Experience-Based Refactoring for Goal-Oriented Software Quality Improvement

Jörg Rech, Eric Ras

Fraunhofer Institute for Experimental Software Engineering
67661 Kaiserslautern, Germany
(rech, ras)@iese.fraunhofer.de

Abstract. In agile software development refactoring is an important phase for the continuous improvement of software quality. Unfortunately, the application of refactorings is very subjective and heavily based on the expertise of the developers resulting in an unstable quality assurance. In this paper, we present an experience-based approach for the semi-automatic and goal-oriented refactoring of software systems based on didactical augmented experiences, following the experience factory paradigm. This approach promises the accelerated acquisition, (re-) use, and learning of knowledge in the refactoring process.

1 Introduction

Today, agile methods take in an increasing part of development methodologies in industry. Agile methods like *Extreme Programming (XP)* consist of lightweight and people-centric processes rather than traditional plan-based software development methods. Some of the main principles of agile software development are to deliver high-quality working software frequently to the customer, to welcome changing requirements – even late in development, and to provide the support and environment to motivate developers [4].

An important process in agile development to improve the structure and associated quality of software systems between development cycles or in maintenance activities is called *Refactoring*. Especially in agile development with high time-pressure, under-engineering – and consequently unmaintainable software – usually occurs when the focus lies on adding more functionality rather than improving the design. But, as systems are getting larger, refactoring gets more and more complex and time consuming to do it manually. Even if one knows where to refactor the software it is often not clear how and under which conditions what refactoring activities should be performed [11].

In order to react fast on changing customer requirements or technological environments the developers must either already have the competence to solve the current tasks or be enabled to quickly learn required skills. Time-pressure and costs prevents developers to participate in class trainings. The same is true if developers are

asked to share their knowledge and experiences in long and complex processes. Capturing their expertise should be as non-intrusive and short as possible.

We propose an approach that supports refactoring activities by reusing knowledge and didactical augmented experiences from experts. By following the *Experience Factory* paradigm, software quality relevant experiences made during the execution of refactoring activities are captured from the developers, stored in an experience repository, and augmented before reuse in new refactoring activities. The underlying thesis is that continuous competence development for agile methods is primarily enhanced by sharing knowledge amongst colleagues, using lightweight knowledge-based systems combined with well-founded experiential learning principles. The key issue of experiential learning is to extrapolate the experience from the learning setting to real world situations.

2 Current Practice and Related Work

To improve the quality of systems developed with agile methods we propose a novel approach related to the fields of Knowledge Management, experiential learning, and Refactoring. The key issue of experiential learning is to extrapolate the experience from the learning setting to real world situations. In the following sections we elaborate on current practices and related work in these fields. A synthesis based on these fields and a description of our approach will be given in the next section.

2.1 Current Practice in Refactoring

Refactoring is a manual process to remove or weaken quality defects and therefore to improve the quality of software systems. Due to the fact that activities in software product maintenance account for the majority of the cost in the software life-cycle [5] refactoring is an effective approach to prolong the software lifetime and to improve its maintainability. Especially in agile software development, methods as well as tools to support refactoring become more and more important [15].

In this paper we use the umbrella term *quality defects* for any defect in software systems that has an effect on software quality (e.g., as defined in ISO 9126) but not directly effect functionality. Examples for quality defects are code smells, design flaws, or anti-patterns.

Current research in the field of software refactoring is very active and has identified the use of metrics for refactoring as an important research issue [15]. Previous research in the sub-field refactoring has resulted in behavior-preserving approaches to refactor object-oriented software systems in general [18], tool support to automatically refactor or restructure applications [13, 21], or methods for pattern based refactoring [8].

Today, various methods for the discovery of quality defects are known. For example, inspections are used to discover defects in early development phases, code analysis is used to quantify code characteristics, testing is used to detect functional defects after implementation, and coding standards are used to ensure quality attributes and prevent quality defects. Research by Simon on code smell discovery

resulted in metrics-based and visually supported quality assurance with a similarity measure [23] to reclassify classes and minimize inter-class coupling. Zimmermann and colleagues pursue the goal to support developers in maintenance activities by using a technique similar to case-based reasoning. In order to support software maintainers with related experiences in form of association rules about changes in software systems they mined association rules from a versioning system by collecting transactions including a specific change [25]. Demeyer et al. proposed a framework to predict refactoring activities with time series analysis based on several metrics [9]. Grant and Cordy proposed another approach to automatic code smell detection by supporting a software developer in the rule-based discovery of code smells as well as the belonging refactorings [12]. Finally, van Emden and Moonen analyzed code for two code smells and visualized them for manual discovery [17].

Tahvildari examined the effect of refactorings on the two qualities maintainability and performance by using several static and dynamic metrics [24]. Additionally, the recently started project QBench pursues a similar goal by using metrics in order to discover quality defects and automatically applying refactorings [1]. The amount of overlap in the area of defect discovery between their work and ours as described in this paper and [20] remains to be seen.

All of the mentioned research approaches support specific kinds of automatic quality defect discovery with different techniques. None of these approaches are comprehensive, what metrics can be used to discover quality defects, or describe how refactoring experiences could be reused. Neither are the effects of refactoring activities on all software qualities or code examined. Hence, in our approach we develop models to detect quality defects with metrics or rules and try to support defect correction activities with tailored experiences.

2.2 Current Practice in Reusing and Learning from Experts' Knowledge

Beck defined four essential values that should be reflected by XP practices to be successful: communication, simplicity, feedback and courage [3]. *Communication* is one value that acts as an essential part of agile development practices like unit testing, pair programming, and task estimation. So far, communication among developers and customers in XP is typically done verbally and on different expertise levels, i.e., between novices, practitioners, and experts. The loss of knowledge when key developers leave the organization and the increasing flood of new arising information lead to the development of Knowledge Management Systems (KMS). It was assumed that KMS could solve the problem of exchanging knowledge between individuals by providing intelligent retrieval mechanisms and innovative presentations techniques. Lindvall and Rus stated that *learning* is considered to be a fundamental part of KM because employees must internalize (learn) shared knowledge before they can use it to perform specific tasks [22]. But most KMS's focus mainly on information and knowledge, e.g., the product of process models and less on learning processes itself and the needs of individuals.

Hence, the aim of effective knowledge transfer is to guide novices or practitioners through knowledge stages [10] to become experts by focusing on appropriate learning processes that foster reflection and systematic thinking about the knowledge to be

transferred. This means that knowledge transfer should not focus only on knowledge types, context or media presentation but also on learning processes that enable learners to develop cognitive skills and competencies, which can be applied during daily work. Recently, Ras and Weibelzahl presented a preliminary framework which could solve the problems when novices reuse experts' explicit knowledge by initiating appropriate learning processes by means of micro-didactical arrangements [19].

3 Experience Based Refactoring

The primary goal of agile methods is the rapid development of software systems that are continuously adapted to customer requirements without large process overhead. Refactoring is typically done between development cycles to improve the developed system.

The basic process of our approach encompasses these methods as depicted in Figure 1. Knowledge about quality defects from defect discovery processes is used to retrieve experiences associated to similar smells from previous refactorings. The process to discover and (re-) use refactoring experiences is described in section 3.1. These experiences are used to handle the quality defect in the defect removal phase. Additionally, these experiences are augmented by so called micro-didactical arrangements (MDA) that initiate learning processes and hence improve the experiences' understandability, applicability and their adaptability to the reuse context. MDA's are further described in section 3.2. An example of our approach incl. details on quality defects, refactorings, and refactoring experiences is given in section 3.3.

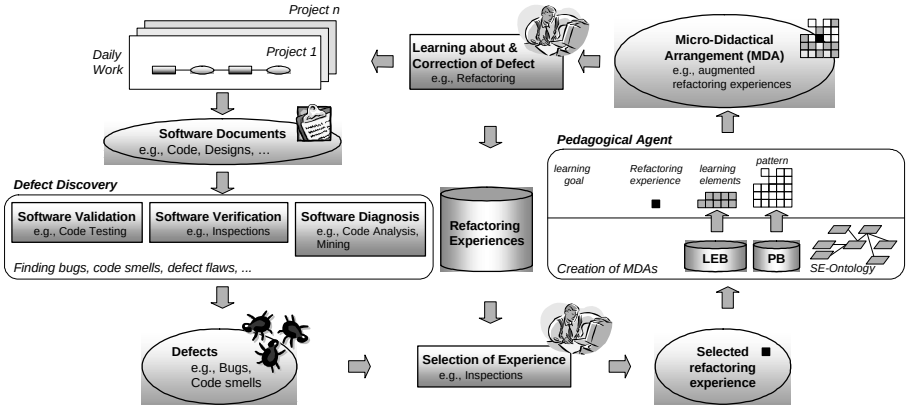

Fig. 1. Experience-based semi-automatic reuse of refactoring experiences

3.1 Defect Discovery and Correction Planning

As shown in Figure 2, we define six phases, based on QIP [2], for the continuous automatic discovery of quality defects. First, we start with the definition of qualities that should be monitored and improved. For example, this may result in different goals (i.e., quality aspects) as, reusability demands more flexibility or "openness" while maintainability requires more simplicity. Phase two represents the application area for Knowledge Discovery in Databases (KDD). It is concerned with the measurement and preprocessing of the source code to build a basis for quality defect discovery. Results from the discovery process (i.e., quality defects) can than be represented (e.g., visualized) and priorized to plan the refactoring in phase three. Here, the responsible person has to decide which refactorings have to be executed in what configuration and sequence in order to minimize work (e.g., change conflicts) and maximize the effect on quality. In phase four the refactoring itself is applied to the software system by the developer that results in an improved product. Phase five compares the improved with the original product to detect changes and their impact on the remaining system. Finally, in the sixth phase we report the experiences and data about tasks, changes, and effects to learn from our work and continuously improve the model of relationship between quality, refactorings, and quality defects.

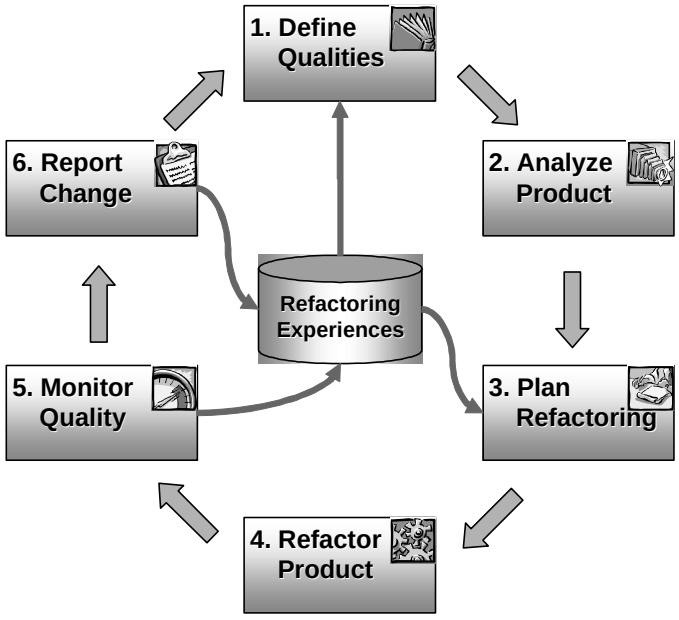

Fig. 2. Quality-driven metrics-based refactoring

As indicated previously, the KDD sub-processes are grouped in phase two. We select source code from a specific build, preprocess the code and store the results in the software repository, analyze the data to discover quality defects, discover

deviations from average behavior, cluster code blocks with severe or multiple quality defects, and represent discovered and priority sorted quality defects to the user.

3.2 Augmenting Refactoring Experiences

Quality defects that are detected during defect discovery are used to search for related refactoring experiences. After the user selects an experience that he wants to apply, the selected experience is forwarded to the pedagogical agent that enriches the experience to an MDA [19].

MDA are generated based on patterns. A pattern consists of a set of learning activities that are covered by the MDA. A learning activity is defined as a tuple of a learning process and the learning content to be used. All learning activities are documented formally in an instructional design model. Each MDA contains learning activities that fit to several adult learning styles. Four learning styles are supported: reflectors, theorists, pragmatists, activists as described in Kolb's *learning style inventory* [14]. The learning processes are related to Kolb's *Experiential Learning Circle* (i.e., making concrete experience, observation and reflection, formation of abstract concepts and test in new situations) and to Merrill's first principles of instruction (i.e., solving real world problems, activating existing knowledge as a foundation for new knowledge, demonstration of new knowledge, applying new knowledge by the learner, integrating new knowledge into the learner's world) [16].

The selection of learning activities is based the educational goal of the learner and the type of experience to be reused (i.e., product, process, tool or lesson learned). *Educational goals* differ widely in dependence of the target audience and the knowledge of the learners. In our approach, we refer to Bloom's taxonomy of educational goals [6], which is widely accepted and applied in various topic areas including Software Engineering [7]. A preliminary framework of the pedagogical agent in [19]describes the generation process and the used technologies in more detail. Semantic relations between learning elements and domain ontology are used by the pedagogical agent to generate MDA's. MDA's support self-directed learning, i.e., they allow the learner to choose his own learning goal and the learning path to reach it. An MDA can be understood as a network of learning elements in an information space, where several guided tours are proposed and specific learning elements are used as landmarks for orientation.

3.3 Refactoring Example

In this section we give a simple example to better understand the process of our approach (as depicted in Figure 1).

> *Imagine an object-oriented software system with a method that has a length of 300 LOC. This code smell, as described in [11], is called "Long Method". A long method is a problem esp. in maintenance phases as the responsible maintainer will have a hard time to understand the function of this method.*

One suitable refactoring for the mentioned code smell might be the refactoring simply called "Extract Method": the long method is reviewed to detect blocks that can be put into new (sub-)methods. The experience "The Extract Method is used to remove code smells of the type Long Method" is enriched with LEs after the novice user selects the learning goal "application" [6]. To fulfill this learning goal, the developer uses a created MDA by which he first acquires knowledge about the topics: method, class and LOC; a part of the refactoring domain ontology will be presented for orientation purposes, a definition and description about the Extract Method will be provided. The intermediate learning goal "comprehension" is reached by providing examples of code smells, a summary of the previous learned theory and a general scenario how such a code smell is removed. The "application" level is reached by acquiring procedural knowledge, i.e., knowing how to remove the Long Method code smell by providing descriptions about processes, principles, laws, rules and strategies. The content is presented in a way that certain learning processes are initiated e.g., asking open questions to activate previous knowledge or to reflect about demonstrated knowledge; showing cause/effects of long methods to improve understanding; providing examples of long methods so that the learner can derive abstractions by systematic thinking; apply the new knowledge by solving examples given by the MDA; integrate the gained knowledge into practice by presenting guidelines and hints to solve the current problem of the project.

The generation of new methods might create another smell called "Large Class" (i.e., the presence of to many methods in a class) which might complicate the case even more.

For example, the developer might remark that every block of code that has a common meaning, and could be respectively commented, could also be extracted into several small methods. Furthermore, he might note that the extraction of (sub-)methods, from methods implementing complex algorithms, can effect performance requirements of the software system and therefore might not be applicable.

Finally, the new experiences are annotated by the developer and stored in the refactoring experience base.

During the refactoring based on the information from the MDA the developer makes two basic types of new experiences. He learns whether or not to apply refactorings for a given code smell in a specific environment. Additionally, during the removal of a quality defect he makes changes to the system and discovers their impact on the rest of the system. While this example only touches a simple quality defect and refactoring, more complex refactorings influence inheritance relations or introduce design patterns [11].

4 Summary

The proposed framework promises the systematic and semi-automatic support of refactoring activities for developers in agile development. Experiences made during the execution of refactoring activities are captured from the developers, stored in an experience repository, and didactical augmented before their reuse in new refactoring activities.

This incremental and low invasive (i.e., cheap) integration of knowledge management and e-learning technology supports developers during refactoring as well as managers for the monitoring and planning of larger refactoring activities.

Acknowledgements

We gratefully thank our students for their work as well as our colleagues Andreas Jedlischka and Torsten Willrich for their valuable feedback. The work is sponsored by the BMBF (German ministry of education and research) in context of the projects RISE (01ISC13D) and indiGo (01AK951A).

References

1. Andriessens C., Mohaupt T., Seng O., Simon F., Trifu A., and Winter M., "QBench Projektergebnis: Stand der Technik," FZI Forschungszentrum Informatik, Project Report, 2004.
2. Basili V. R., Caldiera G., and Rombach D., "The Goal Question Metric Approach," in Encyclopedia of Software Engineering, 1st Edition ed. New York: John Wiley & Son, 1994, pp. 528-532.
3. Beck K., "eXtreme Programming eXplained: Embrace Change." Reading: Addison-Wesley, 1999.
4. Beck K., Beedle M., van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R. C., Mellor S., Schwaber K., Sutherland J., and Thomas D., "Manifesto for Agile Software Development," 2001.
5. Bennett K. H. and Rajlich V. T., "Software Maintenance and Evolution: A Roadmap," presented at Future of Software Engineering Track of 22nd ICSE, Limerick, Ireland, 2000.
6. Bloom B. S., Engelhart M. D., Furst E. J., Hill W. H., and Krathwohl D. R., "Taxonomy of educational objectives: The classification of educational goals," in Handbook I cognitive domain. New York: Longmans, Green and Company, 1956.
7. Bourque P., Dupuis R., Abran A., Moore J. W., and Tripp L., "The guide to the Software Engineering Body of Knowledge," IEEE Software, vol. 16, 35-44, 1999.
8. Cinneide M. O., Automated Application of Design Patterns: A Refactoring Approach. Ph.D. Thesis. Department of Computer Science, Trinity College, Dublin, 2000.
9. Demeyer S., Ducasse S., and Nierstrasz O., "Finding refactorings via change metrics," ACM. In: SIGPLAN Not. (USA), vol. 35, 166- 77, 2000.
10. Fitts P. M. and Posner M. I., Human Performance. Belmont, CA: Brooks-Cole, 1967.

11. Fowler M., Refactoring: Improving the Design of Existing Code, 1st ed: Addison-Wesley, ISBN 0-201-48567-2, 1999.
12. Grant S. and Cordy J. R., "Automated Code Smell Detection and Refactoring by Source Transformation," presented at International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE), Victoria, Canada, 2003.
13. Griswold W. G., Program Restructuring as an Aid to Software Maintenance. PhD Thesis. Department of Computer Science and Engineering, University of Washington, Washington, 1991.
14. Kolb D. A., Experiential Learning: Experiences as the source of learning and development. New Jersey: Prentice Hall, 1984.
15. Mens T., Demeyer S., Du Bois B., Stenten H., and Van Gorp P., "Refactoring: Current Research and Future Trends," Electronic Notes in Theoretical Computer Science, vol. 82, 17, 2003.
16. Merrill M. D., "First principles of instruction," presented at International conference of the Association for Educational Communications and Technology (AECT), Denver, USA, 2000.
17. Moonen L., Exploring Software Systems. Ph.D Thesis. Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Amsterdam, Netherlands, 2002.
18. Opdyke W. F., Refactoring object-oriented frameworks. Ph.D. Thesis. Graduate College, University Illinois at Urbana-Champaign, Urbana, Illinois, 1992.
19. Ras E. and Weibelzahl S., "Embedding Experiences in Micro-didactical Arrangements," presented at 6th International Workshop on Advances in Learning Software Organisations (LSO 2004), Banff, Canada, 2004.
20. Rech J., "Towards Knowledge Discovery in Software Repositories to Support Refactoring," presented at Workshop on Knowledge Oriented Maintenance (KOM), Banff, Canada, to appear 2004.
21. Roberts D. B., Practical Analysis for refactoring. Ph.D. Thesis. Graduate College, University of Illinois at Urbana-Champaign, 1999.
22. Rus I. and Lindvall M., "Special Issue on Knowledge Management in Software Engineering," IEEE software, vol. 19, 26-38, 2002.
23. Simon F., Meßwertbasierte Qualitätssicherung: Ein generisches Distanzmaß zur Erweiterung bisheriger Softwareproduktmaße (in German). Ph.D. Thesis. Fakultät für Mathematik, Naturwissenschaften und Informatik, Brandenburgische TU Cottbus, Cottbus, 2001.
24. Tahvildari L., Quality-Driven Object-Oriented Re-engineering Framework. PhD Thesis. Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, 2003.
25. Zimmermann T., Weißgerber P., Diehl S., and Zeller A., "Mining Version Histories to Guide Software Changes," presented at 26th International Conference on Software Engineering (ICSE), Edinburgh, UK, 2004.