Analyzing Source Code for Automated Design Pattern Recommendation

Name
Affiliation
Affiliation 2
Country
someone@affiliation.org

Another Name
Affiliation
Affiliation2
Country
someone@affiliation.org

ABSTRACT

Mastery of all subtleties of object-oriented programming languages is undoubtedly challenging to achieve. Design patterns have been proposed already some decades ago in order to support software designers and developers in overcoming recurring challenges in the design of object-oriented software systems. However, given that dozens if not hundreds of patterns have emerged so far, it can be assumed that their mastery has become a serious challenge in its own right. In this paper, we describe a proof of concept implementation of a recommendation system that aims to detect opportunities for the Strategy design pattern that developers have missed so far. For this purpose, we have formalized natural language pattern guidelines from the literature and quantified them for static code analysis with data mined from a significant collection of open source systems. Moreover, we present the results from analyzing 25 different open source systems with this prototype as it discovered more than 200 candidates for implementing the Strategy pattern. Finally, we sketch how we are currently extending this work to other patterns.

CCS CONCEPTS

• Software and its engineering -> Software notations and tools -> General programming languages -> Language features -> Patterns

KEYWORDS

Design Patterns, Recommendation System, Code Analysis

1 INTRODUCTION

Design Patterns [1] are both a blessing and a curse – a blessing because patterns are improving source code in several aspects like flexibility, readability etc., and a curse since the growing number of available patterns [6] has made the recognition of opportunities for using them a serious challenge in its own right. Even worse, a potentially incorrect use of patterns may even have a negative effect on source code quality [20]. Hence it is crucial that developers applying patterns not only have a lot of experience in dealing with them, but also a thorough understanding of application structure and behavior. The increasing complexity of software development projects and a steadily growing demand for development personnel in today's digitized world, are just two factors that illustrate how helpful tool support could become in this context, especially to support developers inexperienced with the subtleties of object-oriented programming in building more sustainable solutions.

To date, however, as we will describe in more detail in the following section, there are merely few publications dealing with the challenge of supporting developers to spot opportunities for the use of patterns. Consequently, this still remains a task that is largely based on the experience of architects and developers working on a system under development. Based on the definition of the refactoring community [11] where a "code smell" describes a problem in the code on the class level (such as an overly long or misplaced method), we call a problem with the design (i.e. typically impacting more than one class) that e.g. might hinder future extensibility of a system a "design smell".

Inspired by the recent trend towards automating support for various activities during software development with so-called recommendation systems [7], we have started developing a tool that aims on detecting such design smells in Java-based software systems in order to identify beneficial opportunities for the use of patterns. This prototype integrates seamlessly into common development environments (as a PMD [3] plugin) and points developers directly to those hotspots in their code where the use of a design pattern [1] – namely the Strategy pattern – would improve code quality.

We describe in this paper, how we derived the rules and thresholds we use in PMD in order to identify the "lack of Strategy pattern" design smell, by mining a large source code collection for pattern instances. Moreover, in order to get a first impression of the capabilities of our tool, we present results of examining 25 open source projects (see Appendix A) for lack of Strategy design smells. The relatively large number of over 200 smells found during this evaluation is clearly a striking argument to investigate pattern recommendation approaches further.

The remainder of this paper is structured as follows, first we briefly discuss the state of the art in improving source code quality without and with design patterns in section 2, before we go into technical foundations for our approach in section 3. The approach itself is presented in section 4. The following section 5 explains the evaluation we have run on open source projects and discusses its results. The subsequent section 6 discusses ongoing work and ideas how future work could extend our approach. The last section summarizes all results and concludes our contribution.

2 STATE OF THE ART

In the following two subsections we briefly discuss the relevant state of the art of measuring and improving internal software quality with metrics, refactoring and design patterns.

2.1 Software Quality & Refactoring

The (internal) quality of source code has been in the focus of software engineering researchers for decades and due to the large number of related publications can be only discussed very briefly in this work. The approach with the longest history in this area is probably the use of software metrics as already proposed back in the 1970s [14]. Software metrics were then considered as an easy measure for capturing the complexity of a piece of software. However, since then, it has become apparent that software is not that simple and a non-selective optimization of software metrics is normally not expedient [24], [13], [29], [28]. Moreover, even formerly promising code quality models such as the Maintainability Index [22] have been found not being useful at all. Still, there is little empirical evidence that more recent software quality models utilizing a combination of metrics will actually help in improving source code, although there seems to be some anecdotal evidence that such models can give a helpful indication on the overall complexity of a software system [15].

Two approaches that keep experiencing an increasing usage in practice though, are code analysis tools like FindBugs [10] or PMD [3] and Refactoring [11] as popularized by the agile community. There are various anecdotal reports describing helpful feedback from both of them (such as [16], [17]), and since both focus on detecting and fixing common bad coding practices, it seems logical that their use can improve code quality. Moreover, there are a number of research works that aim on detecting the code smells that Fowler discussed in his seminal book on refactoring [11]. A recent survey of Dallal [30] shows an increasing number of works that have been aiming on automating this task in the last ten years. However, they are all focused on the rather fine-grained refactorings as popularized by Fowler. The work of Tsantalis et al. [2] may be considered as one example where the authors presented a tool that suggests methods to be moved to other classes in order to increase cohesion and reduce coupling in a system.

2.2 Design Patterns

Design Patterns are another well-known approach in software engineering for improving the internal quality of software systems. As opposed to the rather fine-grained refactorings discussed before, they are typically more complex, as using them normally impacts a number of classes and therewith the design of a system (hence the name). Their idea of collecting proven solutions for common design problems has been around for more than twenty years: the well-known pattern catalog by the Gang of Four (GoF: Gamma, Helm, Johnson, Vlissides) [1], for example, was published in 1994 already. A pattern description contained there consists of four essential elements, namely:

1. **Pattern Name**: a clear and concise name for the pattern
2. **Problem Description**: gives the context in which a pattern is considered useful
3. **Solution**: an abstract description how a pattern can help solving the problem, that nevertheless needs to be adapted for each concrete case
4. **Consequences**: explains how the use of a pattern influences the system under development

Since the publication of the GoF book numerous works on various aspects of patterns have been published. For example, researchers investigated the dissemination of patterns in existing systems [18], [19]. However, as it is not a trivial undertaking to recognize patterns in code, these works needed to develop heuristics (such as identifying pattern names in commits to version control repositories) for identifying the patterns in the first place. Unfortunately, these heuristics remained rather fuzzy, so that the results of most of these works remain vague and hard to compare with each other. One more sophisticated approach based on representing code and patterns to be recognized as a graph and using a graph similarity algorithm has been presented by Tsantalis et al. [31]. The authors claim to reach a very high precision with it, even in recognizing patterns that are not fully implemented by the book. Tools for recognizing patterns in code include e.g. "Pattern4" [23] or the pattern detection tool developed by Tsantalis et al, which is available from his website[1].

Given all the praise that patterns have received in recent decades, one would expect a significant number of studies investigating the effects of patterns on source code quality. However, beyond the above-mentioned discussion of consequences by the GoF already, we are merely aware of a handful of mostly partial works in this direction (e.g. [20], [21]). The distillable message from them is threefold: first, it seems that patterns indeed have the potential for improving code quality, but only when they are applied correctly. Second, incorrect implementations of patterns do also happen in practice and make a system design and its associated source code even harder to grasp. This again underlines the necessity of supporting developers in dealing with the correct selection and application of patterns. The third important aspect that has unfortunately only been partially investigated in the literature so far (e.g. by Jafaar et al. [32]), is the question whether the use of patterns always makes sense? As patterns induce overhead (such as additional classes) into a system, the use of a pattern is probably only helpful when a certain degree of complexity is reached. Consequently, the need for using a pattern might in some cases only arise after a system has been extended and might not have been apparent when its initial design was created. Despite these obstacles, the promise of patterns is clear: they are expected to improve the long-term maintainability of a system, especially in terms of extensibility and flexibility.

Consequently, the idea of supporting developers in spotting or avoiding design smells has been around for some time, although

[1] https://users.encs.concordia.ca/~nikolaos/pattern_detection.html

the number of works in this area still remains small. Even worse, most approaches we are aware of, are neither working with code nor with concrete designs of a system, but rather provide abstract guidelines [27] or question catalogs [12] that are intended to support system designers during their work. These approaches obviously still require a lot of manual and with increasing system size likewise a growing cognitive effort, although, as a result, they undoubtedly lead to a better understanding of a system design. On the other hand, however, it is at least questionable whether such upfront design considerations fit into today's wide-spread agile development approaches [9] that often intertwine design, code, and even unit test activities.

3 FOUNDATIONS

In order to prepare the reader for understanding our approach on pattern recommendation in the following, we briefly introduce the Strategy pattern and the PMD code analysis tool in the next two subsections. Readers familiar with them can safely continue reading in section 4 without any loss of information.

3.1 Strategy Pattern as Running Example

According to Gamma et al. [1], the Strategy pattern is a behavioral pattern that "define(s) a family of algorithms, encapsulate(s) each one, and make(s) them interchangeable." One prominent example for its application in Java is the use of *LayoutManagers* in Swing UIs. Instead of having lots of switch statements in all UI containers that need to arrange the elements they contain, the required behaviors for the arrangement are encapsulated externally in different *LayoutManager* classes. These classes are all implementing a collective interface so that a programmer can select a desired concrete layout strategy for a UI container or even develop a new one.

It is important to mention that from a structural perspective, the class diagram of the Strategy pattern is identical with the State pattern in the sense that the various strategies in the former correspond to the states of the latter. The main difference between the two is that state implementations have control over state changes themselves (i.e. a state determines the next state in reply to an event), whereas changes of the applied strategy are made by the client. This is an important aspect that we need to keep in mind for our prototype and that lays out a clear path for supporting the State pattern in the future as well.

3.2 PMD

Without any loss of generality, we have focused our proof of concept on the Java programming language as it is a widespread and well-known object-oriented language with a large body of code analytics tools and numerous open-source projects available online for experimentation. Nevertheless, the presented approach and our preliminary results should be transferable to other object-oriented languages as well as to other programming paradigms provided that suitable patterns are available there.

One prominent example of a code analytics tool is PMD [3], which is normally used to detect common bad programming practices (e.g. dead or cloned code) and non-optimal code structures (such as complex conditionals) on a statement level. It is able to analyze source code in several different programming languages like Java or C#. All its analyses are using the abstract syntax tree (AST) of the underlying language to find the mentioned problematic statements. Fortunately, it is relatively straightforward to extend PMD's functionality with new detection rules and features through the use of two extension points provided. The first one allows to describe how to traverse AST nodes for the intended analysis based on XPath queries. The second option allows to formulate rules in Java code and hence opens a wide range of functionality and direct access to other PMD interfaces.

4 APPROACH

As explained before, our working hypothesis is that it is feasible to recognize the existence and even the exact position of design smells in object-oriented software systems with the help of a dexterous static analysis of that system. Our central idea is to formalize and where necessary extend the guidelines that well-known pattern catalogs are providing their readers into detection rules that can be applied by a tool (based upon PMD). In order to make these rules practically applicable in PMD, we had to perform two steps, namely first the "translation" from often relative vague natural language into precise source code elements that can be found in the code's abstract syntax tree, and, second, we had to analyze existing pattern implementations for thresholds from which it makes sense to use a pattern. The latter is needed as a response to the overhead induced into a system by a pattern so that a developer is able to make a better-informed tradeoff whether it makes sense to actually implement a suggested pattern or not (as discussed in section 2.2).

4.1 Detection Rule Derivation

As indicated before, we have chosen the Strategy pattern to evaluate the feasibility of this approach as it is sufficiently complex while at the same time relatively straightforward to understand. Based on recommendations from the literature and our analysis of numerous Strategy examples contained there, we have distilled the following detection rule for Strategy opportunities in natural language:

There is a significant switch statement in various methods of a class, which always uses the same variable in its conditions and has the same cases. Moreover, this variable is never changed in any of the decision branches.

It is important to emphasize the last sentence of this definition as we see it as the key for a future differentiation of State and Strategy pattern opportunities. (cf. section 3.1)

The following table summarizes the formal detection rules we have derived from our above natural language version of the problem description for the Strategy pattern. The first column names the AST attributes we are using, the second column gives a brief description of it, while the last column lists the required outcome in order for the rule to fire. It is important to understand that, as implied by the natural language description, the individual rules need to be concatenated with a logical AND and have to fire all in one class in order to signal a successful smell detection.

Table 1: Derived detection rules for the Strategy pattern.

AST Attribute	Description	Minimal Threshold
No. of cases	a switch block needs a certain number of cases	>=2
No. of appearances	the switch block needs to appear in several methods with identical conditions	>=2
Identical case conditions	all cases in each switch block are identical	True
Equal no. of cases	the number of cases has to be equal	True
Single class	all methods with that switch have are in the same class	True
Same header attribute	all switches use the same attribute in their condition	True
Same attribute within the body	the value of this attribute must not change in the case clauses	True

Moreover, it is important to understand the relation of the first two AST attributes from Table 1 with the values we analyzed in the following subsection. While Table 1 describes elements from a naïve and thus "smelly" implementation (i.e. the bad case) we want to analyze for pattern opportunities, the following subsection considers good implementations of the Strategy pattern. Cases in switch blocks of the former would be implemented as a concrete strategy in the latter and the number of appearances in Table 1 corresponds with the number of methods implemented per strategy in Figure 1 in the following section.

4.2 Threshold Derivation

In order to better quantify the usefulness of a pattern recommendation in a given system and avoid "over-engineering" a system with too many patterns, we believe it is necessary for a human inspecting the delivered result to get a good sense of the "intensity" of the detected smells. We decided to establish an "intensity scale" by collecting and analyzing the values from existing applications of the Strategy pattern in the wild, coming from an established collection of open-source systems [4]. This should give us a better understanding when it makes sense to actually use a pattern in practice. Or in other words, this is how we collected the statistics to quantify the threshold values for the first two detection rules presented in Table 1.

As mentioned before, finding existing pattern implementation in a given codebase is a significant challenge in its own right. Hence, we used the following heuristics for the identification of Strategy pattern instances: we executed a search for all Java classes in the codebase that ended with "…Strategy" and originated from projects under version control (thus excluding those files that were individually crawled from the open web and thus probably were not part of a well-designed project). This led to a total of 286 potential Strategy implementations. We manually inspected them and found 53 candidates (out of 35 different projects) that actually fulfill the Strategy pattern definition by Gamma et al. In order to avoid biasing our baseline, we added the additional constraint that a maximum of two pattern implementations were considered per project.

All candidates were then manually inspected a second time, in order to count the values for deriving the two necessary thresholds for the Strategy detection rule. As apparent from Table 1, we were interested in the number of different strategies implemented per pattern and the number of methods per strategy. To reiterate once more, these two values would correspond to the number of switch blocks resp. the number of cases in a naïve, i.e. a smelly implementation that the tool needs to detect later. The histogram of these values is shown in the following figure 1.

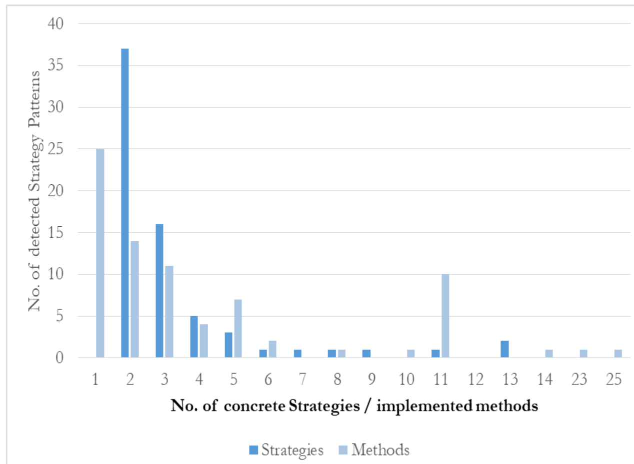

Figure 1: Distribution of number of strategies and implemented methods per strategy in the analyzed pattern instances.

The subsequent table 2 summarizes the statistical key figures for the above distributions:

Table 2: Statistical figures for strategy pattern

	No. of Strategies	No. of Methods
Min.	2.0	1.0
1. Quartile	2.0	1.0
Median	2.0	2.0
Average	3.3	3.4
3. Quartile	3.0	4.0
Max.	13.0	25.0

As a working hypothesis, we defined that we would return pattern suggestions on three different recommendation levels that are allocated to the statistical measurements as follows:

1. Possible: ≥ Median
2. Useful: ≥ Average
3. Recommended: ≥ 3rd Quartile

This definition led to the following matrix of recommendation levels and their corresponding boundary values:

Table 3: Boundary values for detection of Strategy pattern candidates.

No. of Strategies	No. of Methods	Rec. Level
≥ 3	≥ 2	Possible
≥ 3	≥ 3	Useful
≥ 4	≥ 4	Recommended

Consider the *LayoutManager* Strategy implementation from Java Swing mentioned in Section 3: it contains five strategies and more than ten methods and hence would fall into the "Recommended" category.

4.3 Prototypical Implementation

We have prototypically implemented the model presented above as an extension for PMD, based on PMD's Java interface to collect the required information from the analyzed source codes. As discussed before, for a Strategy opportunity detection, the detection rule needs information about switch statements and their contents from the AST. Each required information is stored in a specific AST-node type together with some additional metadata. PMD uses the visitor pattern for walking through the AST of each class. If PMD finds one of the desired node types, it will make a callback to our prototype. This will then extract all required information and metadata from the node and write it into a database for later execution of the next step. There, it is almost trivial for the prototype to iterate over the database and to identify those classes that trigger the detection rule.

5 EVALUATION RESULTS

In this section, we present the results we yielded from executing the Strategy smell detection with our tool prototype. The evaluation was actually twofold. First, we analyzed 25 well-known Java open source projects (as listed in Appendix A), comprising more than 3.6 million lines of code, with our tool, and, second, we asked 9 experienced programmers (mainly industrial developers) to evaluate the quality of two concrete Strategy recommendations delivered by our tool. The first investigation was giving a broad overview of how well opportunities for a Strategy pattern are recognized in various open source systems. The evaluation of these exemplary recommendations by experienced developers gave us a first indication for the quality of the delivered results.

The automated analysis of the 25 open source projects (see Appendix A) delivered a total of 211 design smells that could be rectified with the Strategy Pattern as summarized in the following table.

Table 3: Detected Strategy smells in 25 open source projects.

Rec. Level	No. of Candidates	%
Possible	32	15%
Useful	156	74%
Recommended	23	11%
Overall	**211**	**100%**

This makes an average of almost 9 candidates for using the Strategy pattern in each of the 25 open source projects taken into account. The detailed distribution of candidates on the projects is also listed in Appendix A.

For our evaluation of the results, we have picked two medium-sized examples (found in ArgoUML and Apache Lucene) out of the result set and extracted the relevant code sections, i.e. the methods containing the switch statements our tool considered as a smell. The nine participants of our survey had a total of 54 years of Java experience and all claimed to have a solid knowledge of design patterns. 6 participants were industrial developers, 3 were scientists and 1 was a graduate student. The following table summarizes their responses.

Table 4: Evaluation of Strategy Recommendations.

Candidate	Helpful	Very Helpful	Don't Know	Level
Strategy 1	6	1	2	Possible
Strategy 2	4	5	0	Useful

Other answering options were "not helpful" and "helpful, but too much overhead", which were not chosen by anyone and hence are not listed in the table for the sake of space.

5.1 Discussion

Clearly, the evaluation results presented in the previous section are still in an intermediate state and thus faced with some threads to validity. At the time being, this is mostly related with the internal validity, as it is unclear whether the understanding of a design smell that we have distilled from the literature, extended, formalized, and implemented is completely sound and correct, although the participants of the small pilot study were rather fond with two exemplary recommendations. Hence, it is certainly still justified to take the relatively large number of discovered design pattern candidates with a grain of salt.

On the other hand, as we have explained the detection rule for this smell in great detail, it is replicable for every experienced Java developer resp. researcher and even if one would tend to use different thresholds, the main result of this work will remain significant: given the large amount of Strategy design smells discovered in current open source systems, there seems to be a clear necessity for supporting developers with pattern recommendations. However, as open source systems are merely a specific population of software systems that might differentiate from industrial closed-source systems in various aspects (such as developer experience or development process etc.), it is not clear how far the results can be generalized to other types of systems. This is a thread to external validity that we plan to counter by analyzing a significant closed-source enterprise system of an industrial partner (an international insurance company).

5.2 Limitations

While working out the detection rules for further patterns in our ongoing work, we have found that our approach is probably limited in the sense that it is not possible to detect smells for all 23 GoF patterns. The reason for this is that, as far as we can tell with our current understanding, the use of some patterns requires a conscious decision of a human, i.e. the responsible developer. Take for example the Adapter pattern that is helpful when it comes to interface mismatches during the integration of foreign classes or components into an existing system. As soon as a developer tries to integrate a mismatching component into a given system, he will receive a compiler error so that no smell detection is necessary anymore.

Beyond the Adapter pattern our considerations yielded a similar result for the Composite and the Interpreter pattern. The former is intended to better structure part-whole hierarchies in code, but again requires a conscious human decision that such a hierarchy is useful in a system. The situation is again similar for the Interpreter pattern. It offers a template for the construction of a simple language interpretation kit so that users of a system can control it through a simple domain specific language (DSL). Once more, the decision for the use of a DSL within a system is a conscious one and probably made some time before coding begins at all.

Furthermore, our approach can be considered somewhat limited in the sense that it comes rather late in the development lifecycle. Optimally, as the name suggests, design patterns should be recommended during the design phase and not only after the coding phase as making changes to a system usually becomes more expensive the later a change occurs in the development process.

However, we have decided to derive our recommendations from code for the following practical reasons: First and foremost, we believe that our approach is still helpful (as underlined by the large number of pattern recommendations detected in our pilot study) since modern iterative development approaches often amalgamate design, coding and even refactoring and unit testing with each other (see e.g. [11]). Hence, it makes perfect sense to support the developer during these activities with a tool that is able to recommend the application of design patterns. In relation to the previous discussion of criticality of the pattern recommendation, it should be noted that only incremental changes to a codebase might make a naïve implementation so complex that it becomes worthwhile to refactor to a pattern-based variant.

The second important aspect is, that only code gives a holistic and comprehensive view on a system while e.g. a single UML diagram is usually not able to provide this at a glance, since diagrams often abstract details away. Thus, a UML-based design typically requires various perspectives on a system that must be kept in sync with a lot of effort (cf. e.g. [8]), which is rarely done with the necessary rigor in practice.

Finally, there is simply a lack of freely accessible and machine processable software design documents, which we could have used for this research. On the other hand, there are large amounts of source code available in the repositories of e.g. GitHub and other open source hosters that can be used for rule derivation and experimentation in a straightforward manner.

Another clear advantage of the presented approach is the fact that we can rely on various proven code analysis tools such as PMD [3] in order to implement it and hence we accepted that our tool can be only used during programming and not already when design activities are taking place.

6 ONGOING & FUTURE WORK

We have been working on deriving and implementing detection rules for further GOF patterns (namely State, Builder, Façade, Decorator, and Mediator) and will report on the outcome of this effort at a different occasion, once we have interpreted the results.

Moreover, as discussed before, we feel it is important to challenge our potentially biased interpretation of the detection rules with the opinions of experienced software developers, in order to fine-tune them as well as the recommendation levels we have derived so far. Based on the experience gained from our preliminary study, we have prepared and executed an online survey with 52 professional software developers and exemplary recom–mendations for all of the above mentioned patterns. We are currently in the process of interpreting these results as well and will also report on them at another occasion.

As the results with our tool have been promising so far, we believe it may be worthwhile to extend our work in various directions in the near future. First of all, even though we have already targeted further GoF patterns, there are 17 other GoF patterns left that can be analyzed to find out whether it is possible to automatically detect usage recommendations for them (minus the three for which we do not believe it is possible). Moreover, there is an enormous number of further design patterns (as e.g. listed in the Pattern Almanach [6]) that can potentially be covered by our approach. And, furthermore, it might also make sense to investigate whether smells for even larger-grained patterns such as those listed in Fowler's well-known Enterprise Pattern book [25] are also be detectable with the help of static code analysis.

7 CONCLUSION

Driven by the complexity of an ever-growing catalog of design patterns in software development, in this paper we have described a first proof of concept, demonstrating that the recognition of "design smells" in object-oriented software systems is possible with static code analysis techniques. We defined a design smell as a hotspot in source code that would benefit from the use of a design pattern (although the term can be certainly used in a more general sense as well). As a target for our feasibility study we selected the Strategy design pattern as defined by the Gang of Four [1] and derived formal detection rules as well as a set of recommendation levels for it by manually analyzing 53 Strategy pattern implementations retrieved from an established source code collection [4].

Moreover, these detection rules were implemented in a prototype based upon the PMD code analyzer and applied to a set of 25 well-known Java open-source projects in order to evaluate the quality of results. In total, we have found over 200 design smells that could be remedied by the use of the Strategy pattern. Moreover, in order to get a more neutral view on the quality of these recommendations we have carried out a small pilot study with two selected Strategy pattern recommendations and got a positive feedback for them from 9 experienced Java programmers. Encouraged from these results, we are currently working on analyzing the results yielded from developing detection rules for further pattern opportunities and on interpreting results from a larger survey with more than 50 developers that have given feedback for additional pattern opportunities discovered in our test set. As far as we can tell by now, the results justify to invest further effort into our approach in the future.

APPENDIX A

The following list contains the analyzed open source projects and the number of found Strategy recommendations in the 4th column:

Project	Analyz. Version	Download Link	No. Rec.
ArgoUML	0.34	http://argouml-downloads.tigris.org/argouml-0.34	3
Columba	1.4	http://sourceforge.net/projects/columba	3
JEdit	5.2	http://sourceforge.net/projects/jedit	6
Apache Lucene	4.10.3	http://lucene.apache.org	32
JHotDraw	5.6	http://sourceforge.net/projects/jhotdraw	13
Apache Ant	1.9.4	https://ant.apache.org	6
Apache Wicket	6.18.0	http://wicket.apache.org	1
Ganttproject	2-6-1-r1499	http://sourceforge.net/projects/ganttproject	1
Jrefactory	2.9.19	http://sourceforge.net/projects/jrefactory	3
OpenHab	1.6	http://sourceforge.net/projects/openhab	16
Freedomotic	5.5.0	http://sourceforge.net/projects/freedomotic	0
Jfreechart	1.0.19+	http://sourceforge.net/projects/jfreechart	1
Junit	r4.12	https://github.com/junit-team/junit	0
Recoder	0.97	http://sourceforge.net/projects/recoder	7
Jenkins	1.598	https://github.com/jenkinsci	0
Wind	1.0.1	http://sourceforge.net/projects/wind	41
Derby	10.11.11	http://db.apache.org/derby	31
Elasticsearch	1.4.4	https://github.com/elastic/elasticsearch	4
Freemind	1.0.1	http://sourceforge.net/projects/freemind/	1
Hibernate	4.5.2	http://sourceforge.net/projects/hibernate	2
Jabref	2.10	http://sourceforge.net/projects/jabref	0
Megamek	0.40.1	http://sourceforge.net/projects/megamek/	34
Mina	2.0.9	https://mina.apache.org	0
spring-core	4.1.5	http://sourceforge.net/projects/springframework	3
Triplea	1.8.0.5	http://sourceforge.net/projects/triplea	3

REFERENCES

[1] Vlissides, J., Helm, R., Johnson, R. and Gamma, E., 1995. Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley.
[2] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. T. (2006). Design pattern detection using similarity scoring. IEEE transactions on software engineering, 32(11).
[3] Rutar, Almazan, Foster (2004), "A Comparison of Bug Finding Tools for Java". ISSRE '04 Proceedings of the 15th International Symposium on Software Reliability Engineering, IEEE, DOI: 10.1109/ISSRE.2004.1
[4] Anonymized
[5] Ray Toal, Loyola Marymount University http://cs.lmu.edu/~ray/images/strategy.gif, visited 21.05.2017
[6] Rising, L., 2000. The Pattern Almanac. Addison-Wesley.
[7] Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, Th. (Eds.), 2014. Recommendation Systems in Software Engineering.
[8] Atkinson, C, Bayer, J., Bunse, C. et al., 2001. Component-based Product-line Engineering with UML, Addison-Wesley.
[9] Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R. and Kern, J., 2001. Manifesto for agile software development.
[10] Louridas, P., 2006. Static code analysis. IEEE Software, 23(4).
[11] Fowler, M., 1999. Refactoring: improving the design of existing code. Addison-Wesley Professional.
[12] Durdik, Z., Reussner, R., 2012. Position paper: approach for architectural design and modelling with documented design decisions. Proc. of the 8th Int. ACM SIGSOFT Conference on Quality of Software Architectures.
[13] Anonymized
[14] Halstead, M.H., 1977. Elements of software science. Elsevier.
[15] Rauch, N., Kuhn, E. and Friedrich, H., 2008. Index-based process and software quality control in agile development projects. CompArch2008, pp.1-4.
[16] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J. and Zhou, Y., 2007, October. Using findbugs on production software. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pp. 805-806. ACM.
[17] Rutar, N., Almazan, C.B. and Foster, J.S., 2004. A comparison of bug finding tools for Java. In IEEE Software Reliability Engineering, 15th International Symposium.
[18] Hahsler, M. (2003). A quantitative study of the application of design patterns in java. WU Vienna University of Economics and Business: Institut für Informationsverarbeitung und Informationswirtschaft.
[19] Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., & Di Penta, M. (2007). An empirical study on the evolution of design patterns. Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.
[20] Hegedűs, P., Bán, D., Ferenc, R., & Gyimóthy, T., 2012. Myth or reality? analyzing the effect of design patterns on software maintainability. In Computer Applications for Software Engineering, Disaster Recovery, and Business Continuit, pp. 138-145
[21] Khomh, F., & Guéhéneuc, Y.-G., 2008. Do design patterns impact software quality positively? 12th European Conference on Software Maintenance and Reengineering, pp. 274-278
[22] Coleman, D., Ash, D., Lowther, B. and Oman, P., 1994. Using metrics to evaluate software system maintainability. Computer, 27(8), pp.44-49.
[23] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S.T., 2006. Design pattern detection using similarity scoring. IEEE transactions on software engineering, 32(11).
[24] Jones, C., 1994. Software metrics: good, bad and missing. Computer, 27(9).
[25] Fowler, M., 2002. Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc..
[27] Suresh, S.S., Naidu, M.M., Kiran, S.A. and Tathawade, P., 2011. Design pattern recommendation system: A methodology data model and algorithms. ICCTAI'2011.
[28] Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open source software development. Information Systems Journal, 12(1), 43-60.
[29] Kaur, K., Minhas, K., Mehan, N., & Kakkar, N. (2009). Static and Dynamic Complexity Analysis of Software Metrics. World Academy of Science, Engineering and Technology, 56, 2009
[30] Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information and Software Technology, 58, pp. 231-249.
[31] Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering, 35(3).
[32] Jafaar, F., Geheneuc, Y., Hamel, S, Khomh, F., Zulkernin, M. (2015). Evaluating the Impact of Design Pattern and Anti-pattern Dependencies on Faults and Changes, Springer Empirical Software Engineering.