Towards Knowledge Discovery in Software Repositories to Support Refactoring

Jörg Rech
Fraunhofer IESE
67661 Kaiserslautern, Germany
joerg.rech@iese.fraunhofer.de

Abstract. Software repositories are typically used to store code together with additional information. These repositories are a valuable source to train knowledge discovery algorithms to detect code smells and other qualitative defects. In this paper we present a lightweight framework to detect previously unknown knowledge from software repositories to support refactoring. The results will be usable by software reengineers in the process of inspection and quality assessment of legacy systems.

1. Introduction

During the last years refactoring has become an important part in agile processes to improve the structure of software systems between development cycles. Especially in agile development under-engineering usually happens when the focus lies on adding more functionality to a system without improving its design along the way. When code works it is often simpler to engage the next task than cleaning up the previous work. Additionally, as systems are getting larger refactoring gets more and more complex and time consuming to do manually. Even if one knows how to refactor software it is not clear where and under what conditions what refactoring should be used [1].

In praxis, refactoring is a great challenge, as most software systems are badly implemented and therefore hard to evolve, maintain, and reengineer (e.g., Y2K). This aggravates if the software has to be optimized in order to meet new requirements, remove defects, or improve qualities like maintainability or reusability. Product managers need support to organize refactoring chains and to analyze the impact of changes due to refactorings on the software system. Analogously, quality managers and engineers need information to assess the software quality, identify potential problems, select feasible countermeasures and plan the refactoring process as well as preventive measures.

This paper describes a lightweight framework for the quality-driven, experience- and metrics-based instrument to support the refactoring of large-scale software systems. Developed instruments will give decision-support to software reengineers in the process of managing (i.e., measuring, monitoring, controlling, evaluating, and guiding) corrective, perfective, adaptive and preventive changes (esp., refactorings) to a software system. Based on the problems described our framework is targeted to enable the monitoring and controlling of quality defects (a.k.a. "code smells") in software systems based on software repositories (e.g., nightly, integration and release builds). The semi-automatic diagnosis of quality defects in a software system based on techniques from knowledge discovery in databases will help to detect refactoring candidates. Information from the diagnosis will support maintainers to select countermeasures (e.g., refactorings) and will act as a source for the initialization of preventive measures (e.g., code inspections). The evaluation of the work will be based upon information and source code from open source systems.

1.1. Related Work

Research in software maintenance has been undertaken in several large areas. As Bennett and Rajlich state in their roadmap paper the central research problem is the inability to change software easily and quickly [2]. Current research issues are to gain more empirical information about the nature of software maintenance, to build predictive models, to preserve and manage knowledge for the future maintenance of software, or the restructuring of code and data to remove unnecessary complexity [2, 3].

Previous research has resulted in behavior-preserving approaches to refactoring object-oriented software systems [4], tool support for refactoring application [5], methods for design-pattern based refactoring [6], metrics based and visually supported quality assurance with a similarity measure [7], modeling of object-oriented software to support later reengineering and refactoring [8], automated support for evolution and refactoring of object-oriented frameworks [9], and metrics based visual approaches to understand object-oriented soft-ware systems for reengineering [10]. Publications in the field of this thesis are concerned with collections of refactorings [1] as well as reengineering patterns [11].

Current research in the field of software refactoring is very active and is beginning to address formalisms, processes, methods, and tools to make refactoring more consistent, planable, scaleable, and flexible [12]. Metric based refactoring was currently only done for the refactorings "move method", "move attribute", "extract class", and "inline class" based on one similarity measure with subsequent human interpretation [13].

2. Knowledge discovery and SW repositories

Today, several activities in software engineering like planning, monitoring, controlling, quality improvement, decision support, or automation benefit from knowledge engineering techniques [14]. Knowledge discovery in databases (KDD) is concerned with the detection of previously unknown information from large datasets. Discovery of knowledge is a process that can be divided into the five sub-processes Selection, Preprocessing, Mining, Validation, and Representation [15]. These sub-processes underpin the importance of clean data for the mining process (e.g., numerical without missing data) and the need for representation of clear valid knowledge (e.g., visualization of clusters).

Today, the term data mining is often used as a synonym for KDD. While Data Mining is the detection of "nuggets" in numerical data various forms of mining exists which examines different types of data. For example, text mining focuses on the extraction of knowledge from collections of long texts (e.g., books) while web mining focuses on typically small hypertexts (e.g., web pages), clickstreams, or log data.

The goals of KDD can be divided into the groups cluster discovery (i.e., answering "Are there related elements?"), class discovery (i.e., answering "How to classify elements?"), association discovery (i.e., answering "Do causal relations exist between elements?"), model discovery (i.e., answering "Do valid causal models exist?"), trend discovery (i.e., answering "What will happen in x days?"), pattern discovery (i.e., answering "Are there typical reoccurring structures (e.g., design patterns)?"), and correlation discovery (i.e., answering "Do correlations between (measured) variables exist?").

Specific techniques for these goals like neural networks, decision trees, rough sets, or genetic algorithms can then, for example, be used to construct prediction models for decision support. These techniques as well as fuzzy set theory, case-based reasoning, or Bayesian analysis can be used to support software managers in the planning or controlling of their projects.

KDD promises to support various goals in software maintenance with the detection of knowledge in software repositories. For example, classification techniques can be used to detect similar methods or data structures in software systems. This can either happen on the code itself or on additional information (e.g., software metrics) attached to the code. Another example is the re-classification of methods from old classes into new ones to decrease coupling and increase cohesion of the renovated system. Furthermore, other scenarios are realistic like the automated classification of code fragments (i.e., methods or smaller) for the rapid development of code repositories to support reuse in agile environments.

2.1. Software repositories

Repositories in software engineering are used for nearly all elements, objects, or data related to software. After the speech of McIlroy late in the sixties repositories for code elements became more and more popular [16]. In the early eighties the experience factory (EF) – basically a repository about project experience and products – was established by Victor Basili [17]. Today, various other repositories for configuration management (e.g., CVS, SourceSafe), code reuse (e.g., ReDiscovery, InQuisiX), defect management (e.g., Bugzilla), or project databases exist in software engineering. Furthermore, if nightly- or daily-builds are compiled these also represent file-based repositories with valuable information about a project or software product. Software maintenance and development can benefit from these code repositories (i.e., CVS or nightly builds) if they are used to train defect detection techniques on previous builds of a software product.

In our repository source code from projects is measured and written into an XML document as well as a database for faster access. As depicted in Fig. 1 the code is cut into method blocks and contains metrics on every level ("MetricList"). The source code is attached to methods for later reuse and builds the basis for further diagnosis of defects or reporting.

```
<Package name="views">
 <Class name="TableViewerExample.java"> {
   import org.eclipse.swt.SWT; …
   private Table table; … }
  <Method name="main" modifiers="public"> {
   Shell shell = new Shell();
   shell.setText("Task List Example");
   …
   tableViewerExample.run(shell); }
   <MetricList name="Basic Method Metrics">
    <Metric name="LOC" value="7" /> …
   </MetricList>
  </Method> …
  <MetricList name="Basic Class Metrics">
   <Metric name="NumberOfCasts" value="6"/> …
  </MetricList>
 </Class> …
 <MetricList name="Basic Package Metrics">
  <Metric name="NumberOfClasses" value="6"/> …
 </MetricList>
</Package> …
```

Fig. 1. Repository elements in XML

Comments in methods or javadoc elements are also attached and stored in special tags, but not shown in Fig. 1.

3. Discovery of quality defects in legacy software

Given the fact that activities in software product maintenance account for the majority of the cost in the software life-cycle [2] refactoring is a valid approach to prolong the software lifetime and improve its maintainability. Especially in evolutionary software development (i.e., agile methods) methods as well as tools to support refactoring become more and more important [12].

As shown in Fig. 2 we define six phases for the continuous discovery of quality defects. First we start with the definition of qualities that should be monitored and improved. This may result in different goals as, for example, reusability demands more flexibility or "openness" while maintainability requires more simplicity. Phase two represents the application area for KDD. It is concerned with the measurement and preprocessing of the software to build a basis for the defect discovery. Results form 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 manager or engineer has to decide which refactorings are to be executed in what configuration and sequence in order to minimize work (e.g., change conflicts) and maximize effect on the quality. In phase four the refactoring itself is executed on the software system by the (re-)engineers that results in an improved product. Phase five is used to compare the improved with the original product in order 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 code smells.

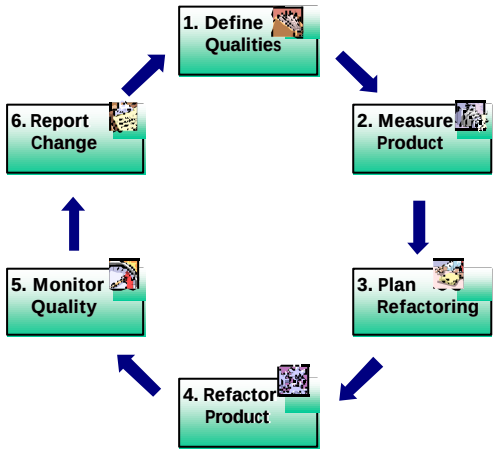

Fig. 2. Quality-driven metrics-based refactoring

As indicated in the previous paragraph the KDD subprocesses are grouped in phase two. We select source code from a specific build, preprocess the code and integrate the results into the software repository, analyze the data to detect quality defects, discover deviations from average behavior, cluster elements with severe or multiple defects, and represent or visualize discovered and priorized quality defects.

For example, we use a classificator to classify if a method belongs to the class "defective methods" (i.e., if it is similar to methods that typically have quality defects). To train the classificator we use old source code and nightly builds to discover potential quality defects. The training code has to previously be analyzed by experts in order to detect and mark potential defects. The classification algorithms can then determine under what attributes can be used by the classifier to distinguish defective from defect-free code. The trained classifier can than be used to discover quality defects in new software or current builds.

3.1. Current Status and Future Work

The current status can be described as work in progress. For the motivation and foundation of our work an extensive literature survey was made. Regarding the technology and tools a first prototype for the measurement of build sequences from software systems is currently in work. To evaluate our approach we will employ a quantitative analysis of open-source software systems, their versions, releases and nightly-build over a longer period of time. For the quantification of the problems in refactoring, evaluate the state of the art and praxis, and to get feedback about specific analysis results it is planed to integrate open-source communities.

To conquer the described problems and reach the chosen goals the following actions and ideas will be realized:

- Investigation of existing metrics as well as the development of new metrics to detect specific quality defects. Histories of nightly builds will be analyzed and used to accumulate massive amounts of data to detect changes and quality defects. Manual investigation of detected defects will help to evaluate the classifier and process. OSS communities will be informed about the analysis (i.e., they get a quality report) in order to receive feedback on the proposed changes and report structure. Investigated metric-defect dependencies will later support the goal-oriented planning of measurement activities with GQM.
- Investigation of existing refactoring as well as the development of new refactorings to increase specific qualities. Existing refactoring catalogs and quality models (e.g., ISO-9126) will be analyzed in order to synthesis a dependency model of refactorings mutually and between qualities and refactorings. Quality measurement plans from our projects or described in literature that are based on software product metrics will be analyzed and relations of metrics to qualities will be used to strengthen the dependency model. Pre-post evaluations of quality changes based on controlled refactoring experiments will be used to assess the impact of refactorings on different qualities. Investigated refactoring-quality dependencies will later support the goal-oriented planning of measurement activities with GQM.
- Elaboration of a quality-driven method to control refactoring processes and know where, when, and why to use what refactoring. A GQM-based process will be defined to support quality and product managers to reach a specific quality goal through refactoring. Furthermore, a CBR based repository of refactoring cases will be created in order to enable managers to reach different goals parallel based on the same metrics or to support him in what metrics he could include to reach other goals.

Optionally, we need to examine if software for specific application areas like embedded, distributed, or knowledge-based systems as well as product lines need additional techniques. In order to detect quality defects in the specifications of hardware (e.g., in VHDL) or knowledge (e.g., in OWL) new metrics might be needed.

Additionally, several side products will be produced like a correlation analysis of metrics to reduce measurement effort, the usage of metrics from different abstraction levels (e.g., requirements), the support of decisions in various phases (e.g., testing), or approaches for the visualization of quality defects in software systems.

4. Conclusion

The proposed framework promises the systematic and semi-automatic support of refactoring activities for product or quality managers. The incremental and low invasive (i.e., cheap) approach for the monitoring of software product quality in order to control refactoring activities will make maintenance activities more simple and increase overall software quality. Optionally, the project manager can use the monitoring of daily builds of the software to detect quality defects and initiate countermeasures during software development.

The framework developed expands the knowledge about quality and its measurement in software systems. It promises knowledge about how to detect quality defects (i.e., where should we refactor?) by software product metrics, the elicitation of knowledge about refactorings and their effect on software qualities (i.e., why should we refactor?), and if and in what sequence to refactor the software (i.e., when and how should we refactor?).

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1st ed: Addison-Wesley, 1999.

[2] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: A Roadmap," presented at Future of Software Engineering Track of 22nd ICSE, Limerick, Ireland, 2000.

[3] H. Müller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley, and K. Wong, "Reverse Engineering: A Roadmap," presented at Future of Software Engineering Track of 22nd ICSE, Limerick, Ireland, 2000.

[4] W. F. Opdyke, "Refactoring object-oriented frameworks," in *Graduate College*. Urbana, Illinois: University Illinois at Urbana-Champaign, 1992, pp. 142.

[5] D. B. Roberts, "Practical Analysis for refactoring," in *Graduate College*: University of Illinois at Urbana-Champaign, 1999, pp. 127.

[6] M. O. Cinneide, "Automated Application of Design Patterns: A Refactoring Approach," in *Department of Computer Science*. Dublin: Trinity College, 2000, pp. 231.

[7] F. Simon, "Meßwertbasierte Qualitätssicherung: Ein generisches Distanzmaß zur Erweiterung bisheriger Softwareproduktmaße (in German)," in *Fakultät für Mathematik, Naturwissenschaften und Informatik*. Cottbus: Brandenburgische Technische Universität Cottbus, 2001, pp. 286.

[8] S. Tichelaar, "Modeling Object-Oriented Software for Reverse Engineering and Refactoring," in *Institut für Informatik und angewandte Mathematik an der Philosophisch-naturwissenschaftlichen Fakultät*. Bern: Universität Bern, 2001, pp. 186.

[9] T. Tourwe, "Automated Support for framework-based Software Evolution," in *Department Informatica*. Brussel: Vrije University Brussel, 2002, pp. 225.

[10] M. Lanza, "Object-Oriented Reverse Engineering: Coarse-grained, fine-grained, and evolutionary software visualization," in *Institut für Informatik und angewandte Mathematik an der Philosophisch-naturwissenschaftlichen Fakultät*. Bern: Universität Bern, 2003, pp. 131.

[11] S. Demeyer, S. Ducasse, and O. M. Nierstrasz, *Object-oriented reengineering patterns*. San Francisco: Morgan Kaufman Publishers, 2003.

[12] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp, "Refactoring: Current Research and Future Trends," *Electronic Notes in Theoretical Computer Science*, vol. 82, pp. 17 pages, 2003.

[13] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," presented at Fifth European Conference on Software Maintenance and Reengineering (CSMR), Los Alamitos, CA, USA, 2001.

[14] L. C. Briand, "On the many ways Software Engineering can benefit from Knowledge Engineering," presented at Software Engineering Knowledge Engineering (SEKE), Ischia, Italy, 2002.

[15] U. Fayyad, S. G. Piatetsky, and P. Smyth, "From data mining to knowledge discovery in databases," *AI Magazine*, vol. 17, pp. 37-54, 1996.

[16] M. D. McIllroy, "Mass-produced software components," presented at NATO Conference in Software Engineering, Garmisch, Germany, 1968.

[17] V. R. Basili, G. Caldiera, and H. D. Rombach, "Experience Factory," in *Encyclopedia of Software Engineering*, vol. 1, J. J. M. (ed.), Ed. New York: John Wiley & Sons, 1994, pp. 469-476.