Individual Code Analyses in Practice

Benjamin Klatt, Klaus Krogmann, Michael Langhammer

FZI Forschungszentrum Informatik, Software Engineering
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany

Abstract: Custom-made static code analyses and derived metrics are a method of choice when dealing with customer-specific requirements for software quality assurance and problem detection. State-of-the-art development environments (IDE) provide standard analyses for code complexity, coding conventions, or potential bug warnings out-of-the-box. Beyond that, complementary projects have developed common reverse engineering infrastructures over the last years. Based on such infrastructures, individual analyses can be developed for project- and company-specific requirements.
In this paper, we oppose MoDisco to JaMoPP as prevailing infrastructure projects for source code analyses, as well as an approach to gain added value from their combination. We further provide insight into two individual analyses we developed for industrial partners. Those example scenarios and the ongoing development of reverse-engineering infrastructure underline the potential of project-specific analyses, which by now is not exhausted by built-in standard analyses from IDEs.

1 Introduction

Ongoing changes in business and usage contexts require continuous adaptation and maintenance of the software systems. According to Seng [SSM06], code quality (e.g. understandability and complexity) and dependency management (e.g. encapsulation and reuse) are success factors for flexible and efficient adaptations. While software grows in size and complexity, taking care of code quality and dependencies becomes expensive and time consuming. Especially with an entirely manual approach, analysis efforts tend to abolish the benefits of the improved adaptability and maintainability.
Over the last decade, several automated static code analyses have been developed to reduce the manual effort for identifying anomalies within a software implementation. Those anomalies range from violated coding conventions up to potential bug detection. Modern software development and management environments provide such analyses out-of-the-box or as extensions. Checkstyle [Che13] and FindBugs [Fin13] are typical tool examples for the Java platform prepared to be generally used by any developer or architect.
However, those widely-used tools are focused on common challenges and problems. Typically, projects and products have also specific and individual challenges to cope with. For example, a project has to reduce its dependency to an unmaintained third party library. Another example is to estimate the change impact of enabling parallel execution (i.e. which source code regions have to run thread safe). Automating such individual analyses does not require a fixed, fully integrated tool, but an infrastructure and know-how to build custom analyses on top of it.
In this paper, we introduce MoDisco and JaMoPP as infrastructures to implement and automate individual analyses. We present their key differences and an approach for their integration to gain added value.

Enabling teams to successfully use such infrastructures is not as easy as applying one of the common code analysis tools. It requires a well-structured approach to start with an initial problem identification and to end-up with a result presentation appropriate for the target audience. We present two example projects from industrial contexts during which we have introduced individual code analyses.

The rest of the paper is structured as follows: In Section 2 we introduce and oppose MoDisco and JaMoPP, followed by some best practices and the presentation of the exemplary industrial projects in Section 3. Finally in Section 4, we conclude our lessons learned and give an outlook how the industrial findings impact our future research.

2 Prevailing Technologies in the Eclipse Context

Today, Eclipse is used as an application platform because of its extendability and its support for model-driven software development. On top of this, the MoDisco [BCJM10] and JaMoPP [HJSW09] projects have developed infrastructures for static code analysis. They are both able to extract Abstract Syntax Tree (AST) models from Java resulting in models based on the Eclipse Modeling Framework (EMF) [Ecl13b]'s Ecore infrastructure. In the following subsections, we distinguish the projects and present an approach to combine the best of both of them.

2.1 MoDisco

MoDisco provides a framework for software model extraction, querying and presentation. The extraction part is strongly related to OMG's *Knowledge Discovery Metamodel* (KDM) specification. OMG's Model Driven Software Modernization task force has developed this specification to provide a standard for software models. It covers an AST model concept for different types of programming languages, an architectural knowledge model (e.g. components), and an inventory model for physical artifacts (e.g. files).

MoDisco provides Ecore implementations of these meta models – except the AST one – and thus is promoted as a reference implementation by the OMG. The project's extendable extraction concept is based on so-called discoverers, each producing software models covering supported types of artifacts. The discoverer provided for Java source code is based on Eclipse's Java Development Tools (JDT) [Ecl13a]. For any further model processing, the models can be persisted by the discoverers. In addition to the discoverers, MoDisco's query infrastructure allows to implement queries on arbitrary Ecore models using Java, OCL, or XPath. Those queries can be executed either programmatically or through MoDisco's user interface. The latter provides a flexible model browser, comparable to the EMF tree editor but with additional browsing and categorization capabilities. Browser and result views are provided for those queries. On top of this query infrastructure, MoDisco provides a facet infrastructure to non-intrusively decorate existing meta models with additional classes and attributes. Summarized, facets allow for comfortable presentations of the model analysis queries. Facets are capable to present additional model elements or attributes if a query evaluates to true. Furthermore, attribute values can be set to the result of a query, even for non-boolean ones. Finally, UI customizations can be used to style the model presentation in the model browser (e.g. coloring or icons), depending on a query result.

Core Concept From a conceptual point of view, a MoDisco discoverer extracts a model by parsing source code. The resulting model is a representation of logical software ele-

ments of the code, decoupled from the original artifacts. Some discoverers also extract an additional KDM inventory as a decorating model linking logical software elements to their physical artifacts (e.g. file).
Limitations The inventory model also contains source code formatting information. This could be used to re-generate the original source code. However, an according reliable model-to-text transformation or generator is not publicly available yet.

2.2 JaMoPP

The JaMoPP (acronym for Java Model Parser and Printer) completely describes its intention to parse Java code into a model representation and to print it back into Java code. JaMoPP is based on the textual modeling framework EMF Text [Dev13] and provides an EMF Text syntax specification for the Java language. An Ecore meta model is derived from this specification and combined with an EMF Text specification respectively a derived Ecore meta model for formatting information. As a result, each Java model element is able to carry formatting information to print source code in a specified format.
JaMoPP provides an according printer as an integrative part. If a software model element has no formatting attached, it is printed in a predefined format.
According to the EMF Text infrastructure, the JaMoPP parser and printer infrastructure are tightly coupled with the EMF resource concept. They are registered as EMF resource reader and writer for `.java` file extension.
Core Concept The JaMoPP core concept is to treat Java source code as a textual representation of a Java model instance. It is tightly coupled with the EMF resource infrastructure to provide a rich and reliable API. The later is ensured by an extensive test suite [HJSW09]. The availability of parser and writer enable round-trip cycles.
Limitations The core JaMoPP tooling does not provide any further processing of the extracted software model. Queries or transformations must be implemented individually but can make use of EMF-compatible infrastructure.

2.3 Comparision

In this section, we summarize the key characteristics distinguishing JaMoPP and MoDisco for individual code analyses. We do not claim for completeness nor for other use cases.
The query and facet infrastructure of MoDisco provides an easy-to-use UI which is also an advantage to promote analyses to development teams. JaMoPP provides no support in this direction. MoDisco publishes performance benchmarks for their discoverers, which is not available for JaMoPP. Also a performance comparison of the two is not available yet. While possible in theory, no reliable code generation is available for MoDisco yet. JaMoPP provides a printer out of the box. From our experience, JaMoPP preserves individual code formatting without any issues. The MoDisco discoverers do not provide the opportunity to influence persistence options of XMI resources. For larger software models, this can lead to non-optimized memory and storage usage. JaMoPP is more lightweight and leaves the EMF resource configuration up to the developer.
MoDisco makes at least partially use of a standardized meta model while JaMoPP completely relies on a proprietary specification. However, for the analyses, the AST model is the most important part which is proprietary in MoDisco as well. From our personal experience, we discovered some lacks in MoDisco's AST model extraction (e.g resolving labeled statement references, and handling of qualified type accesses) but had no issues

with the JaMoPP extraction. From our personal experience, the MoDisco project lacks support for bug fixes. Even provided patches are not integrated into the project. In contrast, we received quick and helpful responses from the JaMoPP project.

2.4 MoDisco-JaMoPP Integration

As outlined in the comparison above, on the one side, JaMoPP provides a more reliable and lightweight extraction (parser). Additionally, the code generation (printing) out-performs MoDisco's capabilities. On the other side, MoDisco provides a powerful infrastructure for queries and model decoration. Due to the query and facet infrastructure's applicability to Ecore models in general, they are not limited to models extracted by MoDisco discoverers. As illustrated in Figure 1, we have integrated JaMoPP's parsing and printing capabilities, with MoDisco's query and presentation features. Due to the common Ecore infrastructure, this integration can be used without any tool adaptation.

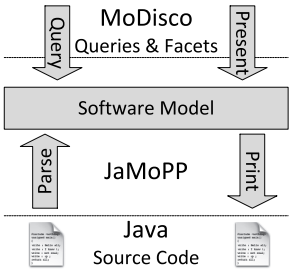

Figure 1: JaMoPP - MoDisco Integration

Illustrating Example Singleton Analysis To give an idea about the straight forward implementation of MoDisco queries for JaMoPP-parsed Java models, we have prepared an example to analyze a software for instances of the Singleton pattern (see [GHJV95] page 127ff). We have published this as an Eclipse project on GitHub [1].

In [Mar13], Lars Martin presents an example using MoDisco to detect Singleton Patterns in Java source code based according to an algorithm described in [Nau01]. A class is identified as singleton if it has i) a static field typed with itself or one of its subclasses, ii) a static getter, and iii) a private constructor.

We have migrated this singleton-detection-query to analyze JaMoPP-extracted software models. In addition, we have created a MoDisco facet representing instances of the Singleton pattern as described by Martin [Mar13].

Thanks to JaMoPP's concept of treating source code as a textual representation of a model, you can drag a Java file and drop it into the MoDisco Model Browser. Activating the facet for the Singleton pattern, the MoDisco model browser shows instances of the Singleton pattern in the file. The screenshot in Figure 2 presents the result of analyzing the exemplary singleton in Listing 1.

[1] https://github.com/kopl/misc/tree/master/examples/org.kopl.jamopp.modisco.pattern.query

```
package org.kopl.singleton.example;
public class MySingleton {
 private static MySingleton instance
     = new MySingleton();
 private MySingleton() {}
 public static MySingleton getInstance(){
  return instance;
 }
}
```

Listing 1: Analyzed Singleton Example

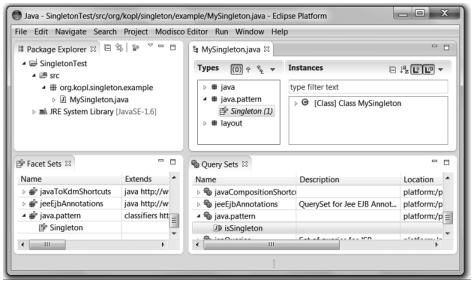

Figure 2: Screenshot Singleton Result

3 Best Practices in Analyses

Applying standard, strongly integrated code analyses is common practice today. Supporting individual challenges with custom code analyses is still not completely understood yet. As far as our experience with industrial projects goes, development teams either do not know about today's possibilities for code analyses at all or do not know how to apply them in their own use case.

We follow four major steps: 1. Problem analyses, 2. Query design, 3. Presentation and KPIs, 4. Actions derivation. To ensure the development of a useful and expressive analyses, first, the problem and analysis goals must be understood and structured to answer the right questions later on. Next, the query must be designed to consider the right elements (e.g. expressions or types) and to produce the appropriate type of results (e.g. true/false or lists of elements). In addition, the analysis-algorithm must be designed with respect to correctness, precision, and performance. Afterwards, an appropriate result presentation for the intended target audience must be developed. The presentation can range from element lists, to visual diagrams, to aggregated Key Performance Indicators (KPIs). Finally, in nearly all cases analyses are performed as a preparation for further actions. Actions include refactoring decisions, task lists, or automated software modifications.

While presented in a strict order, the process must be done in an iterative manner. Meaning that each previous step should be considered again if a potential improvement is detected in a latter one, if the system evolves or quality requirements change.

We have successfully applied this process in a broad range of industrial applications. For example, in a project in the automation industry, we analyzed a software's dependency to third party libraries. As described in [KDK+12], we have developed individual analyses to provide KPIs, beside others, about the adaptability and future perspective of a software system. As a completely different example, we have developed code analyses to accompany a company during a software migration from EJB 2 to EJB 3. In this use case, the analyses have been used to identify new hotspots of potential problem patterns, which come up during the migration. While the migration of the second example is done now, the analyses developed in the first project example, especially the KPIs, are nowadays embedded in some of the company's business units.

In all cases, the automation lowers the effort for analyzing software systems. Hence, larger software systems can be analyzed more frequently and more thoroughly.

4 Conclusion and Outlook

In this paper, we have introduced MoDisco and JaMoPP as Eclipse-based state-of-the-art reverse-engineering infrastructures for Java. We opposed the project's tools and presented a value-adding integration of both of them. Furthermore, we gave examples of industrial applications for individual code analyses.

The results of those analyses lead to ongoing research in this area. We investigate especially in the combination and reuse of existing reverse-engineering tools, code analysis infrastructures, and model-driven modernization techniques.

The experience from industrial applications has led to the KoPL project [2]. The project aims to provide automated support for consolidating customized product copies into a common flexible and sustainable software product line.

References

[BCJM10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and F. Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 173–174. ACM, 2010.

[Che13] Checkstyle Team. Checkstyle. http://checkstyle.sourceforge.net/, 2013.

[Dev13] DevBoost. EMF Text. http://www.emftext.org/, 2013.

[Ecl13a] Eclipse Foundation. Eclipse JDT – Java Development Tools. www.eclipse.org/jdt/, 2013.

[Ecl13b] Eclipse Foundation. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/, 2013.

[Fin13] FindBugs Development Team. FindBugs – Find Bugs in Java Programs. http://findbugs.sourceforge.net/, 2013.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[HJSW09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and java. In *Proceedings of the Second international conference on Software Language Engineering (SLE'09)*, pages 374–383. Springer-Verlag Berlin, Heidelberg, 2009.

[KDK+12] Benjamin Klatt, Zoya Durdik, Heiko Koziolek, Klaus Krogmann, Johannes Stammel, and Roland Weiss. Identify Impacts of Evolving Third Party Components on Long-Living Software Systems. In *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 461–464, Szeged, Hungary, March 2012. Ieee.

[Mar13] Lars Martin. Tanz unter der MoDisco Kugel. *Eclipse Magazin*, 06.13:38–44, 2013.

[Nau01] Sebastian Naumann. *Reverse- Engineering von Entwurfsmustern*. Diploma thesis, TU Illmenau, 2001.

[SSM06] Olaf Seng, Frank Simon, and Thomas Mohaupt. *Code Quality Management*. dpunkt Verlag, Heidelberg, 2006.

[2] http://www.kopl-project.org