the visualization are used to discuss the issues recovered.

Thirdly, sometimes software entities seem to be connected by a line between them, however an entity can be only accidentally placed above the line representing the coupling between two other entities. These type of issues can be resolved if it is possible to somewhat adjust the layout.

Lastly, the user can apply the dragging of software entities to express their mental picture about the part of the software system being analyzed. It means that for instance building blocks can be placed above each other to express that the intension is to have a layered architecture.

D. Adjusting the Abstraction Level

At a given point of time the visualization discussed shows software entities with potentially different levels of abstractions. In Figure 4b, for instance, we can see subsystems,

building blocks and also files at the same time.

Co-change groups can get, however, relatively complex to understand. This is typically the case when there are more then two subsystems and many building blocks are involved. In such a case it may help the user to analyze the relations between building blocks and subsystems first and consider the relations between files afterwards. Setting the lowest level of software entities to be presented on the fly seems to be therefore a useful interaction to be further investigated. Figure 5 depicts a co-change group where the abstraction level threshold is set to the building block level.

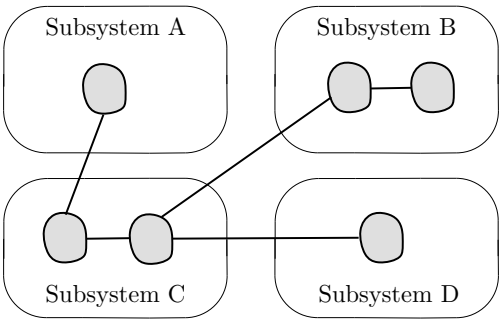


Figure 5: Abstraction Level Increased

VII. The iVIS Tool

In order to give the architects and developers the possibility to interactively explore the co-evolving groups of software entities identified we implemented a tool called iVIS. Next to visualizing the co-change groups, iVIS implements the interaction types as described in Section VI. A screenshot of iVIS is depicted in Figure 6.

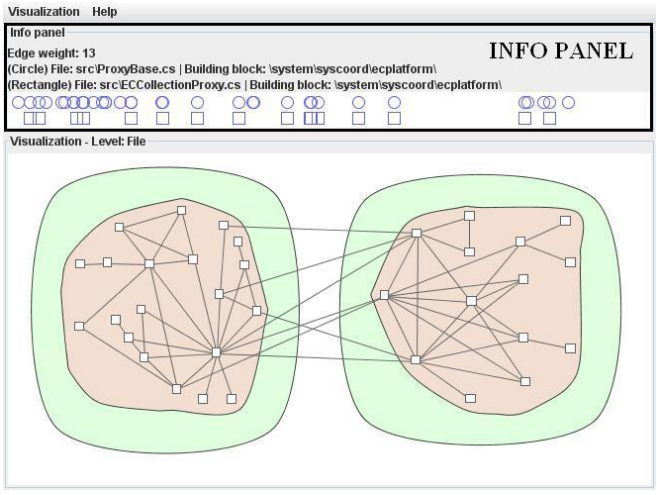


Figure 6: iVIS screenshot

When a software entity or a coupling coupling between those entities is selected then our tool provides additional information about the selected items. This can be the name of the entity, the modification distribution of entities over time or the number of co-evolutions between two entities. This information is presented in the "Info Panel" area surrounded by a rectangle in Figure 6.

VIII. Case Study

The objective of this case study is to understand how the different forms of interactions, as described in Section VI, may help architects and developers to recover the root cause of evolution-type structural issues. To reach our goal, we

1) implemented a simple interactive tool which can visualize co-evolving sets of entities as described in Section V. This tool also supports the forms of interaction which we discussed in Section VI.
2) identified and selected co-evolving groups of files. The historical meta-data available from the version management system of the medical software system studied was the input for this step.
3) organized 10 formal meetings with a software architect and one or more developers. During these meetings the tool was used to analyze co-evolving groups of files identified. The discussions were steered mainly by the information provided by the tool.
4) captured for every meeting how the tool, especially the interaction functionalities, helped the architect and the developers to understand the reason for software entities changing together.

From the four steps described above the last two steps are the ones where we believe our added scientific value resides. Therefore, in the following sections we further elaborate on those two. First, we describe the meetings organized where our visualization tool was used. Second, we give two concrete examples on how interaction with the tool actually helped to understand why files co-evolved.

A. Assessment Meetings

Prior to each of the meetings organized a preparation phase took place. During this phase we had to first determine which developers to invite to the meeting. As we organized the meetings to understand why software entities co-evolved, our main criterion to invite developers was that those developers should be knowledgeable about the development history of the software entities co-evolved. To achieve our goal we determined from the version management meta-data which developers co-evolved the software entities under discussion. Then we prioritized developers based on how many times they modified the software entities. The architect we had the connection with decided then which developers should be invited to the meeting.

Also being part of the preparation phase, we stared up our tool and visualized the co-change group to be analyzed. We set the initial threshold for the support such that only a few

relations between files from different subsystems remained indicated. This step was always necessary since it gave us a hint where to start discuss the structural weakness with the developers. The threshold applied was not always the same; we had to experiment with it every time. If it was needed to make the relations between files more clearly visible then we also adjusted the initial layout by dragging files.

At the beginning of each meeting we explained the developers invited what the purpose of the meeting was. We also briefly outlined how the visualization tool was meant to support this purpose and we gave a short demonstration on the possibilities to interact with the tool. Then the analysis of the co-change group began.

During the analysis the developers interacted with the tool to identify co-evolutions between files which needed explanation. Sometimes developers thought to recall the reason for the co-evolutions. To validate these initial assumptions or to get a clue on why file co-evolved, developers used additional sources of information, like the difference between two versions of a file or the change comments at the end of the files.

The role of the architect during the meeting was to abstract away from the domain level descriptions of the developers and to understand the design decision(s) which resulted in the co-evolutions. He also suggested and discussed possible design alternatives with the developers. The meetings usually finished with an agreement on if and how the structure of the software system needed to be improved.

B. Case 1: Reason for Co-change

The medical system studied includes several different pieces of hardware. There is, for instance, a magnet which produces the magnetic field needed to create medical images. The re-constructor is another example for a hardware component. One of its main responsibilities is to manipulate raw images such that the resulting images can be used by practitioners to set up a diagnosis for the patient under examination.

To add new functionalities to the system or to improve upon the existing ones, these hardware components and/or their software need to be updated and tested. Hardware related tests in our example are preformed by using an application with a graphical user interface. This application provides the tester with a menu structure where he can select and initiate the execution of tests related to the different functionalities of the hardware components available.

Of course, only those hardware tests can be executed from the application which have already been defined and for which there is an item in the menu structure. Defining a test is not straightforward. In our case, it is done by modifying (1) the file of test definitions (.res file), (2) the parameter file (.pset file) and (3) the outcome specification and acceptance criteria (.spec file). Every major hardware component has those files such that they can be tested.

The related co-change group we analyzed with a developer and the architect included one building block with the file defining the menu structure and two other building blocks containing .res, .pset a .spec files. The visualization which we prepared for the meeting looked like the one depicted by Figure 7.

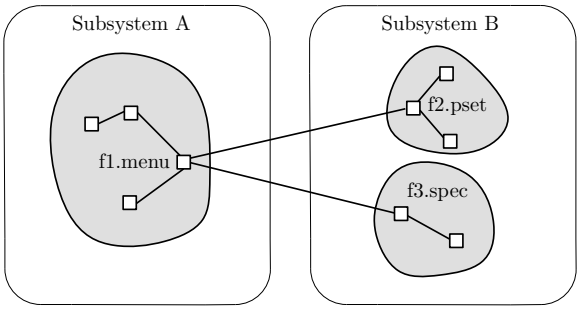


Figure 7: Initial Visualization

The developer invited started to look at the few subsystem crossing co-evolutions. He could explain what the menu file (f1.menu), parameter file (f2.pset) and the spek file (f3.spek) were about but he did not understand why those files would change together. To gain more insight the support threshold was lowered. Then the visualization indicated so many relations that it was of no help to guide our analysis. By setting the support threshold some what higher the patter indicated by Figure 8 emerged.

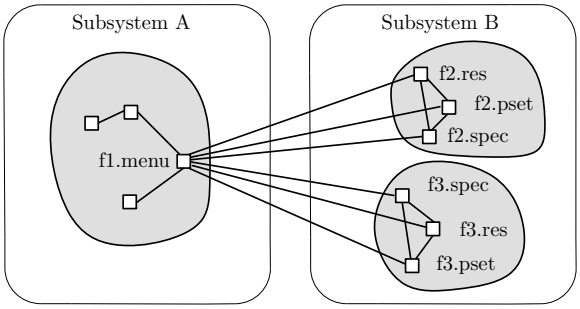


Figure 8: An Emerging Pattern

As we see from Figure 8, the menu file in subsystem A is related to the building blocks from subsystem B always through the set of file types. The developer now understood this pattern and could tell us that every time a new hardware test is defined the .res, .spec and .pset files are modified. Now that the motivation was understood it was also clear that to use the newly defined tests the menu file also needed to be changed. Simply the interaction with the tool in this case therefore helped the developer to recall why software entities changed together.

The architect considered the possible solutions to lessen the structural weakness identified and he discussed them with the developer. One of the those solutions were to automatically generate the menu structure from the tests available.

C. Case 2: Seeking for a Solution

The co-change group which we presented here to the architect and two developers is related to how medical examinations are supported by the system and what methods are used to capture medical images. During an examination a series of medical images are acquired. How each of those images should actually be captured is determined in advance when planning the examination. Reducing the time needed for planning is important. Given that many examinations are similar, examination templates were created to reduce the time needed for planning examinations.

The co-change group we analyzed in this case included seven building blocks related to the examination templates. Those building blocks implemented, for instance, the user interface, database and data model of the templates. What we observed with the developers is that the template building blocks were often changed together with a building block from another subsystem implementing the methods for acquiring images. The developers could recall why that happened: creating or modifying image acquiring methods introduce new possibilities to create or use examinations templates. Therefore, how those templates are presented and stored need to be changed.

In this case the main benefit of the interactions with the tool did not reside in help finding why files changed together, but in how the structural weakness identified could be handled. First, the developers instructed the tool to show internal co-evolutions. Then they manually modified the layout of the building block implementing the image acquisition methods by dragging files. As a result of the modification is pictured by Figure 9.

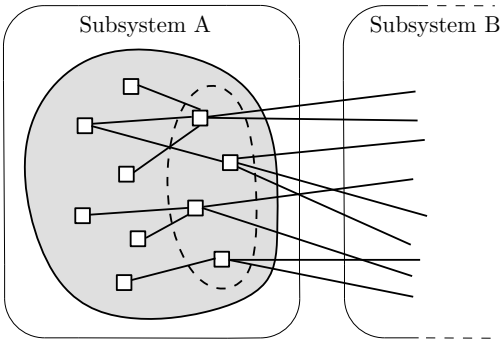


Figure 9: Grouping Interfacing Files

In Figure 9 Subsystem A contains the method building block, whereas Subsystem B accommodates the seven examination template related building blocks. Developers dragged the files together which were participating in a subsystem crossing relationship, see files surrounded by a dashed line. It turned out after dragging the files that those files were all the .NET .cs files in the building block and that they were all internally related to .c files. Because there were less relations between .c and .cs files then subsystem crossing relationships and those relationships where easier to maintain, developers decided to move all the .cs files to one of the examination template building block in subsystem B.

D. Developer Feedback

The architect and the developers using our tool were positive about our visualization, especially about the fact that the tool provides an interactive exploration of the co-change groups to be analyzed. It was a common pattern that either during the meeting or at the end, when the meeting was concluded, developers expressed their appreciation about the visualization without being asked. The architect itself used the following words to describe why he liked our interactive visualization:

E. Follow-up Actions by Industry

Our meetings with the experts of the software system not only resulted in a better understanding of why software entities changes together but based on this improved understanding experts took also further actions. In some cases, the architect started to think about how to resolve the structural issue identified and how other system requirements besides software evolvability would be affected by the solution alternatives. Based on these investigations the architect could make a tread-off weather or not to re-factor the software system. In other cases problem reports (PRs) were initiated for instance to move files from one building block to another.

IX. Discussion

During the meetings with the architect and the developers we have learned more lessons than which we could demonstrate by describing two concrete cases in Section VIII-B and Section VIII-C. In this section we give an overview on all the lessons we learned about the usefulness of interactive exploration of co-change groups.

- Considering all the meetings we had with the experts, setting the support threshold was always the most frequently used type of interaction. Often, the support threshold was adjusted to validated the hypothesis of the experts about the existence or non-existence of a relationship between software entities.
- Although initially we thought that setting the abstraction level of the software entities to be visualized would be a useful interaction type, developers rarely used this

feature. Developers seem to understand the relationships between building blocks without aggregating the co-evolutions between files.

- Initializing the visualization such that it shows the first few subsystem crossing relationships was a good idea since we could always start our discussions with the developers and those relations were the ones developers could typically explain without further investigation of the co-evolutions.
- As for some of the meetings, the architect could not be present and we learned that transferring the knowledge which the architect needs to understand from the developer is relatively difficult. One of the reasons seems to be that in the description of the developer we face an overwhelming amount of domain information and it is difficult to filter out the information relevant for the architect. Therefore we may conclude that the presence of the architect during such meetings is essential.

X. Threads to Validity

Our results are promising with respect to how interactive visualizations can help find the reasons for co-evolutions and potential solutions to resolve structural issues. However, the results we got are based only on ten cases discussed with the architect and developers. Considering how busy architects and developers are we still believe that have managed to collect a significant amount of evidence during the meetings organized.

XI. Conclusion

In this paper we have discussed how interactive visualization of co-evolving software entities may help not only to find structural weaknesses but also to help identify the underlying reason for the co-evolutions and find solution alternatives for the structural weakness.

To assess in which ways an interactive visualization can be useful we have organized meetings with the architect and some developers of the software system studied. During the meetings we have use the our interactive visualization tool (iVIS) to present co-evolving sets of software entities.

Based on the results of the meetings we argued that interactive visualizations indeed help to find out why software entities changed together. Also in some cases, as a result of interaction with out tool, developers could identify solution alternatives for structural weaknesses. Furthermore, we learned in what ways interactive visualization helped. We have elaborated on those lessons learned.

We received a positive feedback from the architect and the developers with respect to the application of the interactive visualization not only to identify but also to further analyze structural issues. As a result of our meetings some steps were decided to be taken to improve the system's structure. For instance, some problem report (PRs) were issued.

Acknowledgment

This work has been carried out as part of the DARWIN project at Philips Healthcare under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

References

[1] G. Antoniol, V. F. Rollo, and G. Venturi, "Detecting groups of co-changing files in cvs repositories," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE '05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 23–32.

[2] A. Vanya, L. Hofland, S. Klusener, P. van de Laar, and H. van Vliet, "Assessing software archives with evolutionary clusters," in *Proceedings of the 16th International Conference on Program Comprehension (ICPC '08)*. Amsterdam, The Netherlands: IEEE Computer Society, June 2008, pp. 192–201.

[3] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–198.

[4] A. Vanya, S. Klusener, N. van Rooijen, and H. van Vliet, "Characterizing evolutionary clusters," in *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE '09)*, 2009.

[5] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 13–23.

[6] J. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," in *Proceedings of the international workshop on Mining software repositories (MSR '05)*. New York, NY, USA: ACM Press, 2005, pp. 1–5.

[7] A. Jermakovics, R. Moser, A. Sillitti, and G. Succi, "Visualizing software evolution with lagrein," in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA Companio '08)*. New York, NY, USA: ACM, 2008, pp. 749–750.

[8] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, 2009.

[9] N. Hanakawa, "Visualization for software evolution based on logical coupling and module coupling," in *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–221.

[10] D. Beyer and A. E. Hassan, "Animated visualization of software history using evolution storyboards," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 199–210.

[11] J. Ratzinger, M. Fischer, and H. Gall, "Evolens: Lens-view visualizations of evolution data," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE '05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 103–112.

[12] A. Hindle, Z. M. Jiang, W. Koleilat, M. Godfrey, and R. Holt, "YARN: Animating software evolution," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '07)*, June 2007, pp. 129–136.

[13] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*. IEEE Computer Society, 2005, pp. 259–268.

[14] M. Lungu and M. Lanza, "Exploring inter-module relationships in evolving software systems," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 91–102.

[15] M. Fischer and H. Gall, "Evograph: A lightweight approach to evolutionary and structural analysis of large software systems," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 179–188.

[16] F. J. van der Linden and J. K. Müller, "Creating architectures with building blocks," *IEEE Software*, vol. 12, no. 6, pp. 51–60, 1995.

[17] T. Zimmermann and P. Weißgerber, "Preprocessing CVS data for fine-grained analysis," in *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR '04)*, Edinburgh, UK, 2004.

[18] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 73.