CVS Release History Data for Detecting Logical Couplings

Harald Gall, Mehdi Jazayeri, and Jacek Krajewski

Technical University of Vienna, Distributed Systems Group
Argentinierstrasse 8/184-1, A-1040 Wien, Austria, Europe
{gall,jazayeri}@infosys.tuwien.ac.at

Abstract

The dependencies and interrelations between classes and modules affect the maintainability of object-oriented systems. It is therefore important to capture weaknesses of the software architecture to make necessary corrections. This paper describes a method for software evolution analysis. It consists of three complementary steps, which form an integrated approach for the reasoning about software structures based on historical data: 1) The Quantitative Analysis uses version information for the assessment of growth and change behavior; 2) the Change Sequence Analysis identifies common change patterns across all system parts; and 3) the Relation Analysis compares classes based on CVS release history data and reveals the dependencies within the evolution of particular entities. In this paper, we focus on the Relation Analysis and discuss its results; it has been validated based on empirical data collected from a Concurrent Versions System (CVS) covering 28 months of a Picture Archiving and Communication System (PACS). Our software evolution analysis approach enabled us to detect shortcomings of PACS such as architectural weaknesses, poorly designed inheritance hierarchies, or blurred interfaces of modules.

1 Introduction

To evaluate the impact of changes over several releases of a software system, we need to understand the relationships (i.e. dependencies) among modules that compose the system. Most methods for identifying dependencies are based on metrics such as coupling and cohesion measures. There are two basic problems with these measures: 1) handling large code size per system release; and 2) recovering dynamic relations.

In fact, some dependencies are not written down either in documentation or in the code. The software engineer just "knows" that to make a change of a certain type, he/she has to change a certain set of modules.

Code-based measures reveal *syntactic* dependencies and what we are really interested in is *logical* dependencies among modules. The purpose of this paper is to present an approach to uncover such logical dependencies by analyzing the release history of a system. Release histories contain a wealth of information about the software structure. The task is just to analyze them and uncover the information.

The evolution of the studied software system was investigated on the level of *classes*; so classes constitute the measured entities for our analysis. The research on software engineering of the last few years suggests that not only the source code with its number of lines of code provides enough information about the complexity of a system, but we have to investigate on modules and programs for the measurement of software systems. Accordingly, classes as basic building blocks of object-oriented systems provide a good decomposition level for the assessment of the size and evolution of a system. Additionally, this level can be used to evaluate functional enhancements.

Our methodology, QCR, investigates the historical development of *classes*. The time when new classes are added to the system and when existing classes are changed has to be measured. Attributes related with changes of classes, such as the author or the date of a change, are additional inputs for our software evolution analysis approach. Changes made to the classes of the studied software system were inspected to reveal common change behavior. Such common change behavior of different parts of the system during the evolution is referred to as *logical coupling*. Through the assessment of classes we can evaluate modules or even the entire system, as they build up a hierarchical organization of classes. Thus, the software system as a whole can be analyzed, but also the system parts may be investigated and related to each other.

The case study used for our analyses was a Picture Archiving and Communication System (PACS), implemented in Java and consisting of half a million lines of code. The evolution of PACS was observed for a period of 28 months. All actions to the code base within this inspection period were noticed to serve as input for the described methodology. With our approach we were able to identify architectural shortcomings of the software system. Exam-

Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)
0-7695-1903-2/02 $17.00 © 2002 IEEE

ples for revealed architectural weaknesses were poorly designed interfaces, redundancy of functionality, god-classes with several thousand lines of code, and many others.

The defined methodology is composed of 3 complementary techniques: *Quantitative Analysis* focusing on observing change and growth rates, *Change Sequence Analysis* revealing parts with common change history, and *Relation Analysis* recovering dependencies by change-related attributes. The results of each technique are inputs for the other ones to define an incremental methodology where the results are improved in each step. Furthermore, each step provides new findings that have to be evaluated as well. All three techniques are examined during the evolution of the software within the inspection period.

The focus of this paper is on the *Relation Analysis*; results of the other two steps are described only to the extent required to understand the whole software evolution analysis results. Details of the other steps can be found in [14].

To validate the accuracy of these logical couplings (i.e. module dependencies) identified by our technique, we validated our findings with the software developers of PACS. The results have shown that our approach is promising in identifying (otherwise hidden) logical couplings among modules across several releases. Such modules are candidates for restructuring or reengineering. The technique requires very little data to be kept for each release of a system. Rather than dealing with millions of lines of code, it works with structural information about programs, modules, and subsystems, together with their version numbers and change information for a release. Such release data is both easy to compute and usually available in a company.

2 Related work

Based on the findings in [15,16,20], our Quantitative Analysis [8] and Change Sequence Analysis [9] uses modules as our unit of investigations, rather than the source code. Our goal is to identify logical coupling of modules that is otherwise hidden in the source code in terms of change patterns [9,12]. If programs change together across module or subsystem boundaries, the decomposition structure of the application should be reconsidered and possibly restructured or reengineered. In this work *classes* were our modules for the investigations.

Related approaches differ from our work in that they mainly focus on a micro-level to analyze the evolution of a software system: the source code is analyzed and source code metrics are used as indicators of the system's quality and complexity [19]. Other approaches identify fault-prone modules using statistical techniques based on design metrics [20] and discriminant analysis [13,18]. Fault and defect metrics are used for in-process project control and for process improvement over time in [3].

Coupling and cohesion measures are a way to measure structural cohesiveness of a design. The main purpose is to evaluate how maintainable a design and resulting implementation are, and to guide improvement efforts. The basic idea is that the more dependencies that exist among modules, the less maintainable the system is because a change in one module will necessitate changes in dependent modules. Approaches to measuring module dependencies fall into two categories according to the information on which it is based:

- *code-level* approaches measure coupling based on analysis of source code; naturally, such measures can only be made after the code has been written.
- *predictive* measures try to measure coupling based on design information; such approaches attempt to evaluate the complexity of the system *before* the code has been written.

Our approach attempts to measure coupling based on analysis of multiple releases of a system. This approach is based on observed change behavior of modules in a system and may be categorized as *retrospective*. Our measures may be used not only as coupling measures to guide restructuring efforts but also to validate the effectiveness of predictive and code-level coupling measures [8,9, 10,12].

Other related work analyzes the structure and the architecture of software systems. Methods for evolutionary architectural assessment such as [21] can be used as input for restructuring or reengineering activities.

Visualization approaches such as software change perspectives [4], sv3D [17], or evolutionary visualization [22] represent software by visualizing source code and change information in different ways.

The remainder of the paper is organized as follows: In Section 3 we describe the case study. Section 4 describes our approach for identifying logical coupling among modules based on CVS release history data. We report on our results in Section 5 and draw some conclusions and point out future work in Section 6.

3 The case study

A Picture Archiving and Communication System (PACS) was selected as case study for our approach. It is implemented in Java and different types of information have to be maintained: there are files that contain the source code of the particular Java classes. These Java classes have to be supported by configuration files, as the runtime behavior of the software system also depends on these configuration files; nevertheless, configuration information is differently maintained than source code.

The application development process also includes other types of files such as images that can be used as icons in an application or as background image of boxes and buttons. Such files are also necessary for the application but are quite static and are often just replaced instead of maintained.

As a result, our analysis concentrates on the Java source code to allow reasoning about architectural weaknesses of the software system. The metrics that are used for the identification of outliers, which may indicate structural problems, are therefore based on the files that contain Java classes only.

As inspection period for our software evolution analysis we selected 28 months from April 2000 until July 2002 as during this period the main parts of the application existed. At the beginning of the inspection period the studied PACS contained approximately 2.000 classes and at the end it had more than 5.500 classes.

The information of the whole application is maintained with the help of CVS. The software structure of the PACS is a tree hierarchy. The top level represents the *system level*. At the next level the application is composed of different *subsystems*. All subsystems may be viewed as separate projects. All these subsystems encapsulate some aspects of the whole application such as the viewing unit, the archiving process or extensions to the viewing unit. These extensions add diagnostic features to the viewing application.

The entities of the subsystem level are further divided into *modules*. These modules may additionally contain *submodules*. At the lowest level of this hierarchical structure are the Java *classes*. These contain the implementation of the application features. For example, 13.c.21.A denotes Class A in Submodule 21 of Module c in Subsystem 13. *The units of our software evolution analysis are classes*. Thus, for example, the size of a subsystem, module or submodule is measured on the basis of classes. Also changes are tracked on the basis of classes. The case study is composed of 35 subsystems, each containing between one and 14 modules. Modules are further subdivided in up to 29 submodules each; between 1 and 196 classes are assembled into a submodule. As subsystems vary much in size, several submodules are even larger than the smallest subsystems.

With the help of the hierarchical structure of the software system the vendor can support entire product families. A PACS may contain many different workstations that support different features. Some could be used as viewing units only, where a doctor can view X-ray pictures of a patient and draw some conclusions from them. Other workstations may additionally provide diagnostic features, i.e., to allow a doctor to change the images, mark a particular region of an image to show a colleague where he recognized a problem, or sort the images and arrange them into new sequences to support further diagnosis. However, the system includes more than just workstations. An archive is necessary to store the X-ray images and the administrative information. The data of the images has to be received from X-ray recorders. The images are assigned to different patients, where information for each patient has to be kept and maintained. Thus, products with different capabilities may by assemble to build up a product family.

3.1 Release history data in CVS

The data about the evolution of the studied software was taken from a Concurrent Versions System (CVS). CVS allows handling of different versions of files in a cooperating team of developers. Each member of such a cooperating team can check out some files, change them and then check them in again. When the files are checked in, CVS merges the newly added content to build up a consistent view of the whole work [7].

All operations that are performed with the help of CVS are logged automatically by the Concurrent Versions System. Thus, the historical development of files, which are maintained by CVS, is traced automatically and may be viewed with the help of appropriate CVS commands. These commands provide information about the history in many different levels of granularity. On the one hand CVS allows for every line of any file to find out in which release the particular line was introduced. On the other hand it is possible to get informed when an entire file changed its release number. Every time a file is change and checked in into CVS, the release number of this file is incremented. Thus, changes of files may be tracked through the release numbers [1].

For each change of a file administrative data is collected by CVS (date, time as hours and minutes, the author, the release number, the file name and the path associated with a single change, etc.). Based on this information a time sequence analysis is possible, which can be enhanced with additional attributes such as the author of a change. As a result, the first challenge was to extract useful historical information to reason about architectural deficiencies of the software system.

4 The QCR approach

In this section, we give an overview of our QCR methodology for identifying common change histories among modules and revealing hidden dependencies among them. To do that, we define 3 techniques that use the CVS data. We give an overview of the 3 techniques here and describe one of them in detail in this paper.

1. The *Quantitative Analysis (QA)* analyzes the change and growth rates of modules across releases and provides indicators for noteworthy change or growth intensities (e.g. outliers).

2. The *Change Sequence Analysis (CSA)* identifies common change histories of modules. Each change of a module (reflected in a change of its version number) is related to the system level (with system release sequence numbers). All changes of a module can then be viewed on the system level and put together to form a change sequence. A change sequence for a module shows the releases in which the module has been changed, e.g. <1,2,5,9,15>. Such change sequences allow comparing different modules in terms of their change history and identify common "change patterns." The output of the CSA process is a set of logical couplings among specific modules that follow the same change pattern. For details we refer to [9].

3. *The Relation Analysis (RA)* compares modules (i.e. classes) based on CVS change history information and reveals module dependencies. RA is based on the previously developed QA and CSA; it enhances and complements them.

The complete QCR results of the case study can be found in [14]. In the following, we are focusing on the Relation Analysis as the main contribution of this paper.

5 Relation Analysis (RA)

The Relation Analysis (RA) was developed to complement the Change Sequence Analysis (CSA). Both analyses support the evaluation of a software system based on historical module interdependencies. Logical coupling refers to patterns of change that are similar or even equal in different parts of the software system.

The RA method tries to incorporate more details of changes applied to each piece of software to gain improved results of the analysis. Thus, within RA the comparison of changes is not only based on the time of check in, but also on the author that actually carried out the particular change. As we will see with the help of RA we can verify some of the results of CSA and even get better and fine-grained results. The logical coupling between different parts of a system points to structural shortcomings. Especially, the relations between separate modules may be an architectural weakness that requires attention.

5.1 The Relation Analysis (RA) approach

While examining the system for common change patterns the attention was drawn to the modularity of the software system. The evolvability can be preserved or improved with a well-formed architecture composed of self-contained software components.

Thus, an ideal situation would allow changing each component independently of the others. If changes are necessary, the smallest possible set of components should be involved in a particular change. In the Relation Analysis (RA) single classes and their historical development are investigated in detail. The evolution of classes is compared to find those classes that were most frequently changed together. Therefore, the changes of each class are compared with the changes of other classes. This comparison is *based on the author name and the date and time of the check in of a particular change*. Each change was considered based on the exact date and the author of the change.

This selection was based on the fact that the analyzed software system was developed with strong ownership of code. A developer was responsible for a particular part of the software system. Thus, a necessary change for a requested improvement of the software was carried out by a single developer. As a result, the comparison of the authors of changes was expected to lead to good results. Additionally, within RA the date of each change of a class is compared with the dates of changes of other classes to discover equal change dates. Dates are compared on equality, but a *time window of four minutes* was chosen, because a check in of a large module takes a while, the according files may get different time stamps.

All changes that are done on the same date and by the same author point to a logical coupling. The more such correspondences can be found between a particular group of classes the stronger is the postulated relationship. This logical coupling can be represented as number of common changes, which defines the "strength" of the logical connection.

The necessary information about changes was extracted from CVS. An excerpt of a CVS log file is shown in the following textbox. For each file the name and path is stored; further some administrative data such as latest version (head) and the number of releases (total revisions) are provided. To compare changes of different classes, the date and author of each change were taken into account for the Relation Analysis. Other interesting information, which could be used for a very detailed investigation, can be the number of added and removed lines of code and the message, which the author of the change included.

```
RCS file: /cvs/SubsystemName/Module/Submodule/OptionalBooster.java,v
Working file: SubsystemName/Module/Submodule/OptionalBooster.java
head: 1.3
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 3;     selected revisions: 3
description:
----------------------------
revision 1.3
date: 2003/01/20 21:43:48;  author: john;  state: Exp;  lines: +15 -4
message: some fixes to arbitrary clipping plane
----------------------------
revision 1.2
date: 2002/11/06 21:52:59;  author: paul;  state: Exp;  lines: +30 -12
message: addition of arbitrary clipping plane
----------------------------
revision 1.1
date: 2002/08/02 10:47:40;  author: john;  state: Exp;
message: native shear support added
```

An interesting aspect of coupling is the distinction between *internal* and *external links*. We define internal coupling as a dependency that happens between classes in respective parts of the system. E.g. the relations between classes of a single module and its submodules are defined as internal couplings. The connections between classes within this module and any other part of the software like another module or another subsystem are considered as external couplings.

With the measures of the relationships between different system blocks we try to answer the following queries. These queries were defined to find outliers to detect architectural anomalies:

- Which parts of the software system have been changed together most often?
- How many classes are involved in an external coupling with a particular module?
- Is the internal or the external coupling stronger?
- Do couplings mostly concern internal or external classes?
- Does a central class exist to which most classes are related?
- Do some modules have significantly more relations than other modules?
- Are there outliers of the previous steps of the analysis (QA, CSA) that cannot be verified by the Relation Analysis?
- Which couplings can be revealed with the help of RA, but could not be found with the help of CSA?
- Are there some other suspicious parts of the systems in RA, which were not recognized within the other 2 steps of the analysis?

As evolvability of the system is a main issue, the attention is directed towards classes with many changes. On the average every class in this software system was changed five times. Thus, we compare only classes which have more than five changes during their development history.

5.2 Evolution observations

The QA and CSA steps of the evolution analysis—not described in this paper due to space limitations—revealed certain structural deficiencies of the analyzed software system. This section describes the observations within the Relation Analysis (RA) to analyze a software system much more fine-grained. In this section we describe the findings of logical coupling via RA that could not be recovered, although the Quantitative Analysis (QA) showed interesting change and/or growth rates in the affected parts of the software system. The results will be analyzed in detail and, furthermore, we will discuss the strongest relations, which were found only with the help of the Relation Analysis. The results will be visualized by graphical representations

5.2.1 Logical coupling based on QA and RA

Two submodules of Subsystem 29 were identified as outliers by the Quantitative Analysis. Further, two submodules of Subsystem 13 could not be recognized with CSA but with QA and RA. So, when we investigate the system by means of RA it can be shown that there are additional logical dependencies.

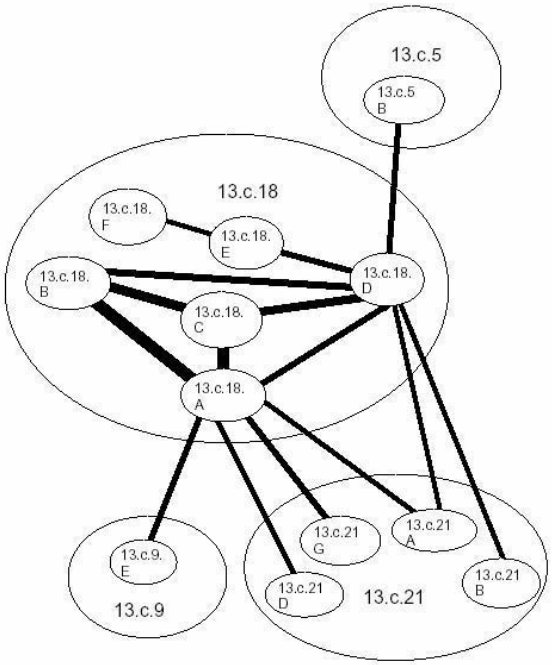

Fig. 1 Couplings of Submodule 13.c.18 (Nodes are classes and linkage is defined by logical couplings; thickness of lines represents coupling strength)

The information about the evolution of Submodule 13.c.18, depicted in Fig. 1, includes all discovered corre-

spondences with strengths greater than 8 between classes of Submodule 13.c.18 and other parts of the software.

In Fig. 1 the *focus* is on Submodule 13.c.18 meaning that the picture just shows dependencies pointing from this Submodule to other classes and Submodules, but not a complete bi-directional view is provided. The stronger the dependency is, the thicker is the line connecting two classes (i.e. nodes). Additionally to the internal relationships, each figure includes the external ones of the analyzed software part. Again only strong external connections are shown in the graphical representation. Thus, external and internal couplings are displayed, whereas the lower level of strength is chosen to be equal for both types. In Fig. 1 all couplings with strengths greater than 8 are combined to provide a more homogenous picture.

In Fig. 1 Submodule 13.c.18 has several internal and external couplings. A crucial point is Class 13.c.18.A that has strong internal coupling. The strength of the connection is measured by the number of common check-ins. So, Class 13.c.18.A was 21 times checked in together with Class 13.c.18.B and Class 13.c.18.C. Although a relation with coupling strength of 21 may seem not that important with respect to the maintenance of the entire software system, the connection is still strong, as the classes of Fig. 1 change 40 times on average during their evolution.

Class 13.c.18.D is an important outlier, because it has many internal couplings and additional external couplings both with Submodule 13.c.5 and Submodule 13.c.21 (with strength of 9 and 10 common check-ins, respectively).

For external coupling, only 2 classes (i.e. Classes 13.c.18.A and 13.c.18.D) of Submodule 13.c.18 are related with other parts of the software system. It is noteworthy that from a more general point of view Submodule 13.c.18 is several times coupled with Submodule 13.c.21. All the subsystems involved in the interdependency of Submodule 13.c.18 were also outliers of the Quantitative Analysis done in the first step of QCR [14]. So the prior results were confirmed with RA.

Submodule 13.c.9 was an outlier of the Quantitative Analysis but could not be filtered out with the Change Sequence Analysis: it has the third highest changing intensity and a high changing rate. Fig. 2 depicts the relations of Submodule 13.c.9 that were discovered by means of RA. Again complementary results were achieved.

The coupling of Submodule 13.c.9 shows a totally different picture than the relationships of other parts of the software system. Submodule 13.c.9 has more external than internal links. There are even twice as many external couplings than internal ones. Two classes of Submodule 13.c.9 have only internal relations. The classes of Fig. 2 changed on the average 100 times during the evolution of the system.

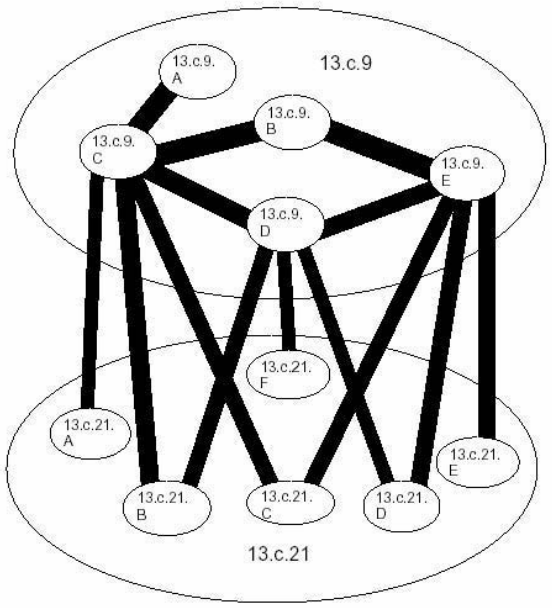

Fig. 2 Couplings of Submodule 13.c.9

Class 13.c.9.C has the most interdependencies of all classes of Submodule 13.c.9: it has 3 internal and 3 external relations whereas the internal couplings are slightly stronger. Class 13.c.9.E is the only class that has light relations with Submodule 13.c.18, which is depicted in Fig. 1.

All three classes of Submodule 13.c.9 that have external relationships show a similar behavior: they are each connected with (three) classes of Submodule 13.c.21. So, the external coupling from Submodule 13.c.9 to Submodule 13.c.21 is quite intensive and spread over several classes of 13.c.21. Submodule 13.c.21 itself is already coupled with Submodule 13.c.18, which was previously analyzed.

As a result, Submodule 13.c.9 has strong internal coupling (which looks reasonable at first sight) but also several external couplings with Submodule 13.c.21 that should be further investigated for architectural shortcomings or design erosion.

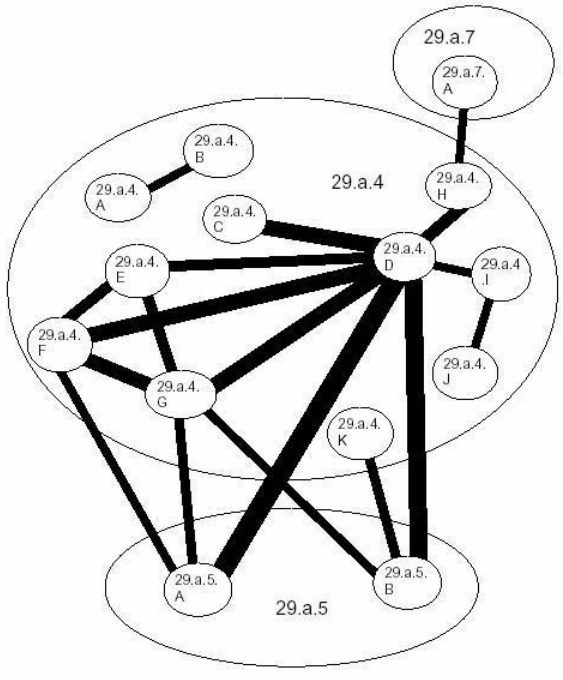

Fig. 3 Couplings of Submodule 29.a.4

The next part of this section analyses two other submodules that were outliers in the Quantitative Analysis, but for which no common change sequences were found.

Submodule 29.a.4 is the first and depicted in Fig. 3. The first outstanding sign is the manifold internal coupling around the dominating Class 29.a.4.D. Additionally, this class has strong external couplings with two classes of Submodule 29.a.5. The strongest relations exist between Class 29.a.4.D and Classes 29.a.5.A and 29.a.5.B (with strength 30-35 common check-ins).

The classes of Submodule 29.a.4 that are involved in the coupling with Submodule 29.a.5 constitute the framework for *accessing storage media*. These connections are quite strong compared with the internal relations of Submodule 29.a.4. Submodule 29.a.5 on the other hand forms a *dictionary for protocol translation*. Submodule 29.a.4 does not only exhibit interdependencies with Submodule 29.a.5, but also with Submodule 29.a.7 (with strength 15). The link with Submodule 29.a.7 is weaker than the coupling with Submodule 29.a.5, because Class 29.a.4.H encloses data that is used to display images to the user.

Submodule 29.a.5 is the second submodule that was identified during CSA, where no common change sequences could be found, although these submodules were outliers of QA. When analyzing Submodule 29.a.4 we could already spot Submodule 29.a.5; there existed some strong connections between these two submodules (see Fig. 3).

In Fig. 4, again the dependency between Submodule 29.a.5 and Submodule 29.a.4 is evident. Nevertheless, the internal coupling of Submodule 29.a.5 between Class 29.a.5.A and 29.a.5.B is the strongest relationship (with strength 64). This is among the strongest couplings measured with the help of RA; Class 29.a.5.A was checked in 78 times and Class 29.a.5.B 86 times in total.

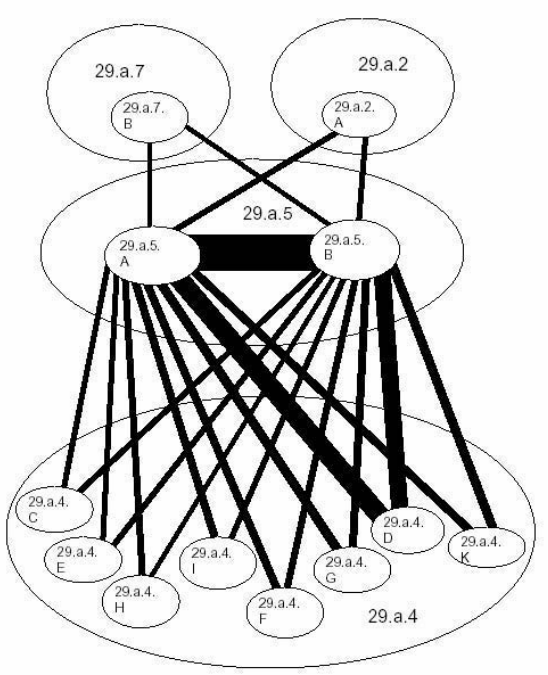

Fig. 4 Couplings of Submodule 29.a.5

Fig. 4 shows a very symmetric coupling. Both classes of Submodule 29.a.5 have 10 external couplings. Each class is coupled with 8 classes of Submodule 29.a.4, once coupled with Submodule 29.a.7 and once with Submodule 29.a.2. In Fig. 3 the focus was on Submodule 29.a.4 and in Fig. 4 the focus is on Submodule 29.a.5; so both figures complete our observations about these two Submodules, one from each direction (focus). The result shows a strong internal coupling and many external couplings with classes of Submodule 29.a.4 that should be further investigated.

5.2.2 Couplings based on results of CSA and RA

This section addresses the interdependencies identified by CSA and RA. During CSA some interesting relationships were found for Submodule 13.c.5, depicted in Fig. 5.

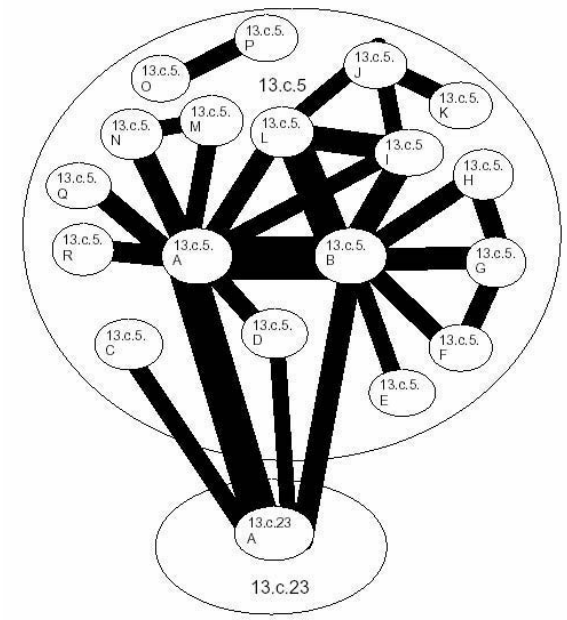

Fig. 5 Couplings of Submodule 13.c.5

In Fig. 5 we see many connections inside Submodule 13.c.5. The structure is dominated by the relation of Class 13.c.5.A to Class 13.c.5.B (with strength 78). This represents the strongest coupling that was identified during RA across the whole case study. The other entities within Submodule 13.c.5 gather around these two classes and produce a star-shaped image.

During CSA the central relation between Class 13.c.5.A and Class 13.c.5.B could not be recognized; both classes had other changes additional to the changes done on both classes together. Thus, the change sequences of these classes had commonalities, but each change sequence of these two classes also contained changes in months different to the change sequence of the other class. The strongest coupling that could be discovered with CSA was the connection between Class 13.c.5.O and Class 13.c.5.P. Although they were checked in together only 32 times, these classes were changed always in the same months of the inspection period.

A strong divergence could be identified between the results of CSA and RA concerning external dependencies. CSA identified dependencies between Submodule 13.c.5, Subsystem 11 and Submodule 13.c.5. However, no dependencies between these two parts can be identified with the help of RA. A common change sequence of 10 months linked Module 13.a and Submodule 13.c.5. A logical coupling between these two entities may be identified through RA, but this link is very weak, as it contains only 6 common changes.

Within CSA a weak dependency was identified between Submodule 13.c.5 and Submodule 13.c.14. A common change sequence of 5 months matched classes in both submodules. This coupling can be verified with RA (with strength of 24).

The strongest external dependency of Submodule 13.c.5 is the relation with Submodule 13.c.23. It is the only coupling that was strong enough to be covered by Fig. 5. Between Submodule 13.c.23 and Submodule 13.c.5 common check-ins occurred 66 times. Both submodules contain a class, which changed within each month of the evaluation period.

Fig. 5 describes only one type of external coupling of Submodule 13.c.5: the strong relationship with Submodule 13.c.23 via 4 classes. In QA we recognize that Submodule 13.c.23 has a high changing rate and a high changing intensity. This high changing activity was mainly caused by one class within this submodule which exactly forms the connection to Submodule 13.c.5.

5.3 Validation of the Relation Analysis

We evaluated our findings to find explanations for the figures received with the help of RA and, therefore, integrated domain knowledge to evaluate which kind of structural inadequacy is responsible for the affected part of the software system.

Submodule 13.c.18 as a self-contained unit provides the implementation of the *printing framework* of PACS. It receives information from the user about the data that should be printed including the appropriate images and the requested details to be printed. The printing itself is executed as a separate batch process within the application. For the communication with the user, classes such as 13.c.18.A are necessary. It is interesting that the entire Submodule 13.c.21 implements many features for the interaction with users. Thus, the fact that Submodule 13.c.18 has logical coupling with Submodule 13.c.21 is not unusual. However, Class 13.c.18.D is part of the threading engine and, therefore, should not be tightly coupled with the user interaction. Especially the necessity to change classes of the threading engine together with classes of the user interaction framework is probably an undesired property of the evolution of the system and the use of patterns such as the Model-View-Controller (MVC) should be considered.

Submodule 13.c.5, which is part of the *imaging framework*, also has dependencies on Submodule 13.c.18. The imaging data have to be handled by the printing framework to send the data to printers. Class 13.c.18.D is part of the threading engine and should therefore not be related to the imaging framework.

During the Quantitative Analysis Submodule 13.c.18 could be discovered as the submodule of Module 13.c with the highest growing and changing rates. These striking measures were traced to the difficulties of printing to

many devices with different capabilities. With the help of RA it is further recognizable that the submodule bears architectural deficiencies based on the separation of concerns within the software.

Thus, it would probably be necessary that the threading engine is subject to restructuring or reengineering. A possible approach could be to extract the threading part of Submodule 13.c.18 and build a separate component for threading. Such a component needs a well designed interface that should improve the evolvability of the PACS software.

Submodule 13.c.9 is another outlier of the Quantitative Analysis and could not be recognized within the CSA step. This submodule exhibits high changing rates. The part of the observations of the RA already outlined that logical coupling can be noticed in conjunction with Submodule 13.c.9. Four classes of this submodule are coupled in such a form that a quadrangle of dependencies arises. Each of these four classes is connected to two other classes of the quadrangle.

These four classes together with Class 13.c.9.A are the main factor for the internal links of Submodule 13.c.9. Class 13.c.9.A has logical coupling with Class 13.c.9.C, which is one of the classes composing the quadrangle of dependencies.

Nevertheless, the internal coupling is probably not the only reason for the high changing values of this submodule. The external coupling of Submodule 13.c.9 is even more frequent than the internal one. Class 13.c.9.C, Class 13.c.9.D, and Class 13.c.9.E are part of the quadrangular relations and also form the connection to other submodules. All three classes have logical coupling with Submodule 13.c.21. This submodule was already suspicious during the previous evaluation of Submodule 13.c.18. Submodule 13.c.21 has dependence with Submodule 13.c.9 as well as with Submodule 13.c.18.

Each class of Submodule 13.c.9 that shows external coupling has three strong links with classes of Submodule 13.c.21. The affected classes of Submodule 13.c.9 are part of the graphical user interface included in Submodule 13.c.9. Additionally, the classes also contain parts of the implementation of the configuration of Submodule 13.c.9. As it was already mentioned, Submodule 13.c.21 makes up a framework for user interaction within PACS. Considering all the available information we can conclude that the external relationship of Submodule 13.c.9 is the result of an architectural flaw. As the graphical user interface and, therefore, the related framework for user interaction is frequently changed, it should be separated from the rest of the application.

Module 29.a is the second largest and contains about 15% of all classes of the entire system. Two submodules could not be measured by CSA although they provided high changing values.

By means of RA, **Submodule 29.a.4** was already identified as a module with many internal couplings. Class 29.a.4.D plays a central role and has 6 strong connections with classes of its own submodule and 2 external relations with classes of Submodule 29.a.5. Another major point when considering logical coupling within Submodule 29.a.4 is built up by three classes: Class 29.a.4.E, Class 29.a.4.F, and Class 29.a.4.G. Furthermore, Class 29.a.4.F and Class 29.a.4.G have external connections to Submodule 29.a.5.

Most classes of Submodule 29.a.4 are involved in the implementation of a framework for the access of external *storage media*. All the classes of Submodule 29.a.4 that were outlined in the previous paragraphs are part of this storage framework.

Although all regarded classes build the connection with Submodule 29.a.5, they implement the access mechanism to different types of storage media. Submodule 29.a.5 implements a translation mechanism for different protocols. We drew the conclusion that the inheritance hierarchy of the classes that form the storage framework could bear structural shortages and should be rearranged.

The other classes of Submodule 29.a.4 that are involved in the strong coupling of this submodule are implementations of different data structures. One of these classes is Class 29.a.4.H which is outstanding, because it has internal dependence on Class 29.a.4.D and external coupling with Submodule 29.a.7. Thus, this class seems to play an important role as bridge between the storage framework and the data structures of Submodule 29.a.4. It builds a similar data structure as Class 29.a.4.I, although it does not show relationships with this class. Thus, the data structures are probably better designed than the storage mechanism, because they evolve independently from each other.

Submodule 29.a.5 provides a striking picture: Only two classes of Submodule 29.a.5 are involved in the interdependence of this submodule. These two classes are strongly coupled with each other and provide many external couplings too. Class 29.a.5.A and Class 29.a.5.B are connected to a very large quantity of classes. The internal coupling of Submodule 29.a.5 between these two classes is very strong, but there are many more external couplings. Another important fact about the connection structure of Submodule 29.a.5 is that the image is rather symmetrical. Each external class that is connected to Class 29.a.5.A has also a coupling with Class 29.a.5.B.

These results lead to the conclusion that the classes are *representations of two aspects of the same concept*. The hypothesis is confirmed by the fact that one of these two classes is a large collection of constants, whereas the other class provides the access mechanisms to these values. A possible solution to the evolutionary problem of Submodule 29.a.5 could be that the two classes should be inte-

grated into one single class. However, the size of the classes contradicts this suggestion, as each class contains many hundred lines of code. Thus, the entire concept of Submodule 29.a.5 should be reconsidered and be subject to reengineering.

The multiple links between Submodule 29.a.5 and Submodule 29.a.4 are a remarkable architectural characteristic of the studied software system. One of the considerations should be to integrate these two submodules. This suggestion is further confirmed by the other external couplings of Submodule 29.a.5. This submodule is not only coupled with Submodule 29.a.5, but also with Submodule 29.a.7 and Submodule 29.a.2. The first attention is drawn towards Submodule 29.a.7. It is like other external parts of the software system evolutionary related with both Class 29.a.5.A and Class 29.a.5.B. However, the previous part describes the relationship of Submodule 29.a.4 with Submodule 29.a.7. Thus, perhaps the integration of Submodule 29.a.4 and Submodule 29.a.5 could help to develop a good external interface to separate the evolution of the new integrated submodule and other parts of the software. Then each part could be changed without demanding the change of too many other system blocks, as the design-for-change principles suggest.

5.4 Lessons learned

In this section, we discuss certain properties of RA based on the experiences during the RA of the case study:

- *RA combines all levels of decomposition*: The Relation Analysis is based on information concerning changes applied to single classes. With these smallest building blocks it is possible to relate parts of the system on different decomposition levels with each other. As the sizes of the different parts on different levels have large deviations it seems promising to compare different levels of decomposition with each other, to receive a better insight into the structure of the software system.
- *RA reveals many couplings*: With the help of RA it was possible to find logical dependencies. Despite this high number of findings it seems that no false positives have been revealed. Many discoveries help in the architectural reasoning of a broad range of system blocks, which may contain structural weaknesses and draw the attention to those parts that should be developed carefully.
- *Different types of results*: The evaluation of the results gives rise to the assumption that many kinds of structural deficiencies may be discovered with the help of RA. Examples of such findings are spaghetti code, bad inheritance hierarchies, and poorly designed interfaces.
- *Most findings based on submodules*: In the case study most couplings that were discovered were located on the submodule level. Based on the strong deviations of size on different levels it is interesting to find submodules, which are related based on their historical development.
- *Frequent dependencies between system blocks*: RA revealed many internal and external couplings. Internal links are likely to point out limitations within classes. External couplings are even more interesting, because they may bring to light limitations of the architecture of the entire system.
- *Visualization simplifies navigation*: Due to the huge base of results as output of the RA method, additional visualization of the findings would improve the navigation to the system blocks of interest. This visualization could be supported by more sophisticated tools.
- *Some metrics were beneficial*: To spot outliers easily, metrics based on the attributes used within RA are helpful. For the evaluation the number of common check-ins was taken into account. In addition the average number of check-ins within the inspected part of the software is useful. Other such metrics could be integrated to enhance the Relation Analysis.
- *Domain knowledge*: For the evaluation of the findings resulting form RA the integration of domain knowledge into the analysis technique is essential. By applying RA certain problematic points in the architecture can be found. Then human knowledge has to be incorporated into the method to assess the necessity to revise the discovered part of the software system.

6 Conclusions and future work

In this paper, we described our Relation Analysis that allows a deep analysis of logical coupling of modules. Classes as smallest entities are compared based on dates and authors of changes. With this information, parts of the system that were changed together can be discovered. The approach was evaluated on 28 releases of an industrial Picture Archiving and Communication System (PACS) consisting of half a million lines of Java Code; for that, we consider our results as representative.

Release history data stored in CVS enables the detection of logical coupling of modules across the evolution of a software system effectively, thereby not analyzing any piece of source code. Design flaws such as god classes or spaghetti code could be discovered, although the source code was not analyzed at all. Additionally, the results of the case study pointed to architectural weaknesses such as poorly designed inheritance hierarchies and blurred interfaces of modules and submodules. Hence, this approach seems promising to be studied further.

Our QCR methodology revealed a large amount and many different types of architectural shortcomings. Nevertheless, the recall factor seems reasonable as the three analysis steps regard different aspects of the case study. The entire methodology provides a good overview of the system and allows detailed analysis of particular parts. The graphical representation contributes to understand certain characteristics of the software structure easily. QCR as a self-contained methodology allows reasoning about software architecture on a macro level, where the historical data of a system are taken into account. As a result QCR requires very little data to be kept, rather than dealing with many thousand lines of source code.

So far, we have used only simple tools in our work. In the future, we plan to add more sophisticated visualization capability to enable viewing the identified relationships with 3-dimensional graphs (e.g. with [17] or [22]).

7 Acknowledgments

We thank our industrial partner that provided all the information and helped us with the interpretation of the results. This work was in part supported by the EUREKA project CAFÉ ip00004 within the ITEA programme and the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT).

8 References

[1] Cederqvist P., "Version Management with CVS," Network Theory Ltd., December 2002.

[2] Cubranic D., Murphy G.C., "Hipikat: Recommending Pertinent Software Development Artifacts," ICSE 2003, Portland, IEEE CS Press, May 2003.

[3] Daskalantonakis M.K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola," IEEE Transactions on Software Engineering, Vol. 18, No. 11, pp. 998-1010, November 1992.

[4] Eick, S.G., Graves, T.L., Karr, A.F., Mockus, A., Schuster, P., "Visualizing software changes," IEEE Transactions on Software Engineering, 28(4):396-412, April 2002.

[5] Fenton N.E., Pfleeger S.L., Software Metrics—A Rigorous & Practical Approach, International Thomson Computer Press, Second Edition, 1996.

[6] Fischer M., Gall H., Pinzger M., "Populating a Release History Database from Version Control and Bug Tracking Systems," International Conference on Software Maintenance, IEEE CS Press, Aug. 2003 (to appear)

[7] Fogel K., "Open Source Development with CVS," Cariolis Open Press, November 1999.

[8] Gall H., Jazayeri M., Klösch R., and Trausmuth G., "Software evolution observations based on product release history," International Conference on Software Maintenance (ICSM '97), Bari, Italy, pp.160-166, October 1997.

[9] Gall H., Hajek K., and Jazayeri M., "Detection of logical coupling based on product release history." International Conference on Software Maintenance, IEEE CS Press, Nov. 1998.

10] Gall H., Jazayeri M., and Riva C., "Visualizing software release histories: the use of color and third dimension." International Conference on Software Maintenance (ICSM '99) (Oxford, England), pages 99-108. IEEE CS Press, Aug. 1999.

[11] Gefen D. and Schneberger S.L. "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications," International Conference on Software Maintenance, pp. 134-141, November 1996.

[12] Jazayeri M., "On architectural stability and evolution." Reliable Software Technologies - Ada-Europe 2002, Vienna, Austria, June 17-21, 2002.

[13] Khoshgoftaar T.M. and Halstead R., "Detection of Fault-Prone Software Modules During a Spiral Life-Cycle," International Conference on Software Maintenance, pp. 69-76, November 1996.

[14] Krajewski J., "QCR - A Methodology for Software Evolution Analysis," Master's Thesis, Technical University of Vienna, April 2003, available at http://www.infosys.tuwien.ac.at/Teaching/Finished/MastersTheses/

[15] Lehman M.M., "Programs, life cycles and laws of software evolution," Proceedings of the IEEE, pp. 1060-1076, September 1980.

[16] Lehman M.M. and Belady L. A., Program evolution, Academic Press, London and New York, 1985.

[17] Maletic J.I., Marcus A., and Feng L., "Source Viewer 3D (sv3D) – A Framework for Software Visualization," ICSE 2003, Portland, IEEE CS Press, May 2003.

[18] Ohlsson N. and Alberg H., "Predicting Fault-Prone Software Modules in Telephone Switches," IEEE Transactions on Software Engineering, Vol. 22, No. 12, pp. 886-894, December 1996.

[19] Pearse T. and Oman P., "Maintainability Measurements on Industrial Source Code Maintenance Activities," International Conference on Software Maintenance, pp. 295-313, October 1995.

[20] Turski W.M., "Reference Model for Smooth Growth of Software Systems," IEEE Transactions on Software Engineering, Vol. 22, No. 8, pp. 599-600, August 1996.

[21] Zimmermann T., Diehl S., Zeller A., "How History Justifies System Architecture (or not)," Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2003), Helsinki, IEEE, September 2003.

[22] Lanza M., "Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization," Ph.D. thesis, University of Berne, May 2003.