Towards Rule Based Refactoring

Lukáš MARKOVIČ*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
`lukass.markovic@gmail.com`

Abstract. This paper presents the advanced approach to source code refactoring using XML technologies and rule based expert system. The article also describes particular refactoring problems such as dependencies between code smells, or order effectiveness of refactoring operations. Our attention is dedicated not only to well-known code smells, but also similar concept called anti-patterns. Some of these problems are used as example to explain possibilities of expert system with defined rules in refactoring process. Main part of the article describes a proposal of software system, able to perform automated refactoring using rule based decision making based on code smells dependencies analysis.

1 Introduction

Refactoring, first defined by Martin Fowler, is a process of changing a software system source code in a way, that it does not affect external behaviour, but yet improves its internal structure [2]. Besides refactoring definition, Fowler also describes the target of refactoring – code smells, as indications of a deeper problem in source code structure. In addition to code smells, there are also anti-patters, firstly mentioned by Andrew Koenig [3]. They are described as commonly occurring solutions that can cause problematic consequences [1]. There is only small difference between code smells and anti-pattern. These terms are therefore often being confused, or used without a difference. In fact, by definition, only code smell should be the real target of refactoring, because anti-patterns can also cause system problems. Removing an anti-pattern can therefore cause change in system behaviour, and that is not consistent with the refactoring definition. Despite of these terminology problems, refactoring process is nowadays used for removing both code smells and anti-patterns from the source code. Software system thus becomes easier to maintain, alternate, adapt and further develop.

However refactoring costs time, it is very necessary process during the life cycle of almost every software system. Therefore, there is a great effort in refactoring automatization. Also here in Faculty of Informatics and Information technologies was created a lot of research studies in field of refactoring automatization [7][8], [9][11]. This paper, which is related to previous research works proposing the rule-based refactoring [9], [11], is dedicated to the less known approach to

* Master degree study programme in field: Software Engineering
Supervisor: Dr. Ivan Polášek, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

refactoring automatization using XML technologies in combination with expert system based decision making, for obtaining the best possible result from automated refactoring process.

2 Automatized source code refactoring

Refactoring automatization is a very difficult process that varies based on the selected technologies of representing the source code, searching for anti-patterns or code-smells and refactoring of these problems itself.

Code smell search and refactoring can be performed on many source code representations. Technique of code smell search and refactoring itself is subsequently unwound based on selected representation. Table 1 describes some of the possible source code representations and corresponding techniques of code smell search and refactoring.

Table 1. Possible representations and corresponding techniques of code smell search and refactoring.

Source code representation	*Search technique*	*Refactoring technique*
plain text	metrics	text processing tools
AST	AST libraries, metrics	AST libraries
JSON	JavaScript, …	JavaScript, …
XML	XPath, XQuery	XSLT, XQuery, XQuery Update

As presented, automated refactoring can be performed on a clean code itself. Then, many well-known source code metrics can be used for code smell search [10], such as a *lines of code*. On the other hand, refactoring is very hard to perform on such a "*clean*" representation. One option is to use text processing tools, that are able to work with simple text, but this option is complicated and unnecessary.

Abstract syntax tree is probably the most widely used method for refactoring automatization. This method is based on a tree representation that removes all insignificant source code elements, for example parentheses or spaces. Many languages, or IDE's provides libraries, that are able to programmatically work with AST representation. Typical example is Eclipse AST library that is used in many refactoring and source code manipulation tools under Eclipse [4].

Despite that most widely used technique for refactoring automatization is abstract syntax tree, in our work we focused on XML technologies. XML, similarly as JSON notation allows serialization of source code. Afterwards, searching and refactoring in such representation can be simply performed with technologies that are able to manipulate with given representation [5].

3 XML based refactoring process

XML language, which is a widespread language for data serialization can be also used for a source code representation. There are several tools able to convert source code into XML representation and vice versa[1]. Many XML technologies can be then used for search of an anti-patterns and code smells [6]. There is also a lot of useful Technologies [6], such as XSLT[2] or XQuery[3], for refactoring itself.

One problem with XML representation is, that every source code file is represented as a single XML document. When refactoring is performed, there is often a needs to work with more than one file in time. For example, refactoring method "*push up method"* brings selected method

[1] http://www.srcml.org/
[2] https://www.w3.org/TR/xslt-30/
[3] https://www.w3.org/TR/xquery-3/

into other class, that is often in its own file. This means, there is need to work with system source code as with single document, which removes the need to solve documents interconnections.

Solution to these problems are XML databases, that are kind of NoSQL documents databases. Project can be represented as a single collection of documents in this kind of databases. Each query into project collection works with project de facto as one document, and this effectively solves presented problem.

Nowadays, there is only few actively supported native XML databases. These includes:

- BaseX
- eXist
- MarkLogic
- Sedna

Despite that XML databases are relative old, not a wide spread technology, they are still supported and they have actively developed an interface for many languages, such as Java – XQJ API[4].

XML databases find only small usage nowadays, but are highly suitable for application in XML based refactoring process. They also usually offer processor of many XML technologies, such as XPath, XSLT or XQuery. XML Technologies can be used to search and mark a concrete problem and also to refactor problems in the source code, when using XML databases. XML based processes of anti-pattern or code smell search and refactoring will be described in the following subchapters.

3.1 Code smells search using XML technologies

As described in [7], XPath language is very suitable tool for XML based code smells search. Despite its simple, not hard to learn language, it is able to effectively locate and return desired nodes from XML documents. Using simple XPath expressions, it is possible to detect a lot of common, as well as little-known code smells. However, there are some reasons to select different XML associated language for code smell search.

XQuery, which used XPath for node locations, is a language for querying in XML documents. Native XML database systems are often using XQuery as main language to query into database. According to this and also needs to store information’s about found problems, what XPath is not able to, the XQuery language is probably best technology for search and also refactor code smells, as described in subchapter 3.2.

3.2 Refactoring of code smells using XML technologies

Refactoring itself is the most difficult part of refactoring process. There are several possibilities to select a refactoring technology, for example XSLT [7]. Another options are XQuery and its extension XQuery Update. These are able of functional programming over XML documents, and are more sufficient in case of proposed tool.

Refactoring, as an operation of source code structure changing, requires modifying of XML documents in the database. Only XQuery itself is insufficient for this purpose, because it does not contain operations of modifying the document itself. This problem is solved by *XQuery Update* extension. Example of XQuery script is provided in fragment code below. Very simple refactoring operation *Remove exception throw* is shown, in which *delete* operation is XQuery Update command.

```
declare variable $tag external := "CR1";
       for $node in xquery:eval(fn:concat("//", $tag))
return
       delete node //throw
```

[4] http://xqj.net/

4 Rule based refactoring

Refactoring has many areas, which need to be considered during the process. One of them is the influencing of the code smells each other's.

Influencing means that by removing one code smell, other code smell can arise on a given place. Typical example is, that by removing *middle man, message chains* code smell often arises. Besides this, positive code smell influencing can also be taken into mind. This means, that by removing one code smell on given place, other code smell can be removed as well. This behaviour is very interesting, knowing that every refactoring affects the structure of the code. In general, the target of refactoring is to remove code smells in effective way. This means with as minimum as possible refactoring operations. Removing code smell, which removes also other code smell on given position is better, than firstly removing inner code smell and then removing outer code smell.

These two kind of influences between code smells are simple to take into consideration when refactoring is performed in a manual process. But during refactoring automatization, it is hard to consider these influences. One possibility is to introduce rule based expert system, able to decide about optimal or sub optimal refactoring sequence. An expert system, which contains knowledge of code smell influences, can be then included into automated refactoring system. An expert system can select refactoring operations and their order based on search analysis of source code. Nowadays, there are few tools able to create such an expert system. Jess[5] tool, which represents expert system programing language based on Java language is one possibility which was also used in proposed tool.

4.1 Dependencies between code smells

Identification of positive and negative influences between code smells is necessary in order to build such a rule based on refactoring system.

Identification of influences between each code smell and anti-patterns is very difficult, because more than one hundred source code problems are identified. In proposed tool we focused on identification of influences between the well-known 22 code smells defined by Martin Fowler. Despite of that, the number of problems is reduced into 22, there can still be identified a lot of relations between them. The reason for this is that each code smell can be removed by application not only one refactoring operation. Each operation can arise, or remove different type of code smell.

Due to these complicated relations, it is necessary to introduce the concept of code smell groups, which can be presented in form of super classes for concrete code smells. Consequently, it is possible to simplify relations between code smells using these subclasses. Identified group of code smells are presented in Table 2 below.

Table 2. Classification of code smells into groups.

Group	*Code smells*
Bad Size	Large Class, Long Method, Long Parameter List
Bad Location	Feature Envy, Comments, Duplicated Code, Divergent Changes, Shotgun Surgery, Switch Statement
Bad Class Content	Data Class, Lazy Class, Feature Envy, Large Class
Bad Inheritance	Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies, Refused Bequest
Needless Part	Comments, Duplicated Code, Refused Bequest, Speculative Generality
Attribute Problem	Data Clumps, Temporary Field, Primitive Obsession
Bad Communication	Inappropriate Intimacy, Middle Man, Message Chains

[5] http://www.jessrules.com/jess/index.shtml

Given the complex character of some code smells, it is appropriate, to include them into more than one group. With such hierarchy of code smells, there is possibility to create complex views at both kinds of code smells relations. UML class diagram is a sufficient tool for creating such views, as can be seen on Figure 1, that presents positive dependencies between code smells.

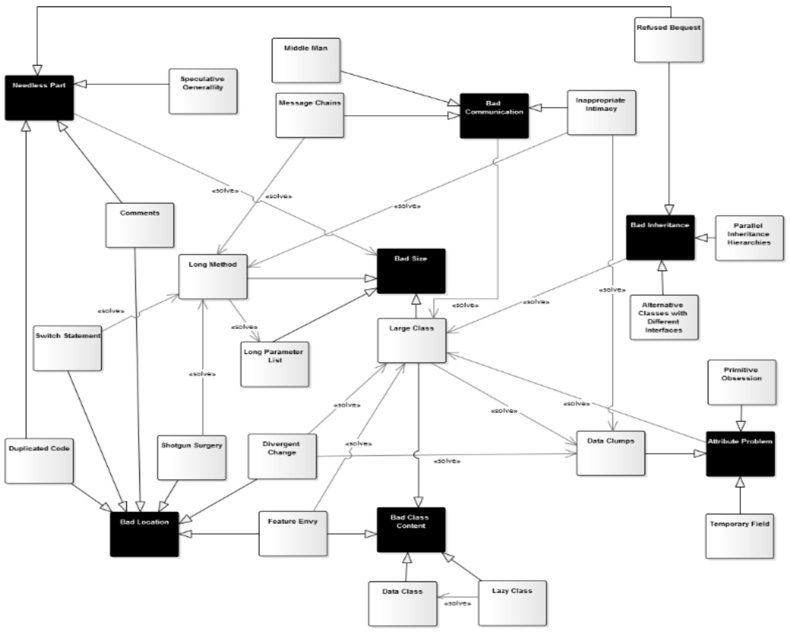

Figure 1. Positive dependencies between code smells.

Solid arrow in combination with <<*solve*>> stereotype is used for positive dependencies, as can be seen in Figure 1. On the other hand, for the negative dependencies (refactoring can cause another smell) <<*cause*>> stereotype and dashed arrow can be used and similar diagram can be created.

4.2 Design of refactoring rules

It is possible to straightforward design refactoring rules based on previous analysis of code smells dependencies. Based on given rules, rule engine can decide about selected refactoring operations.

Such refactoring rule can be very simple, in form of *if – then* rule, as presented below.

```
(defrule empty-catch-clause
              "Empty catch clause refactoring"
              ?o <-(JessInput {refCode == "ECC"})
              =>(add (new JessOutput ?o.code "LE")))
```

However, it can also take into consideration other conditions, for example the size of concrete problem, or, as presented, dependency to other found problem on given place. On the code fragment below, there is a simple Jess rule, which takes such dependency between code smells into account.

```
(defrule long-parameter-list
       "Long parameter refactoring with possible collision check"
       ?o <-(JessInput {refCode == "LPL"})
       (JessInput (parents ?parentsList))
       (not(test (?parentsList contains "LM")))
       => (add (new JessOutput ?o.code "IPO")))
```

Based on the second presented rule, there is only selected refactoring operation *Introduce parameter object* for refactoring of code smell *Long parameter list*, only if code smell *Long method* was not found in place. Otherwise, *Long parameter list* refactoring will not be executed, before *Long method* refactoring. On the other hand, based on first presented rule, *Log Exception* refactoring method is selected unconditionally as solution for every *Empty Catch Clause* problem.

5 Conclusions

This paper presented new, advanced approach to refactoring. Rule based refactoring, in combination with XML technologies, offers great potential into the future. XML database, that has strong processing power is a very important part of XML based refactoring process. In future work, there is a need to examine dependencies between all known code smells more precisely. Refactoring rules and strong refactoring tool can be created based on such analysis.

Acknowledgement: This contribution is the partial result of the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] Brown, H.B., Malveau, R.C., McCormick, H.W, Mowbray T.J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., (1998).

[2] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, (1999).

[3] Koenig, A.: Patterns and Antipatterns. In: *Journal of Object-Oriented Programming*, (1995), vol. 8, pp. 46-48.

[4] Kuhn, T., Thomann, O.: *Abstract Syntax Tree*. [Online; accessed November 20, 2006]. Available at: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST

[5] Mamas, E., Kontogiannis, K.: Towards Portable Source Code Representations Using XML. In: *WCRE '00 Proceedings of the Seventh Working Conference on Reverse Engineering*, IEEE Computer Society, Washington, (2000), pp. 172–182.

[6] Markovič. L.: *Refactoring support using transformations between Java a XML*. Bachelor's thesis, Slovak University of Technology in Bratislava, (2014).

[7] Markovič. L.: Refactoring Support Using XSLT Transformations. In.: *IIT.SRC 2014 Proceedings in Informatics and Information Technologies Student Research Conference*, Slovak University of Technology in Bratislava, (2014), pp. 457-462.

[8] Pipík, R., Polášek, I.: Semi-automatic refactoring to aspect-oriented platform. In: *CINTI 2013: proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics*, Budapest, (2013), pp. 141-145.

[9] Polášek, I., Snopko, S. Kapustík, I.: Automatic identification of the anti-patterns using the rule-based approach. In: *SISY 2012: IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, Subotica (2012), pp. 283-286.

[10] Simon, F., Steinbrückner, F., Lewerentz, C.: Metrics based refactoring. In: *CSMR '01 Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, Lisbon, (2001), pp. 30–38.

[11] Štolc, M., Polášek, I.: A Visual based Framework for the Model Refactoring Techniques. In: *SAMI 2010, 8th IEEE International Symposium on Applied Machine Intelligence and Informatics*, Herľany, (2010), pp. 77-82.