Improving Evolvability through Refactoring

Jacek Ratzinger, Michael Fischer
Vienna University of Technology
Institute of Information Systems
A-1040 Vienna, Austria
{ratzinger,fischer}@infosys.tuwien.ac.at

Harald Gall
University of Zurich
Department of Informatics
CH-8057 Zurich, Switzerland
gall@ifi.unizh.ch

ABSTRACT

Refactoring is one means of improving the structure of existing software. Locations for the application of refactoring are often based on subjective perceptions such as "bad smells", which are vague suspicions of design shortcomings. We exploit historical data extracted from repositories such as CVS and focus on change couplings: if some software parts change at the same time very often over several releases, this data can be used to point to candidates for refactoring. We adopt the concept of bad smells and provide additional *change smells*. Such a smell is hardly visible in the code, but easy to spot when viewing the change history. Our approach enables the detection of such smells allowing an engineer to apply refactoring on these parts of the source code to improve the evolvability of the software. For that, we analyzed the history of a large industrial system for a period of 15 months, proposed spots for refactorings based on change couplings, and performed them with the developers. After observing the system for another 15 months we finally analyzed the effectiveness of our approach. Our results support our hypothesis that the combination of change dependency analysis and refactoring is applicable and effective.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance and Enhancement—*restructuring, reengineering*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, evolution measures*

Keywords

software evolution, refactoring, change smells

1. INTRODUCTION

The notion of "bad smells" was introduced by Fowler [4] and describes a vague suspicion that the software contains design deficiencies that should be restructured. Our research question is: Are there data sources other than source code to identify such kinds of smells for refactorings? We address this question by exploiting change history data of software and analyze them to identify smells and, as a consequence, hot-spots for refactoring. We utilze our visualization approach of change couplings to help software engineers to locate places that deserve design improvements.

Furthermore, we then apply appropriate refactorings to the identified software parts and again observe the evolution for a period of release. Then at some point we again investigate the change history to see whether the initially suggested and implemented refactorings were effective with respect to change couplings. We can positively answer the question of effectiveness, if the refactored software keeps that status over the observed post-refactoring releases. As a result, we derive that, given the refactored structure does not again show high change couplings, these hot-spots were the right places to apply refactorings.

To evaluate our approach, we used a 500 000 lines of code (LOC) industrial Picture Archiving and Communication System (PACS) written in Java and observed it twice for a period of 15 months, with a change coupling driven refactoring between the two observation periods. The results show that change couplings point to highly relevant refactoring candidates in the code and that after refactoring the code has a low change coupling characteristics, which means that the refactorings were successful.

The origins of this work are our previous results described in [5], in which we concentrated on the measuring of software dependencies: Common change behavior of modules to be discovered on a macro level exploiting information such as version numbers and change reports. Source code control systems such as CVS provide necessary information about change requests and usually also about the change implementation, as the developer can use such systems for documentation purpose [3]. Thus, hidden dependencies not evident in the source code can be revealed. Such common change behavior of different parts of the system during the evolution is referred to as *logical or change coupling*. As a result, change couplings often point to structural weaknesses that should be subject to reengineering.

We propose to use refactoring based on change smells detected with the help of mining source code repositories. We extend the concept of "bad smells" introduced by Fowler to change smells, as some structural weaknesses are not evident in the code but in the software history. When developers

have to change some system part they often work on several files containing source code. Sometimes the dependencies are not easily detectable within the source code, e.g. when similar patterns or clones of source code are used but for different functionality. Nevertheless, in such a situation the engineer has to consider all the involved files to keep the consistency of the entire system. After detecting change smells and its cause, we suggest certain refactorings to improve the software.

The remainder of the paper is organized as follows: In Section 2 we present two of our changes smells that are relevant for this paper. Next, in Section 3 we describe our industrial case study. In Section 4, we describe the core contribution of this paper, the change smell guided refactoring using examples from the case study. Section 5 positions our work in relation to other works, and in Section 6 we draw conclusions.

2. CHANGE SMELLS

Software often encloses change smells. These are spots in the system, which do not evolve smoothly but cause changes through a long period in the development process. To improve the development effort, we need decision support where to apply restructuring. Fortunately, most development teams collect historical data about the product's life cycle as they use software configuration management systems such as the Concurrent Versions System (CVS).

Refactoring is a vital technique to improve the design of existing systems by changing a software system in such a way that the external behavior of the code is not changed yet the internal structure is improved. It is an activity that accompanies all phases of the software life cycle. Many different refactorings have already been identified [4]. When applying refactorings on detected change smells we can demonstrate how the evolvability of software improves.

As CVS logs every action, it provides the necessary information about the history of a system. The log-information is pure textual, human readable information and retrieved via standard command line tools, parsed and stored in a relational database. Following the import of the logs, the required evolutionary information is reconstructed in a post processing phase. Log groups L_n are sets of files which were checked-in into the CVS by a single author within a short time-frame—typically a few minutes. The degree of logical coupling between two entities a, b can be determined easily by counting all log groups which both a and b are members of, i.e., $C = \{\langle a, b\rangle | a, b \in L_n\}$ is the set of logical coupling and $|C|$ is the degree of coupling.

We define *change coupling* (or logical coupling) as follows: *Two entities (e.g. files) are logically coupled if modifications affect both entities over a significant number of releases.* An interesting aspect of coupling is the distinction between internal and external. 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 couplings between classes within this module and any other part of the software (i.e. another module or another subsystem) are considered as external couplings.

In addition to Fowler's "bad smells", we investigate two change coupling smells in this paper:

Man-in-the-Middle: A central class evolves together with many others that are scattered over many modules of the system. Thus, we detect change couplings between the central class and the related ones; these related classes often exhibit change couplings among each other as well. A *Man-in-the-Middle* smell hinders the evolution of single modules, because of the strong dependencies to other parts of the system. The central class does not necessarily contain much code. We detected that it is just a class, which represents a kind of a mediator for many other classes or even other modules and has to be changed often if other parts of the software change. Refactorings such as *Move Method* and *Move Field* can repair such a smell. Then the functionality can be pulled to the data and slim interfaces may be introduced.

Data Container: This smell is similar to the data containers defined in the "Move Behavior Close to Data" reengineering pattern of Demeyer et al. [1] that defines data containers as "classes defining mostly public accessor methods and few behavior methods (e.g., the number of methods is approximately two times larger than the number of attributes)". The difference is that in our change smell Data Container *two classes* make up the smell instead of a single one. One class holds all the necessary data whereas the second class interacts with other classes implementing functionality related to the data of the first class. This violates the principle of encapsulating data and their related functions. In our case, when two classes have common change patterns we should check for the reason. We detected situations where the change smell of *Data Container* was responsible for the unintended evolution of the software. This smell is detectable within the visualization when we encounter two classes, which have a strong change relation conneting them and additionally a net of other classes surround these two classes with weaker coupling. Usually, both the data container and the class with the interaction methods are related with each of the other classes. Hence, we obtain a lot of triangular relationships. The refactorings *Move Method* and *Extract Method* should be used to enrich the *Data Container* with behavior operating on the data and to combine the two classes into one. The aim of the improvement is that the data is well encapsulated.

3. CASE STUDY

A Picture Archiving and Communication System (PACS) was selected as case study for our approach. The PACS includes a viewing workstation, which supports concurrent displaying of pictures as well as an archive. The images are acquired from different modalities like magnetic resonance, or ultrasound scanning and save in distributed archive storages. The software is implemented in Java. The information of the whole application is maintained with the help of CVS.

All subsystems of the PACS can be viewed as separate projects that encapsulate some aspects of the whole application such as viewing unit, archiving process or extensions to the viewing unit. These extensions add diagnostic features to the viewing application. The case study is composed of 35 subsystems, each containing between one and fourteen modules.

Single classes represent the lowest level of decomposition. The history of the PACS system was inspected over a period 30 months. During this time the software grew from approximately 2000 to more than 5500 classes. At the end it was composed of over 500 000 LOC. Regarding these simple numbers the system seems quite well designed, as each class has less than 100 lines of code on the average.

4. CHANGE SMELL BASED REFACTORING

In this section we present an example from the case study, where we detect change smells and use refactorings to improve the evolvability of the software. During the analysis of the historical data received from CVS we identify a small module (i.e. Java package) with a high changing activity. So we calculate the logical couplings of this module called *jvision/workers*. To get a better understanding of logical couplings for the classes of *jvision/workers* we create a graphical representation (see Figure 1).

In this representation classes are depicted as small ellipses. The ellipses are grouped by their membership to modules. Modules themselves are depicted as bounding ellipses surrounding their classes. This structural information is enriched with historical data. From CVS we extract the evolution of classes and calculate logical coupling between classes. This coupling is depicted in Figure 1 through edges connecting the ellipses whereas the thickness of the edges describes the "strength" of the visualized couplings. The more often a pair of classes is changed at the same time the thicker is the representing edge. This visualization approach has been extended with class based metrics and implemented in EvoLens.

The navigation through the change couplings based on our visualization approach helps to locate the change smell *Man-in-the-middle* for the class *ImageFetcher*. This class has multiple strong logical couplings with other classes. The situation is even worse as it often changes together with classes of different packages. Thus, when a change has to be done by an engineer the editing is scattered over the software.

ImageFetcher is one of the largest classes of the entire system; it contains almost 2000 lines of code. The methods of this class are of exceptional length: Some of them contain more than 100 lines of code. When trying to reveal the reasons for such "spaghetti code", we discover that many methods are similar. Thus, the entire class is internally redundant. The length of the class itself does not automatically lead to the necessity of refactoring, but *ImageFetcher* often changes together with other classes. Thus, each change has to be thoroughly analysed in order not to miss any important change, which may be scattered over a large part of the system. This has a severe impact on the maintenance effort: When a bug is discovered within one of the methods of this class, many other methods have to be changed in a similar way. Often such changes are missed and have to be fixed later when the bug reoccurs. This results in a high changing activity.

Additionally, this class seems to have divergent changes [4], because it changes together with a lot of classes of other

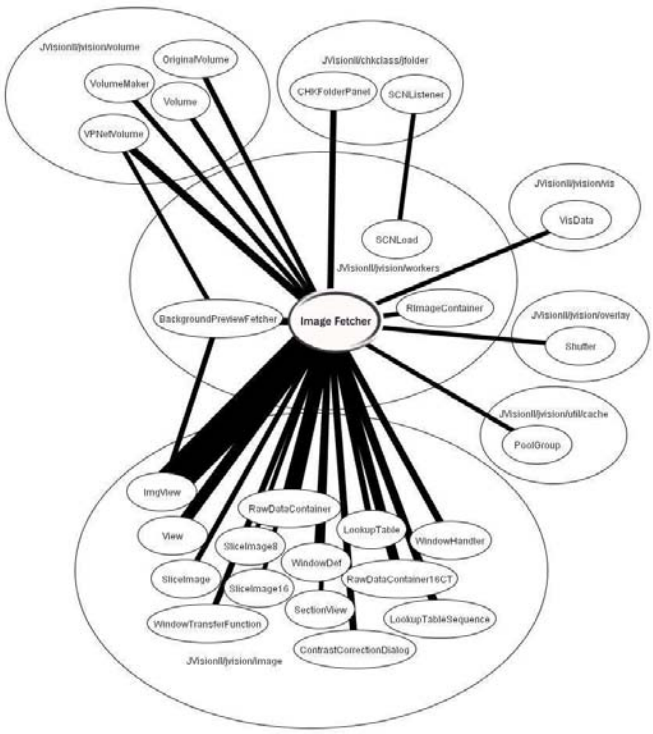

Figure 1: Change smell: Man-in-the-Middle

modules. Thus some methods seem stronger related with classes of a particular module, whereas other methods of *ImageFetcher* have to be changed in conjunction with classes of other modules. When inspecting the source of the *ImageFetcher* we determine that the principle of separation of concerns is violated. This class implements a thread pool, a queue for work items, and logic for loading images altogether. As a result, different classes implementing different functionality are related with *ImageFetcher*.

4.1 Refactoring to improve Evolvability

We apply several refactorings to reduce the weaknesses of this change smell. Then we continue to observe the evolution of the module *jvision/workers* to see if the evolvability has been improved through evolution guided refactoring.

To minimize code duplication, we first extract code clone parts of methods and reuse the newly formed methods where appropriate. For that, we apply the *Extract Method* refactoring that helps to get reusable items. After these improvements the class contains just 1100 lines of code, because of the removal of duplication.

To further improve the evolvability, we split *ImageFetcher* into new classes encapsulating the different concerns. We move the methods and data for image loading into a separate class called *FetchWorker*. The logics for thread pooling and the handling of the work queue are left together in *ImageFetcher*. After the movements we obtain a surprising result: *FetchWorker* contains just one public method called *loadimage()*. This simple interface results in reduced coupling. Also *ImageFetcher* has a simple interface after the

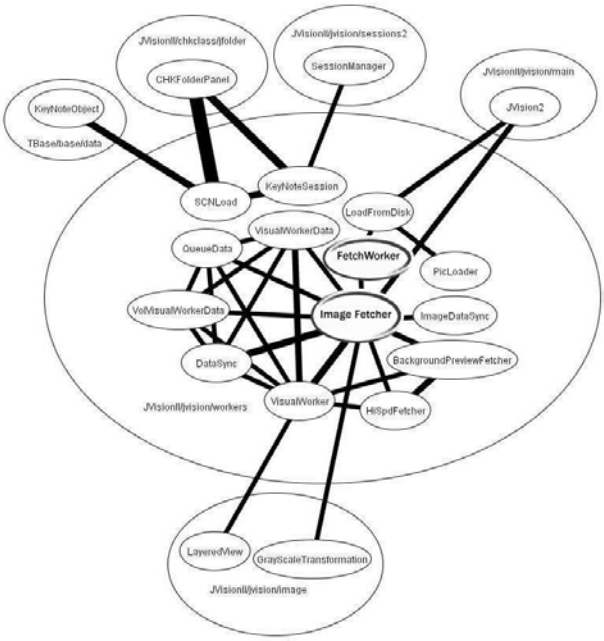

Figure 2: Evolution after refactoring of change smell

refactorings. It provides methods for starting and stopping the thread pool and for adding orders to load certain images into the work queue.

4.2 Evolution after Refactoring Change Smells

After our refactorings, we observe the software again for 15 months, which is exactly the period we analyzed the system before the refactoring. We inspecte the system for such a long period to gain more accurate assessments. Fig. 2 describes the change couplings of module *jvision/workers* after the refactorings, in contrast to Fig. 1 that presents the logical couplings, which are used as trigger for the refactorings. In Fig. 1 we observe the system from January 2002 until March 2003. At the end of March 2003 we refactor *ImageFetcher* and Fig. 2 represents the development from April 2003 until June 2004.

During the second 15 months the development of the module *jvision/workers* continued on a high level. A lot of functionality was added and improved. As a result new classes such as VisualWorker, VisualWorkerData, VolVisualWorkerData, and HiSpdFetcher were added during this time. However, Figuer 2 exhibits no strong logical coupling for the classes of module *jvision/workers*. Thus, several classes are changed during the second observation period, but not even two classes have been changed more than six times together. The refactored classes *ImageFetcher* and *FetchWorker* have fewer then 4 common changes with other classes. The external coupling to other classes can be reduced significantly. When asking developers for the reason of this evolution, they stated that the interfaces of the new classes were now much clearer and the classes could be developed more individually. As a result of the refactoring based on change couplings we can improve the structure of evolutionary hot spots and the evolvability of the software system.

Fig. 2 contains a web of logical couplings within module *jvision/workers*. Especially, *ImageFetcher* is connected with many other classes. One of these classes is the newly refactored class *FetchWorker*. These two classes have been changed together twice. Thus, the absolute level is low. What about the entire web of connected classes? Many of the involved classes provide different load strategies to *ImageFetcher*, but they are not organized in a well-designed inheritance hierarchy. Therefore, we need a second refactoring step to build up an inheritance hierarchy for different image loading approaches. Again, this situation can be detected with the help of change couplings.

4.3 Tool Support

We provide some techniques for the improvement of software structures to facilitate further improvements and bug fixes during the software evolution. Therefore, we utilize the history data to find change smells that should be refactored. A visualization of the large amount of historical data of a system, which can be extracted from software repositories such as CVS, enables to spot change smells more easily.

Hence, we implemented a tool for the visualization of evolution data such as change couplings [11]. This tool, called EvoLens, parses log files of CVS and calculates change couplings between classes based on their common change behavior. The couplings are then visualized together with structural information. Our EvoLens tool provides the capability to navigate easily through structure and time. For every selectable software part and every period in the system's history, EvoLens can show the internal and external couplings of the system. Additionally, growth metrics of classes are also visualized with EvoLens to help assessing the necessity of reengineering.

5. RELATED WORK

Software metrics, provide a key to measure and improve the quality of software [2]. Long-term empirical studies show potential in identifying phases in the life cycle of software where different activities need to be taken to stabilize the process. By focusing on the types of changes, costs and efforts to evolve, Kemerer and Slaughter [8] suggest that future trends within a particular system are predictable. However, the authors concentrated on the historical development of the software without the stronger relation with the internal structure of the system. Hence, other metrics like cohesion and coupling could be incorporated to round up the approach. Stevens et al. [13], who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another." Coupling measures are often based on source code. Nevertheless, different modules of a system may be strongly related to each other although this relationship is not easily detectable in the source code. In such a situation historical data can help to get better results (for example [15]).

Simon et al. [12] postulate that refactoring should be regarded as a key issue to increase internal software quality. Their approach demonstrates how to use cohesion metrics based on generic distance measures to identify, which software elements are closely related and should therefore be assembled in object-oriented entities. Source code inspec-

tions are another field where code smells have to be evalutated. jCOSMO [14] was developed to automatically detect code smells such as the ones defined by Fowler [4] and to visualize their distribution over the system.

Complementing to our approach in which we correct hot spots in the evolution of software systems, also the risk of a change to break an already existing feature can be assessed by analyzing software changes [9]. Ostrand et al. [10] predict the quality of certain parts of software. They estimate the number of faults per file for the next release based on a negative binomial regression model using information from previous releases. Additionally to source code repositories several other information sources such as mail messages and defect reports can by explored to get a better understanding how a software product has evolved since its conception [6]. In [7] four different kinds of studies for software evolution are presented and compared. The studies consider long-running observations of growth and evolution as well as fine grained issues like code cloning and software architectures.

6. CONCLUSIONS

We have shown an approach to exploit historical data extracted from repositories such as CVS in terms of change couplings: We adopted the concept of "bad smells" and provided additional *change smells* based on change coupling analysis. Such a smell is hardly visible in the code, but easy to spot when viewing the change history.

Based on these change couplings and the proposed change smells, the developer obtains support where to apply refactorings efficiently. In an industrial case study comprised of 500 000 LOC in Java, we have shown how these change smells can be cured and how refactoring can be based on them. It turned out that after the refactorings had been implemented, the evolution of the system, that we observed for another 15 months, was facilitated and did not lead to the originally strong change couplings or change smells. In talking to the developers, they stated that the directed refactorings were effective for them and the new interfaces and classes were much clearer and easier to use.

From this we conclude, that such an approach can help in improving the maintainability and evolvability of a large software system. The change coupling data itself to get is rather straightforward, as are the two described change smells *Man-in-the-Middle* and *Data Container*.

Our prototype tool EvoLens integrates many of these concepts already but it will be enhanced to better deal with change smells in the future. The next steps will further investigate change smells and their curing with appropriate refactorings.

7. ACKNOWLEDGMENTS

We thank Tiani Medgraph, our industrial partner, which provided the case study and helped us with the interpretation of the results. The work described in this paper was supported by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission in terms of the EUREKA 2023/ITEA project FAMILIES (`http://www.infosys.tuwien.ac.at/Cafe/`).

8. REFERENCES

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented Reengineering Patterns*. Morgan Kaufmann Publishers, An Imprint of Elsevier Scienence: San Francisco CA, USA, July 2002.

[2] S. Demeyer and T. Mens. Evolution metrics. *Proc. of the 4th Int. Workshop on Principles of Software Evolution*, pages 83–86, 2001. Session 4A: Principles.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. *Proc. Int. Conf. on Software Maintenance*, pages 23–32, September 2003.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, June 1999.

[5] H. Gall, J. Krajewski, and M. Jazayeri. CVS Release History Data for Detecting Logical Couplings. In *Proc. 6th Int. Workshop on Principles of Software Evolution*, pages 13–23. IEEE Computer Society Press, September 2003.

[6] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softchange. *Proc. 16th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 336–341, June 2004.

[7] M. Godfrey, X. Dong, C. Kapser, and L. Zou. Four interesting ways in which history can teach us about software. *Int. Workshop on Mining Software Repositories*, May 2004.

[8] C. F. Kemerer and S. A. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July-August 1999.

[9] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April-June 2000.

[10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. *Proc. on the Int. Symposium on Software Testing and Analysis*, pages 86–96, July 2004.

[11] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. *Technical Report: Vienna University of Technology*, December 2004.

[12] F. Simon, F. Steinbrückner, and C. Lewernetz. Metrics based refactoring. *Proc. European Conf. on Software Maintenance and Reengineering*, pages 30–38, March 2001.

[13] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, May 1974.

[14] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. *Proc. of the 9th Working Conf. on Reverse Engineering*, pages 97–108, October 2002.

[15] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *Proc. Int. Conf. on Software Engineering*, pages 563–572, May 2004.