Towards Automated Design Smell Detection

A Proof of Concept in Detecting Opportunities for the Strategy Design Pattern

Stefan Burger
Software Engineering Group
University of Mannheim
Mannheim, Germany
sburger@mail.uni-mannheim.de

Oliver Hummel
Software Design and Quality
Karlsruhe Institute of Technology
Karlsruhe, Germany
hummel@kit.edu

Abstract— **Patterns are widely seen as an important ingredient to improve structure and maintainability of object-oriented software designs. In order to fully recognize opportunities for them, however, developers usually need a lot of experience as well as a good understanding of a given system. Hence, they often miss possibilities to use design patterns and produce code containing "design smells". With a view to overcome this unsatisfying situation, we have derived predicates that allow automatically identifying those locations in software systems where the Strategy design pattern would be beneficial. Moreover, we have implemented a prototypical tool that is able to apply these predicates. Using it on eight open-source projects with roughly 850K lines of code as an explorative study has discovered a variety of places where the pattern would improve the design. As ongoing work has demonstrated that this approach is transferable to other patterns, we believe that it has a good potential to increase the use of design patterns and therewith code quality in the not too distant future.**

Keywords-Design Patterns; Pattern Recommendation; Strategy; Code Quality

I. INTRODUCTION

Creating a clean and comprehensible design is probably one of the most challenging aspects in the development of complex software systems [1]. Hence, it does not surprise that it usually requires a lot of time and experience until software engineers have mastered all subtleties involved therein. In order to mitigate this steep learning curve, the object-oriented development community has collected a comprehensive set of so-called design patterns over the last decades. The best known pattern compilation is probably the seminal book of Gamma et al. (the "Gang of Four" (GoF), [2]) that lists 23 of them. However, since patterns are merely abstract solutions for common problems, they need to be tailored to a given context and consequently, applying the right pattern in a concrete situation is already a challenge in itself.

In order to break out of this vicious circle, the support of a (potentially proactive) recommendation system [3] that is able to recognize and suggest opportunities for the use of design patterns directly in common programming environment certainly seems like a promising idea. In recent years, numerous recommendation engines have been developed, including tools intended to simplify the usage of complex application programming interface (API) [4] or generally aiming on increasing the amount of reuse in software development [5][6]. However, despite the popularity of design patterns, there have only been few attempts to automate the detection of existing patterns in source code (such as [7]). Obviously, the idea of detecting pattern opportunities is remotely related with works on smell detection in the context of refactoring (such as by van Emden et al. [9]). Nevertheless, pattern recommendation requires an "understanding" for larger source ensembles that is usually not necessary for the relatively fine granular refactorings collected in Fowler's well-known book [10]. One important work on pattern recommendation has been presented by Briand et al. [11]. The authors presented a semi-automated decision support system intended to help developers find places for the use of patterns in Unified Modeling Language (UML) design diagrams and proved its feasibility for one pattern on a small case study with 15 classes. [8]. To the best of our knowledge, the only approach that directly aimed at automatically recommending promising "hot spots" in the code for the use of design patterns so far was recently presented by Christopoulou et al. [25]. We will discuss this and other related work in more detail in Section II.

Hence, the fully automatic approach for the detection of "design smells" and prospective design patterns based on static code analysis we describe in this paper is entering a largely unexplored territory. The most obvious benefits of such a pattern recommendation system are its support for novice developers who want to learn about good design in order to enhance the structure of their code. Moreover, it would also disburden experienced colleagues, for whom the recognition of pattern opportunities often still remains a challenging cognitive task, even after decades of experience [12]. Finally, such a system could also be used to get a new impression on code quality, as it would allow judging whether a system is well structured or still bears improvement potential in terms overlooked pattern opportunities.

In order to explain our pattern recommendation approach, we start by briefly discussing related work on design patterns and refactoring recommendations in Section II. In Section III, we exemplarily explain our pattern recommendation approach with the Strategy pattern, before we discuss how meaningful thresholds for the used metrics can be found in Section IV. The section following thereafter briefly describes the prototypical tool we have developed and its application on eight open source systems with 850 thousand links of cod (KLOC), before we conclude our paper with an outlook on future work and a summary of its contributions.

 ISBN: 978-1-61208-367-4

II. Related Work

The general idea of supporting developers in the selection of design patterns in order to improve source code quality has been discussed in various publications. However, the degree of automation so far used to be low. One early approach that has been published by Palma et al. [14] proposes the use of an expert system. It is based on the Goal-Question-Metric (GQM) method and uses a specific question template for every pattern. A developer can go through these templates in order to find the best matching pattern for a specific situation. However, this approach is completely manual and independent from the actual source code. Durdik et al. [12] have also been working on a set of questions intended to help documenting decisions for design patterns in order to facilitate replicability and hence program maintenance and evolution. A different approach was presented by Suresh et al. [15] who were using information about pattern usage (motivation, consequences, etc.) from other developers to create a pattern recommendation system. Again, the recommendation is based on disruptive questioning about a given situation and has no direct connection to the source code. Briand et al. [11] have proposed a similar semi-automatic approach that uses decision trees to identify places where GoF patterns might be useful within UML designs. Since not all necessary information can be derived automatically by this system, the developer needs to answer questions there as well. Moreover, their approach, supporting seven patterns in total, needs a comprehensive set of UML design diagrams that is often not available in practice. To our knowledge, the only approach similar to our work was recently published by Christopoulou et al. [25]. Their work also focusses on identifying Strategy pattern candidates, however they merely use an analysis of conditional statements without analyzing the surrounding method or class. Moreover, they do not give any rationale when it is worthwhile to recommend a pattern.

While such pattern recommendation is a relatively new research strand, automatically identifying potential code smells and related refactorings have been researched to some extend in recent years: As an example, consider Seng et al. [16] who have utilized software metrics in order to detect code smells and therewith identify potential places for code refactorings. However, the recommendations generated by their tool tended to break higher level structures such as design patterns. Hegedűs et al. [17] aimed to connect the usage of design patterns with software maintainability. They measured several hundred revisions of the open source project JHotDraw [27]. During their analyses, they found evidence that patterns can improve source code quality. Huston [18] analyzed the effects of design patterns on applications and their metrics scores. He developed a mathematical model based on software metrics (such as Coupling between Objects) to compare source code with a pattern and the same code without a pattern. His conclusion was that patterns can reduce high metric scores, but the usage of software metrics seems generally questionable in this context. This conclusion is also supported by Burger and Hummel's work that showed that refactorings often worsen metric values. Tourw'e et al. [28] have been working towards detecting refactoring opportunities or, in other words, code smells [10]. They are using logic meta-programming (LMP) for identifying smelly structures in the source code and for choosing an appropriate refactoring.

Another interesting challenge is identifying already implemented patterns in a given source code to be able to assess whether they have been applied in a meaningful way. The pattern detection community, e.g., comprising researchers like Baranski et al. [6] and others has been tackling this challenge for several years and has reached significant results, i.e., they have created pattern detection tools using various different technologies and approaches. Heuzeroth et al. [8] use static analysis of the source code for this purpose. Guéhéneuc et al. [20] have developed a combined approach based on a numerical signature (e.g. size/complexity, number of methods/parents, etc.) and a structural analysis of code files to identify design patterns. Tsantalis et al. [7] have proposed an approach which uses graph algorithms for identifying potentially modified design patterns. Fabry et al. [19] have developed an approach for detecting existing patterns, which is independent from the used programming language through extracting meta-information, such as method calls or variable references from the parse tree for this purpose.

III. Pattern Recommendation

This section explains our generic approach for a fully automated recommendation of design patterns and the necessary steps for detecting concrete candidates (we will use the abbreviation DPC for "design pattern candidate" in the following) in a source code, exemplarily using the GoF Strategy pattern to illustrate it. Step one of our approach is deriving the abstract syntax tree (AST) of a given Java source code, i.e., usually one .java file. Step two is extracting the necessary information (metrics and structural information) from the AST as a base for identifying DPCs. For this purpose, we aim to create a predicate for each supported pattern (to be explained in the upcoming subsections) that helps in recognizing the candidates. A DPC is found whenever all metric thresholds of a predicate are triggered by the underlying source code, or in other words, whenever the predicate evaluates to true. A graphical summary of this process is presented in Figure 1 .

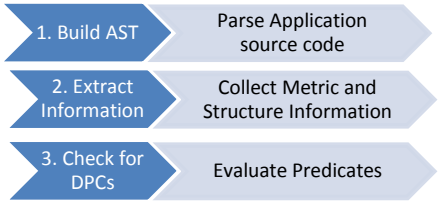

Figure 1 Pattern Candidates Identification Process.

Based on this model, we exemplarily describe the predicates we have defined for the Strategy pattern (see Figure 2) in the following subsections in more detail. According to Gamma et al. [2] the Strategy pattern is defined as follows: "*Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*" Following this defini-

 ISBN: 978-1-61208-367-4

tion, the important part of the Strategy design pattern is the separation of different algorithmic strategies from the context in order to better support the open/closed principle, which states that code should be open for extension without the need for modification [21]. Thus, the strategies are implemented independently in separated classes that each gives home to a "family" of different algorithms (i.e. the methods A and B in Figure 2). Obviously, they all need to implement the common IStrategy interface that defines which methods should be available.

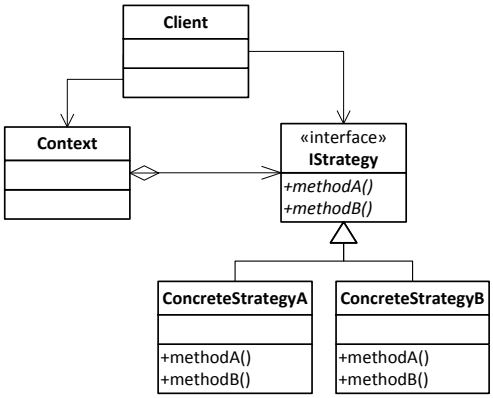

Figure 2 Class diagram for the Strategy pattern.

Although the GoF book and other literature (as e.g. [12]) provide some general guidance when to use a pattern, all are relatively imprecise when it comes to concrete rules for actually using a pattern. For an automatic recommendation system, however, it is obviously essential to define precise detection rules with good thresholds so that a suggested pattern is helpful and does not induce more complexity than it actually resolves.

The predicate for the Strategy pattern differs only slightly from the one for the State pattern, which we have also started to investigate. This is no surprise, since both patterns are aiming at encapsulating program behavior in separate classes in order to make it more exchangeable. Hence, candidates for both can be detected within large conditional (if/switch) statements depending on the same variable. The central difference of the two is conceptual: states typically "decide themselves" when an object should switch into another state in order to behave differently. For Strategy, this decision is triggered by an external event, such as a decision of the developer or the user of a system so that no object variable should be changed in the body of the conditional. Thus, the predicate for the Strategy pattern can be written as in Table I.

TABLE I. PREDICATE FOR STRATEGY CANDIDATES.

Rule	Description
R1.1 **OR**	*In serveral methods of a class there exists an if/switch statement, which has a similar number of cases and uses the same attribut or parameter in the condition.*
R1.2	*In a class hierarchy there exists a number of subclasses, which are all overriding the same method(s) of the super class.*

Each rule of the predicate aims at identifying a different design smell indicating a possibility to use the Strategy pattern and is composed of a number of metrics based on code characteristics like number of subclasses or common attributes. If all metrics of a rule are passing a predefined threshold, a smell has been identified. Table II describes the metrics defined for the rules R1.1 and R1.2.

TABLE II. METRICS FOR STRATEGY DETECTION RULES.

No.	Metric	Type	Rule
M1.1	*Number of methods containing a conditional statement*	*Numeric*	*R1.1*
M1.2	*All methods of M1.1 are in the same class*	*Boolean*	*R1.1*
M1.3	*Every method identified in M1.1 has a conditional with an identical number of cases*	*Boolean*	*R1.1*
M1.4	*There exists a common attribute / parameter used in all cases of M1.1*	*Boolean*	*R1.1*
M2.1	*Common super class*	*Boolean*	*R1.2*
M2.2	*Number of overridden methods*	*Numeric*	*R1.2*
M2.3	*Number of subclasses overriding the same method*	*Numeric*	*R1.2*

In order to avoid choosing "arbitrary" thresholds, we have chosen them based on a careful analysis of numerous Strategy implementations retrieved from the Merobase software search engine [23], as explained in the next section.

IV. THRESHOLD DEFINITION

One of the most critical aspects for the acceptance of our envisaged approach is determining the thresholds that trigger a recommendation. Only with meaningful thresholds, it is possible to decide if a metric result indicates a design smell that should be resolved through the use of a design pattern or not. Thus, in this section, we explain exemplarily how we have derived the thresholds for the Strategy pattern. It should nevertheless be obvious that this procedure can also be used for the analysis of other patterns. The basic process contains four different steps, beginning with identifying the needed characteristics of the target pattern, i.e., the rules that might indicate the use of a pattern (cf. Table II). In order to establish grounded thresholds for a pattern, we considered an empirical analysis of existing pattern uses as the best solution so that the second step aims at identifying them with the help of a code search engine. The next step then is to measure the characteristics defined in step one for the discovered pattern instances. Finally, the thresholds can be derived from the measured values through a statistical analysis. Figure 3 illustrates the overall process graphically.

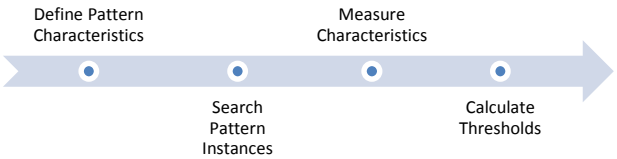

Figure 3 Finding meaningful thresholds for pattern recommendation.

The definition of the pattern characteristics in step one can be analytically derived from the explanation of a pattern, as described in the last section. Spotting concrete pattern instances in source code, however, as needed for the second step, is still an area of active research (e.g. [20]) without any tools that would be readily usable "off the shelf". Since we

 ISBN: 978-1-61208-367-4

nevertheless needed to come up with a way for finding a serious number of pattern instances with a reasonable amount of effort, we decided to use a software search engine. Since we are not aware of any search engine that would recognize patterns based on their structure, we needed to "guess" names that could be used as search terms. Fortunately, the Merobase [5] search engine allows to search for wildcards under a certain constraint, which is that the asterisk as wildcard character cannot be used as the first or the last character of a search. Under the assumptions that many Java programmers start interfaces with a capital 'I', as, e.g., suggested by Beck [22] and that the pattern name will also be reflected in the interface, we derived the following query:

*I*Strategy lang:java type:interface (protocol:svn OR protocol:CVS) original:yes*

Moreover, as is visible in the query, Merobase is able to limit searches on a desired programming language (here: Java), and a certain file type (i.e. interfaces). Moreover, we limited our analyses on Subversion (SVN) and Concurrent Version System (CVS) repositories as we assumed to find more mature projects there than in results from the open web and excluded identical duplicates. Thus, the delivered results contain every Java interface that starts with an 'I' and ends on Strategy. Merobase finds something over 250 matches for this query. We have analyzed the first 50 projects of the result set with a maximum of three patterns from one project in order not to bias the results towards the habits of a specific project. Moreover, we filtered out about 33 obviously "incorrect" implementations that did not comply with the recommendations of the Gang of Four [2] (e.g. they were just implementing a single Strategy) so that a total of 68 Strategy implementations remained.

The histogram in Figure 4 illustrates the size of Strategy implementations on the x-axis, i.e., how many subclasses of the Strategy interface or methods the analyzed instances of the pattern contained. The y-axis shows how often each case has occurred during the analysis.

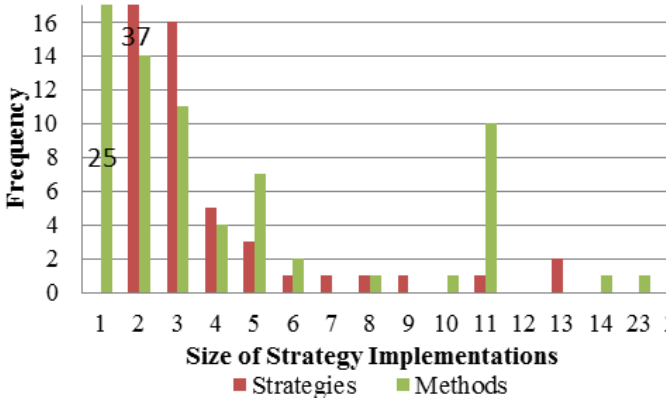

Figure 4 Distribution histogram of strategies and methods.

A statistical overview of the analysis results is shown in Table III.

TABLE III. STATISTICAL FACTS OF THE MEASUREMENTS.

	Strategies	*Methods*
Minimum	2.00	1.00
Median	2.00	2.00
Average	3.29	3.37
Maximum	13.00	25.00

As mentioned before, we merely considered Strategy patterns containing 2 or more concrete strategies. On the other hand, interfaces with only one method in at least two strategies were included in the results.

As the statistical analysis has revealed, existing Strategy implementations are quite different so that it is hard to come up with fixed threshold values. Hence, we decided to create a staged recommendation model based on the average and median results. Although M1 and M2 (cf. Table II TABLE II. aim at identifying strategies "hidden" in the code in a different way, the same thresholds can be applied since both are based upon the number of strategies and the number of implemented methods per Strategy. Therefore, our model illustrated in Figure 5 assigns one of three levels of usefulness to each detected pattern recommendation as follows:

1. **P**ossible: a pattern is reasonable and it is likely that it could improve the code especially if further extensions are to be expected. A possible place for a Strategy is found in this case if the number of strategies and methods is at least equal to the median of the analysis presented in Table III, i.e., both values are at least 2.
2. **U**seful: a pattern is useful for an analyzed source code if the measurement results are at least 3, which roughly corresponds to the average of the analyses.
3. **R**ecommended: a pattern is definitively recommended when all measurements are over the average, i.e., if they are equal or larger than 4.

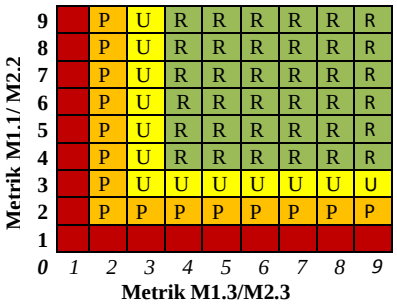

Figure 5 Graphical threshold model for Strategy smells.

Figure 5 illustrates the three levels of usefulness grpahically: Orange for **P**ossible, yellow for **U**seful und green for **R**ecommended.

V. DETECTION EXAMPLES

In this section, we demonstrate how our predicates can be used for automated design smell detection and pattern recommendation. We have analyzed eight open source programs with a total of about 850 thousand Lines of Code (KLOC) in 10,000 classes and found 41 candidates where the Strategy pattern was deemed helpful. Before we present the detailed numbers, we briefly explain the tool we have been developing for this purpose.

 ISBN: 978-1-61208-367-4

A. Detection Tool Preview

In order to evaluate the explained rules, metrics and thresholds automatically, we have implemented a detection tool based on PMD [24]. PMD is a code quality tool using the abstract syntax tree (AST) of Java for identifying code smells. Our tool is using the plugin interface of PMD in order to benefit from the PMD platform and avoid reinventing the wheel. It is able to extract the required data for the application of the rules defined in Table II from there and to finally present recommended patterns together with the measured values and of course the places (i.e. classes) where they should be integrated.

The code of the tool is separated into four components, respectively packages. The first package is collecting the necessary information from the AST. Package two implements the data model for storing the extracted information, while package three processes the data and measures the required metric values. After collecting all necessary information, the fourth package stores and evaluates the metrics as well as the structural information and finally applies the predicates to identify and present the potential pattern.

B. Detecion Results and Examples

As mentioned previously, we have chosen eight well-established open source projects for a first explorative study intended to illustrate the effectiveness of our approach and to help us gain a better understanding of its mechanics. Table IV provides an overview of all discovered Strategy design smells. Execution times were measured on an old 1-core computer with 2 GHz and can hence at least be divided by four on more recent machines. However, in order to provide acceptable times for a proactive recommendation system, applying an incremental analysis seems necessary.

TABLE IV. ANALYSIS RESULT OVERVIEW.

	jEdit	Lucene	Helix	Megamek	jHotdraw	Columba	tyrant	Jaffa
Possible	2	0	2	13	3	2	3	1
Useful	1	4	1	12	2	0	0	0
Recomm.	0	0	0	0	0	0	0	0
KLOC	117	90	35	283	80	91	41	113
Time (h)	1.5	1.1	0.6	18	1.9	3	1.4	3

Interestingly, no clear recommendation for the use of the Strategy pattern has been found, but a total of 45 occasions where the pattern at least appears to be possible. For the moment, we have manually inspected the discovered suggestions and consider them as appropriate. The next section on future work will discuss planned additional evaluations.

In order to illustrate the results more vividly, we have chosen a design smell discovered in the open source tool jEdit (Version 5.1) [29] as an example. The code snippet shown in Figure 6 was extracted from its *TextUtilities* class. It contains two methods (*findWordEnd*/*findWordStart*) with a switch statement in turn containing three cases using the same case condition (i.e. `WHITESPACE`, `SYMBOL` and `WORD_CHAR`) and the same switch parameter (`type`). Due to limited space, code details have been omitted.

```
public static int findWordStart(…) {
    switch(type) {
    case WHITESPACE:
        …
    case WORD_CHAR:
        …
    case SYMBOL:
        …
    } return 0;
}
```

```
public static int findWordEnd…) {
    switch(type) {
    case WHITESPACE:
        …
    case WORD_CHAR:
        …
    case SYMBOL:
        …
    } return line.length();
}
```

Figure 6 Examplary opportunity to use the Strategy pattern in jEdit.

Table Vsummarizes the assessment of the predicate defined in Tables I and II, respectively, for this example.

TABLE V. MEASUREMENT RESULTS FROM TEXTUTILITIES CLASS.

Metric	Value	Metric	Value
M1.1	2	M1.3	3
M1.2	True	M1.4	True

As visible in the Table V, both Boolean metrics (M1.2 and M14) are true and hence fulfill the first requirement for design smell detection. Moreover, M1.3 is equal to the average of 3 as well as M1.1 is equal to the median. According to the model described in Figure 5 , a Strategy pattern can be considered as a useful improvement for this piece of code.

VI. FUTURE WORK

We have planned to improve our prototypical recommender application so that it is able to detect and recommend pattern candidates for design patterns automatically for each Java project a developer is working on in a common integrated development environment. In this context, it is important to find a convincing way to present pattern recommendations to the users. A well-designed user interface that clearly indicates where a pattern could be introduced and which classes should participate in the pattern in what role is probably the key to achieving user acceptance. Another requirement is that it will most likely be necessary to extend the use of thresholds to the size of the code base, i.e. the size of code in the case blocks in case of the Strategy patterns. According to informal feedback of colleagues, it seems to be the case that developers are very sensitive when patterns create a relatively large overhead compared to the actual functionality they "contain". Moreover, we will continue working on recommendation predicates for further GoF patterns. We currently assume that we will be able to develop detection possibilities for most of the GoF patterns. Only prospective Adapter and Interpreter patterns cannot be suggested based on existing code since they require a cognitive decision of the developer to integrate a novel piece of code into an existing system. It is also impossible to recommend opportunities for the Composite pattern since a domain analysis has to detect the part-whole hierarchies between objects to be represented by this pattern.

Another important open topic is of course the evaluation and the fine tuning of the developed predicates and thresholds that we use for pattern candidate detection. We plan to analyze further open source projects in order to see whether

 ISBN: 978-1-61208-367-4

our tool is able to recommend appropriate pattern opportunities. In order to increase the validity of the results, we want to give the identified recommendations to various professional developers (or even the authors of the investigated systems themselves) in order to get an independent feedback whether they consider the discovered candidates as useful. Another validation we plan to tackle soon is scanning the repositories of open source projects for concrete refactorings that have integrated design patterns into their code base. Using our tool on the version before such a refactoring obviously should yield a recommendation for the appropriate pattern and further demonstrate the effectiveness of our approach.

VII. CONCLUSION

In this paper, we have presented a prototype of a design pattern recommendation tool that can be directly integrated into common development environments. It comprises the following three contributions. First, we have explained how opportunities for the use of design patterns can be identified through analyzing the AST of Java programs based on so-called detection predicates. Nevertheless, the presented ideas are not be limited to Java, but should be transferable to other object-oriented languages as well. Second, we have presented an approach on how meaningful thresholds for the metrics used in the detection predicate can be derived from mining existing Strategy implementations in open source projects.

Third, in order to demonstrate the practical feasibility of our ideas, we have presented concrete predicates for the GoF Strategy pattern as well as a concrete Java implementation for a detection utility and evaluated it on eight open source projects together comprising more than 850 thousand lines of code 10,000 classes. Our tool was able to present numerous meaningful opportunities for the utilization of the pattern. Hence, we are encouraged to continue our work in order to also define predicates for various other GoF patterns and extend our prototype accordingly.

REFERENCES

[1] C. Larman, Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e, Pearson Education India, 2012.

[2] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: Abstraction and reuse of object-oriented design, Springer, 1993.

[3] M. Robillard, R. Walker and T. Zimmermann, "Recommendation Systems for Software Engineering", IEEE Software, Vol. 27, No. 4, pp. 80-86, 2010.

[4] R. Holmes, R. J. Walker and G. C. Murphy, "Strathcona example recommendation tool", ACM SIGSOFT Software Engineering Notes, pp 237-240, 2005.

[5] O. Hummel, W. Janjic and C. Atkinson, "Code Conjurer: Pulling reusable software out of thin air", IEEE Software, Vol. 25, No. 5, pp. 45-52, 2008.

[6] M. Baranski and J. Voss, "Genetic algorithm for pattern detection in NIALM systems", IEEE Intern. Conference on Systems, Man and Cybernetics, , pp. 3362-3468, 2004.

[7] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring", IEEE Transactions on Software Engin., pp. 896-909, 2006.

[8] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection", IEEE International Workshop on Program Comprehension", pp. 94-103, 2003.

[9] E. Van Embend, and L. Moonen, "Java quality assurance by detecting code smells", Working Conference on Reverse Engineering, pp. 97-106, 2002.

[10] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[11] L. Briand, Y. Labiche, and A. Sauve, "Guiding the Application of Design Patterns based on UML Models", Intern. Conf. on Software Maintenance, pp. 234-243, 2006.

[12] K. Beck, et al., "Industrial experience with design patterns". Intern. Conf. on Software Engin., pp. 103 -114, 1996.

[13] Z. Durdik, and R. Reussner, "Approach for architectural design and modelling with documented design decisions", Int. Conf. on Quality of Software Architectures, p.9, 2012.

[14] F. Palma, H. Farzin, Y. Gueheneuc, and N. Moha, "Recommendation system for design patterns in software development", Intern. Workshop on Recommendation Systems for Software Engineering, pp. 49 – 54, 2012.

[15] S. Suresh, M. Naidu, S. A. Kiran, and P. Tathawade, "Design pattern recommendation system: a methodology, data model and algorithms", International Conference on Computational Techniques and Artificial Intelligenc, 2011.

[16] O. Seng, F. Simon, and T. Mohaupt, Code Quality Management, dpunkt Verlag, Heidelberg, 2006

[17] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, "Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability", in Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, Springer, pp. 138 - 145, 2012.

[18] B. Huston, "The effects of design pattern application on metric scores", Journal of Systems and Software, Vol. 58, No. 3, pp. 261-269, 2001.

[19] J. Fabry and T. Mens, "Language-independent detection of object-oriented design patterns", Computer Languages, Systems & Structures, Vol. 30, No. 1, pp. 21-33, 2004.

[20] Y. Guéhéneuc, G. Jean-Yves and S. Houari. "Improving design-pattern identification: a new approach and an exploratory study." Software Quality Journal 18.1, pp. 145 - 174, 2010.

[21] R. Martin, Clean code: a handbook of agile software craftsmanship, Pearson Education, 2008.

[22] K. Beck,. Implementation Patterns. Pearson Education, 2007.

[23] W. Janjic, O. Hummel, M. Schumacher, and C.Atkinson, "An unabridged source code dataset for research in software reuse". Intern. Workshop on Mining Software Repositories, pp. 339 – 342, 2013.

[24] N. Rutar,C. Almazan, and J. Foster. "A comparison of bug finding tools for Java". In 15th International Symposium on Software Reliability Engineering, pp. 245 – 256, 2004.

[25] Christopoulou, A., Giakoumakis, E. A., Zafeiris, V. E., and Soukara, V., "Automated refactoring to the Strategy design pattern". Information and Software Technology, pp. 1202-1214, 2012.

[26] S. Burger and O. Hummel, "Über die Auswirkungen von Refactoring auf Softwaremetriken" (in German), GI-Fachtagung Softwaretechnik, pp. 113–126, 2012.

[27] http://www.jhotdraw.org/, JHotDraw as Open-Source Project, Accessed on 02/11/2014

[28] T. Tourw'e und T. Mens, „Identifying refactoring opportunities using logic meta programming," in Software Maintenance and Reengineering, 2003.

[29] http://www.jedit.org/, Progammer's Text Editor, Accessed on 02/11/2014

 ISBN: 978-1-61208-367-4