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 the idiosyncrasies of object-oriented programming languages is undoubtedly challenging to achieve. In order to improve this situation, various design patterns aim on supporting software designers and developers in overcoming recurring design challenges. 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 propose a recommendation system that aims on detecting those "design smells" in object-oriented systems that can be mitigated by the application of well-known design patterns. We describe a proof of concept for which we mined a significant collection of source code in order to derive a detection rule for identifying a smell that can be rectified by the Strategy pattern. Moreover, we present results from analyzing 25 open source systems with this rule, indicating that developers indeed can benefit from tool support in recognizing design smells, as more than 200 candidates were detected. Finally, we briefly 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 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 have a negative effect on source code quality. 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. Given 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.

To date, however, as we will describe in more detail in the following section, there are merely very 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.

Inspired by the recent trend towards automating support for all kinds of activities during software development with so-called recommendation systems [7], we have started developing a suitable tool that it integrates seamlessly into common development environments (as a PMD [3] plugin). Its goal is to point developers directly to those hotspots in their code where the use of a design pattern would remove a design smell and thus improve code quality. Admittedly, identifying design smells already in the design of a system would be the better alternative, however, due to various pragmatic reasons (such as the better availability of open-source compared to "open designs") we have decided to investigate the potential of a pattern recommendation system that uses source code for the time being.

The resulting prototype of the tool we describe as a proof of concept in this work, is delivering a list of (Java) code sections that can be improved by the application of one design pattern (the Strategy Pattern [1]). We explain in this paper how we derived the rules we use in PMD in order to identify these the opportunity for this pattern by mining a large source code collections for instances of it. We illustrate how we used these instances to get a thorough understanding for how pattern opportunities could be detected and when it actually makes sense to implement the pattern from a code complexity perspective. Moreover, in order to get a first impression of the capabilities of such a tool, we present results of examining 25 open source projects (such as Apache Lucene or Elasticsearch) for Strategy pattern opportunities. The relative large number of over 200 pattern opportunities found during this evaluation is another 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 measuring and improving source code quality in section 2, before we go into some 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 discusses ongoing and ideas how future work could use or extend our approach. The last section, summarizes all results and concludes our contribution.

2 STATE OF THE ART

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 at this point. The longest history of approaches in this area is probably exhibited by software metrics that were proposed already back in the 1970s [14] and 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]. Moreover, even formerly promising code quality models such as the Maintainability Index [22] have been found not being useful at all. Unfortunately, there is still little empirical evidence that more recent software quality models utilizing a combination of metrics will actually help in improving source code. At the very least there seems to be some anecdotal evidence that such models can give helpful indication on the overall complexity of a software system [15], nevertheless it remains unclear, however, how a bunch of metrics aggregated to a single value could be used to derive concrete recommendations how and where to improve a system under observation.

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.

It is interesting to mention a recent work [2] that has contrasted common Refactorings with software metrics and has come to the conclusion that Refactoring leads to deteriorating metric values in roughly 50 percent of the (admittedly not too many) cases that were investigated in this study. This can be seen as another indicator that a mere mechanical optimization of metric values may be unrewarding in terms of code quality in practice.

2.1 Design Patterns

Design Patterns are another well-known approach in software engineering targeting more large-grained “design smells” (i.e. bad practices to solve common problems) in the design of software systems. Their idea of collecting proven solutions for common design problems has been around for more than twenty years. For example, the well-known pattern catalog by the Gang of Four (GoF: Gamma, Helm, Johnson, Vlissides) [1] was published in 1994 already. A pattern description contained there usually consists of four essential elements, namely:

1. **Pattern Name**: a clear and concise name for the pattern
2. **Problem Description**: describes the context in which a pattern is considered useful
3. **Solution**: an abstract description how a pattern can help solving the problem that 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 other 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 existing 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 these works also remain vague and hard to compare with each other. Even worse, at the time being, there is only one tool available (called “Pattern4” [23]) that can be used for identifying existing patterns in arbitrary source code.

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, as appearances are often deceitful, we are merely aware of a handful of mostly partial works in this direction (e.g. [20], [21]). The distillable message from them is twofold: 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 harder to grasp. This again underlines the necessity of supporting developers in dealing with the selection and application of patterns.

Consequently, the idea of supporting developers in spotting or avoiding design smells has been around for some time, nevertheless the number of works in this area also remains small so far. 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 effort; as a result, on the one hand, they undoubtedly lead to a better understanding of a system design, however, on the other hand, it is at least questionable whether such upfront design considerations fit into current agile code- or test-driven development approaches and how they can actually be derived and applied for large and complex software systems at all, where this will probably require an immense cognitive effort.

3 FOUNDATIONS

In order to prepare the reader for understanding our approach 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 if or switch statements in all UI containers that need to arrange the elements they contain, the various behaviors are encapsulated externally in so-called LayoutManager classes that are all implementing a collective interface so that a programmer can select the desired layout strategy for a UI container.

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 state of the latter. The main difference between the two is that the state implementations have control over state changes, while changes of the applied strategy are made by the client. We will discuss this in more detail in section 4 when we explain the detection rule for the Strategy pattern and how it could be extended to also recognize State pattern opportunities.

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 for experimentation. Nevertheless, the presented approach and our preliminary results should be transferable to other object-oriented languages as well as to other 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 (i.e. dead or cloned code) and non-optimal code structures (such as complex conditionals) on a statement level. It is able to analyze source code from 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

We are convinced that it is feasible to recognize the exact position of design smells in an object-oriented software system 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 for the time being). 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 abstract syntax tree of the code and, second, we had to analyze existing pattern implementations for thresholds from which it makes sense to use a pattern, given the fact that usually there is some overhead induced into a piece of code when a pattern is used.

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 for a proof of concept while at the same time relatively straightforward to understand. Moreover, its structural similarity with the State pattern is a welcome challenge for a proof of concept in order to find out whether such an approach is also able to detect such subtle differences between pattern opportunities.

Based on this working assumption, we have distilled the following detection rule for Strategy opportunity 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 not changed in every decision branch.

It is important to emphasize the last sentence of this definition as it is the key to distinguish State and Strategy pattern opportunities. As mentioned before, both are structurally identical, but semantically the difference comes from the fact that in a State implementation the state transitions are derived from the state model and encoded in the "then clause" of the cases themselves. The concrete strategy to be used in a Strategy implementation is always chosen from the outside (i.e. via some kind of setter method) and thus never changed in the conditionals. This small difference can be detected in the AST so that we first have formulated the Strategy detection rule in the following table and directly created a variant for State that requires the mentioned changes of the switch variable. The first column in the table names the attributes we are using, the second column gives a brief description of it, while the last column list the expected outcome in order for the rule to fire. It should be obvious that the sub-rules for each attribute need to be concatenated by an AND in order to signal a successful smell detection.

Table 1: Detection rule for the Strategy pattern.

Detection Attribute	Description	Threshold
No. of cases	a switch needs a certain number of cases	>=2
No. of methods	the switch has to appear in several methods	>=2
Identical case conditions	all cases in each switch need to be identical	True
Equal no. of cases	the number of cases has to be equal as well	True

Single class	all methods with that switch have to be part of 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 clause	True

4.2 Threshold Derivation

Rules based on these abstract descriptions alone would not be sufficient to produce practically usable pattern recommendations, since patterns usually come at the price of introducing overhead into the code. Hence it was necessary to derive meaningful thresholds for the first two attributes in Table 1 to provide a lower limit for the complexity of the smells for which we want to recommend pattern opportunities.

We decided to accomplish that by collecting and analyzing existing applications of patterns in order to gain a better understanding when it makes sense to use a pattern in practice. In other words, we collected statistics to define the threshold values that are required to process the detection rule presented in table 1. For this purpose, we have download a source code collection [4] that is available on the Web and mined the contained Java classes for Strategy implementations.

However, as mentioned before, finding existing patterns in a given codebase is a serious challenge in its own right. Hence, we have used the following heuristic process for this task: 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 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 were actually fulfilling the State pattern definition by Gamma et al. under the additional constraint that a maximum of two pattern implementations were considered per project. In total, the 53 pattern implementations contained 182 strategies.

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. The histogram of these values is shown in the following figure 1.

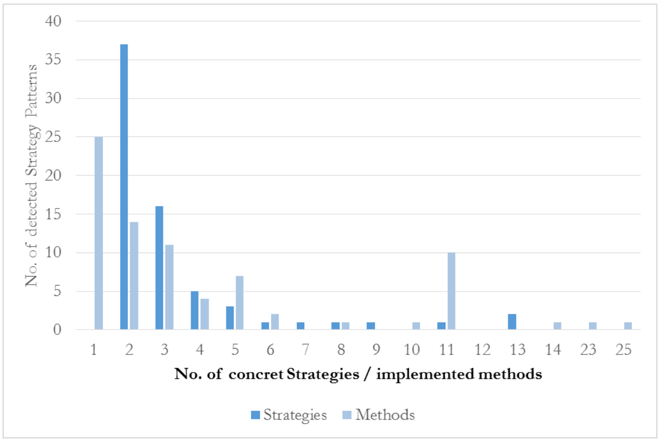

Figure 1: Distribution of number of strategies and contained methods contained in the analyzed pattern implementations.

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

Table 2: Statistical figures for state pattern

	Strategies	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 recommendations on three different urgency levels that are allocated to the statistical measurements as follows:

1. Possible: > 1st quartile
2. Suggestive: > median
3. Recommended: > mean

This definition the led to the following matrix of urgency levels and boundary values:

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

No. of states	No. of methods	Recommendation level
≥ 3	≥ 2	possible
≥ 3	≥ 3	useful
≥ 4	≥ 4	recommended

The Strategy example from Java Swing presented in section 3 contains five strategies and more than ten methods and hence would be clearly attributed with the level "recommended" by this model.

4.3 Prototypical Implementation

We have implemented the model presented above prototypically as an extension for PMD. It uses PMD's Java interface to collect the required information from the analyzed source codes. As an example, for a Strategy opportunity detection, the detection rule needs information about switch statements and their contents.

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. In this, it is almost trivial for the prototype to iterate over the database and to identify those classes that trigger the detection rule.

As mentioned earlier, it is straightforward to derive a similar rule for the State pattern that merely needs to ensure that the header attribute of the switch statement is changed in a number of cases. We have also implemented this, in order to find out whether the prototype would be able to distinguish opportunities for Strategy and State.

5 EVALUATION RESULTS

In this section we present some preliminary results delivered from the Strategy (and the State) smell detection of 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 in a mini survey. The first investigation has the advantage of giving a broad overview of how well opportunities for a Strategy pattern are recognized in various open source systems and gave us a first impression whether developers are able to recognize these opportunities in practice. The second investigation gave us a first impression on the quality of the results delivered by our tool.

The evaluation of the 25 open source projects 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.

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 a Strategy implementation in each of the 25 open source projects taken into account. On note on the detection of State pattern candidates needs to be made here as well. Unfortunately, our tool was not able to spot a single opportunity for the use of this pattern. There could be various reasons for this behavior, e.g. State is typically most often used in embedded systems, which were not part of our collection, though. Moreover, it might be the case that the State pattern is simply better understood by developers so that they already recognized where to use it. Nevertheless, as it could also be a problem with the definition of our detection rule, clarifying this issue should be certainly attacked in future works on this topic.

For our mini evaluation, we have picked two of these results (found in ArgoUML and Apache Lucene) and distilled their code so that they were as easy to grasp as possible while we made sure not to falsify any important aspects. The nine participants of our mini 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 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 are henced faced with some threads to validity. At the time being 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 large number of discovered design pattern candidates with a grain of salt. On the other hand, however, we have explained the detection rule for this smell with in great detail so that it should be replicable for every experienced Java developer and even if one would tend to use different thresholds, the main result of this work will remain the same: given the large amount of Strategy design smells in current open source systems, there seems to be a significant necessity for design pattern recommendation (at least for the open-source community).

5.2 Limitations

While analyzing further patterns in our ongoing work we have found that our approach is limited in the sense that not all 23 GoF patterns can be recommended based on static code analysis. The reason for this is that some patterns simply require 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. No tool would be able to detect a smell on the component to be integrated and merely the developer knows exactly where an Adapter makes sense in order to fit the component into the system under development.

Beyond the Adapter pattern our considerations yielded the same 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 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). Again, the decision for the use of a DSL within a system is a conscious one and probably made for before coding begins at all.

Our approach is also limited in the sense that it comes rather late in the development lifecycle. Design patterns should clearly be recommended during the design phase and not only after the coding phase. However, we have decided to derive our recommendations from code for the following practical reasons: First and foremost, only code gives a holistic and comprehensive view on a system while e.g. a single UML diagram is usually not able to provide such a view. 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]). Moreover, agile approaches (valuing working software higher than comprehensive documentation [9]) seem to have been forcing back the use of "heavyweight" CASE tools for more than a decade so that most design decisions today are rather made in front of a whiteboard than could be analyzed automatically in a design repository. As a consequence, there is simply a lack of freely accessible and machine processable software design documents, which could be used for this purpose. 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. Another clear advantage of the chosen approach is the fact that we can rely on various proven code analysis tools such as e.g. PMD [3] in order to implement it and hence we accept that our tool can be rather used during refactoring and not already during design activities.

6 ONGOING & FUTURE WORK

We have been working on deriving and implementing detection rules for further GOF patterns (namely 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 other experienced software developers. Based on the experience gained from our preliminary mini study, we have prepared and executed an online survey with 52 professional software developers and exemplary recommendations for all of the above patterns. We are currently in the process of interpreting these results as well and will also have to report on them at another occasion.

As the results with our tool have been promising so far, we believe it is 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 GoF patterns left that can be analyzed to find out whether it is possible to automatically detect recommendations where to use them. Moreover, there is an enormous number of further design patterns (as e.g. listed in the Pattern Almanach [6]) that can potentially covered be by our approach. And obviously, it also makes sense to investigate, whether larger grained patterns such as those that are listed in Fowler's well-known Enterprise Pattern book [25] might also be detectable with a similar approach.

Assuming that a more comprehensive pattern recommendation tool becomes available in the future, it is certainly interesting to apply this to various existing systems in order to find out whether the amount of missed pattern opportunities might be an indicator for the overall source code quality as well.

7 CONCLUSION

Driven by the unsatisfying situation caused by 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 in our context 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 pattern as defined by the Gang of Four [1] and derived a formal detection rule as well as a set of urgency levels for it by analyzing 53 Strategy implementations retrieved from a well-known source code collection [4].

Moreover, this detection rule was implemented in a prototypical tool 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 recommendations and got a very positive feedback from 9 experienced Java programmers.

Encouraged from these results, we are currently working on analyzing the results applying detection rules for further pattern opportunities and on interpreting results from a larger survey with more than 50 developers that have given feedback for six different pattern opportunities discovered in our test bed of 25 open source systems. As far as we can tell so far, the results definitively justify to invest further effort in this novel approach in the near future.

APPENDIX A

List of analyzed open source projects:

Projekt	Version	Link
ArgoUML	0.34	http://argouml-downloads.tigris.org/argouml-0.34/
Columba	1.4	http://sourceforge.net/projects/columba/
JEdit	5.2	http://sourceforge.net/projects/jedit/
Apache Lucene	4.10.3	http://lucene.apache.org/
JHotDraw	5.6	http://sourceforge.net/projects/jhotdraw/
Apache Ant	1.9.4	https://ant.apache.org/
Apache Wicket	6.18.0	http://wicket.apache.org/
Ganttproject	2-6-1-r1499	http://sourceforge.net/projects/ganttproject/
Jrefactory	2.9.19	http://sourceforge.net/projects/jrefactory/
OpenHab	1.6	http://sourceforge.net/projects/openhab/
Freedomotic	5.5.0	http://sourceforge.net/projects/freedomotic/
Jfreechart	1.0.19+	http://sourceforge.net/projects/jfreechart/
Junit	r4.12	https://github.com/junit-team/junit/
Recoder	0.97	http://sourceforge.net/projects/recoder/
Jenkins	1.598	https://github.com/jenkinsci
Wind	1.0.1	http://sourceforge.net/projects/wind/
Derby	10.11.1.1	http://db.apache.org/derby/
Elasticsearch	1.4.4	https://github.com/elastic/elasticsearch
Freemind	1.0.1	http://sourceforge.net/projects/freemind/
Hibernate	4.5.2	http://sourceforge.net/projects/hibernate/
Jabref	2.10	http://sourceforge.net/projects/jabref/
Megamek	0.40.1	http://sourceforge.net/projects/megamek/
Mina	2.0.9	https://mina.apache.org/
spring-core	4.1.5	http://sourceforge.net/projects/springframework/
Triplea	1.8.0.5	http://sourceforge.net/projects/triplea/

REFERENCES

[1] Vlissides, J., Helm, R., Johnson, R. and Gamma, E., 1995. Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley, 49(120), p.11.
[2] Anonymized
[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 Longman Publishing Co., Inc..
[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), pp.58-61.
[11] Fowler, M. and Beck, K., 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 (ADMD3). Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, pp. 49-54
[13] Anonymized
[14] Halstead, M.H., 1977. Elements of software science (Vol. 7, p. 127). New York: 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, November. A comparison of bug finding tools for Java. In Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on pp. 245-256. IEEE.
[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, pp. 385-394
[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), pp.98-100.
[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.