

University of Szeged
Department of Software Engineering

**Analyzing the Effect of Design Patterns on
Software Maintainability**

Research paper for the
Scientific Students' Associations

Author:
Dénes Bán
MSc student

Supervisors:
Dr. Rudolf Ferenc
Assistant Professor

Péter Hegedűs
PhD student

Szeged
2012

Contents

Abstract	3
1 Introduction	4
2 Related Work	6
3 Approach	8
3.1 Probabilistic Software Quality Model	9
3.2 Mining Design Patterns	11
3.3 Machine learning	12
4 Results	14
4.1 Effect of design pattern changes	15
4.2 Correlation of $PDens_r$ and M_r	16
4.3 Pattern recognition from metrics	17
5 Threats to Validity	21
6 Conclusions	23
Acknowledgements	25
Bibliography	26
A Decision trees	30

Abstract

Although the belief of utilizing design patterns to create better quality software is fairly widespread, there is relatively little research objectively indicating that their usage is indeed beneficial.

In this paper we try to reveal the connection between design patterns and software maintainability. We analyzed more than 300 revisions of JHotDraw, a Java GUI framework whose design relies heavily on some well-known design patterns. We used our probabilistic quality model for estimating the maintainability and we parsed the javadoc annotations of the source code for gathering the pattern instances.

We found that every introduced pattern instance caused an improvement in the different quality attributes. Moreover, the average design pattern line density showed a very high, 0.89 Pearson correlation with the estimated maintainability values. However this connection can also be used the other way around: using the available metrics, machine learning algorithms could predict whether a given class or method is part of a design pattern with 89.3% and 86.7% accuracy respectively in a chosen revision.

Although the amount of available empirical data is still very small, these first results suggest that the usage of design patterns do improve code maintainability.

Keywords: Design patterns, Software maintainability, Empirical validation, Object Oriented design

Chapter 1

Introduction

Since their introduction by Gamma et al. [7], there has been a growing interest in the use of design patterns. Object-Oriented (OO) design patterns represent well-known solutions to common design problems in a given context. The common belief is that applying design patterns results in a better OO design, therefore they improve software quality as well [7, 16].

However, there is a little empirical evidence that design patterns really improve code quality. Moreover, some studies suggest that the use of design patterns not necessarily result in good design [13, 20]. The problem of empirical validation is that it is very hard to assess the effect of design patterns to high level quality characteristics e.g.: maintainability, reusability, understandability, etc. There are some approaches that manually evaluate the impact of certain design patterns on different quality attributes [11].

We also try to reveal the connection between design patterns and software quality but we focus on the maintainability of the source code. As many concrete maintainability models exist (e.g. [2, 8, 4]) we could choose a more direct approach for the empirical evaluation. To get absolute measure for the maintainability of a system we used our probabilistic quality model [2]. Our subject system was JHotDraw 7, a Java GUI framework for technical and structured Graphics (<http://www.jhotdraw.org/>). Its design relies heavily on some well-known design patterns. Instead of using different design pattern mining tools we parsed the javadoc entries of the system directly to get all the applied design patterns. We analyzed more than 300 revisions of JHotDraw, calculated the maintainability values and mined the design pattern instances. We gathered this empirical data

with the following research questions in mind:

Research question 1. *Are there any traceable impacts of the application of design patterns on software maintainability?*

Research question 2. *What kind of relation exists between the design pattern density and the maintainability of the software?*

Research question 3. *Can this suspected connection be used in reverse to predict pattern objects from metrics?*

There are some promising results showing that applying design patterns improve the different quality attributes according to our maintainability model. Regarding the second question, the ratio of the source code lines taking part in some design patterns in the system has a very high correlation with the maintainability in case of JHotDraw. In addition, pattern object prediction from code metrics also seems favorable as machine learning methods could categorize them with high accuracy. However, these results are only a small step towards the empirical analysis of design patterns and software quality.

The rest of our paper is structured as follows. In Chapter 2 we highlight the related work then in Chapter 3 we present our approach for analyzing the relationship between design patterns and maintainability. Chapter 4 summarizes the achieved empirical results. Next, Chapter 5 lists the possible threats to the validity of our work. Finally, we conclude the paper in Chapter 6.

Chapter 2

Related Work

Although the concept of utilizing design patterns in order to create better quality software is fairly widespread, there is relatively little research that would objectively indicate that their usage is indeed beneficial.

Since design patterns and software metrics are both geared towards the same goal - improving quality - Huston [10] attempts to prove their correlation by representing the system's classes in connection matrices and defining algorithms for applying patterns and evaluating metrics. This approach shows promising results but it is purely theoretical.

In an empirical study, - replicated in 2004 [18] and in 2011 [12] - Prechelt et al. [15] gave groups identical maintenance tasks to perform on two different versions - with and without design patterns - of four programs. Here, the impact on maintainability was measured by completion time and correctness while this article uses objective quality metrics and analyzes a more complex software.

In another case study, Vokáč [17] measured the defect frequency of pattern classes versus other classes in an industrial C++ source for three years and concluded that some patterns - Singleton, Observer - tend to indicate more complex parts than others, e.g.: Factory. However, the used pattern mining method could have introduced false positives or true negatives and the defects are also based on subjective reports. In contrast, we rely on the official pattern documentation of source code and the quality model published in [2].

Khomh and Guéhéneuc [11] used questionnaires to collect the opinions of 20 experts on how each design pattern helps or hinders them during maintenance. They bring evi-

dence that design patterns should be used with caution during development because they may actually impede maintenance and evolution. Another experiment conducted by Ng et al. [14] decomposes maintenance tasks to subtasks and examines the frequency of their use according to the deployed design patterns and whether these patterns are utilized during the change. They statistically conclude that performing whichever task while taking existing patterns into consideration yields less faulty code. Trying to evaluate the effectiveness of patterns in software evolution, Hsueh et al. [9] defined their context and their anticipated changes and then checked whether they held up to the expectations. Their conclusion is that although design patterns can be misused, they are effective to some degree in either short or long term maintenance. Aversano et al. [1] also investigate pattern evolution by tracking their modifications and how many other, possibly unrelated modifications they cause. In this paper we do not use questionnaires or evaluate design patterns manually, but rather measure its impact on maintainability directly. Moreover, we focus on their impact on the maintainability of the system as a whole and not only the evolution of the code implementing design patterns.

Antoniol et al. [21] supposes from the start that there is a strong correlation between design patterns and certain low level object oriented metrics and uses this assumption to mine pattern instances from source code. Our aim is to prove that such correlation also exists with higher level metrics like analyzability, testability or maintainability.

In more loosely related articles, Prechelt et al. [23] showed that explicit pattern documentation in itself can further help maintenance and Brito e Abreu and Melo [22] demonstrated the positive effects of object oriented design in general.

Chapter 3

Approach

For analyzing the relationship between design patterns and maintainability we calculate the following measures for every revision of JHotDraw system:

- M_r - an absolute measure of maintainability for the revision r of the system. We used our probabilistic quality model [2] to get this absolute measure.
- $TLLOC$ - the total number of logical lines of code in the system (computed by the Columbus toolset [6]).
- $TNCL$ - the total number of classes in the system.
- PLn_r - the number of pattern instances in revision r of the system.
- PCL_r - the number of classes playing a role in any pattern instances in revision r of the system.
- PLn_r - the total number of logical lines of classes playing a role in any pattern instances in revision r of the system.
- $PDens_r$ - the pattern line density of the system defined by the following formula:
$$\frac{PLn_r}{TLLOC}$$

To answer our research questions we examine the tendency of M_r in comparison to the pattern related metrics and changes in the number of pattern instances. The pattern related

metrics are calculated by our own tool that is able to process the structured *javadoc* comments. The details of the parsing process and program, together with our maintainability model is described below.

3.1 Probabilistic Software Quality Model

Our probabilistic software quality model is based on the quality characteristics defined by the ISO/IEC 9126 standard. The computation of the high level quality characteristics is based on a directed acyclic graph whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). Internal quality properties characterize the software product from an internal (developer) view and are usually estimated by using source code metrics. External quality properties characterize the software product from an external (end user) view and are usually aggregated somehow by using internal and other external quality properties. The nodes representing internal quality properties are called *sensor nodes* as they measure internal quality directly. The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. The edges of the graph represent dependencies between an internal and an external or two external properties. The aim is to evaluate all the external quality properties by performing an aggregation along the edges of the graph, called Attribute Dependency Graph (ADG). We used the particular ADG presented in [2] for assessing the maintainability of JHotDraw Java system. Figure 3.1 shows the applied ADG assessing the maintainability of JHotDraw Java system.

We want to measure how good or bad an attribute is. For every attribute we calculate a goodness value which can basically be regarded as sample of a random variable from the range $[-\infty, \infty]$. Interpolation of the empirical density function leads us to the goodness function of the internal property. To be able to perform the construction of goodness functions in practice, we have built a source code metric repository database, where we uploaded source code metrics of more than 100 open source and industrial software systems. This database serves as the baseline of the maintainability estimation.

Besides constructing goodness functions for internal properties, we need a way to aggregate them along the edges of the ADG. We created an online survey where we asked

many experts (both industrial and academic people) for their opinion about the weights between quality attributes. The number assigned to an edge is considered to be the amount of contribution of source goodness to target goodness. Taking into account every possible combination of goodness values and weights, and also the probabilities of their outcome, we defined a formula for computing goodness functions for every aggregate node.

The aggregation method ensures that the goodness functions of aggregate nodes are really expressing the probabilities of their goodness from the aspect represented by the node.

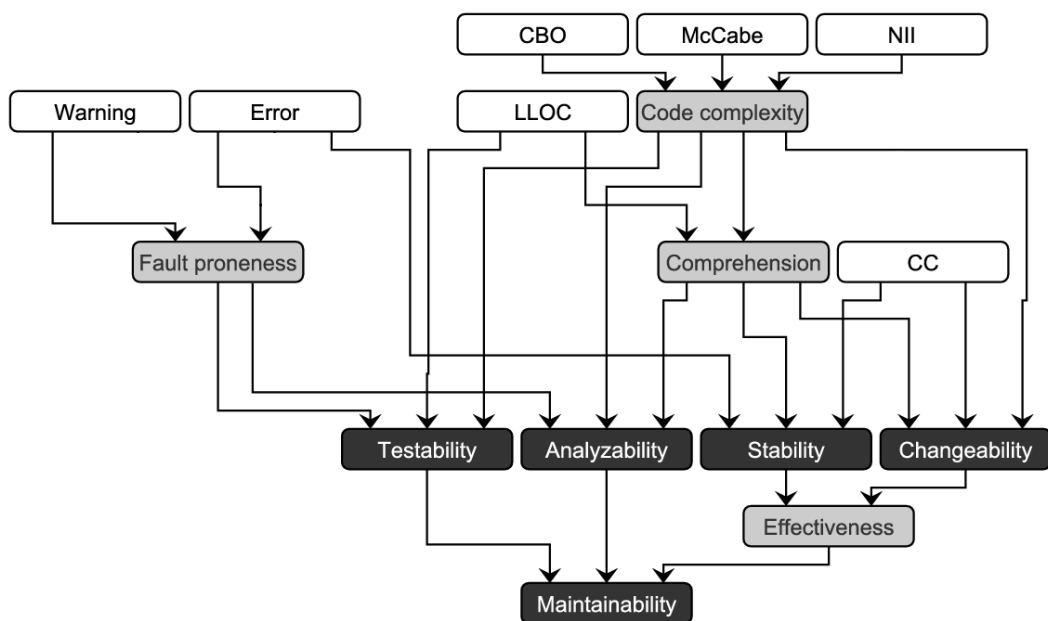


Figure 3.1: The applied maintainability model

A method based high-level quality metric is also derived from these goodness values by proportionally dividing the value of its enclosing object. Its absolute value is usually really close to zero - a magnitude of $x \cdot 10^{-5}$ or less - but its sign can be both positive or negative depending on whether it improves or reduces the quality of the system as a whole. This value will be used in pattern instance prediction in Section 4.3.

3.2 Mining Design Patterns

Instead of applying one of the design pattern miner tools (e.g. [5, 19]) we used a more direct way for extracting pattern instances from different JHotDraw versions. Since every design pattern instance is documented in JHotDraw 7 we could easily build a text parser application to extract and collect all the pattern information. This approach guarantees that no false positive instances are included and no true negative instances are left out from the empirical analysis. A sample of design pattern *javadoc* documentation can be seen in Listing 3.1:

```
/**
 *
 * <b>Design Patterns </b>
 *
 * <p><em>Strategy </em><br>
 * The different behavior states of the selection tool are implemented by
 * trackers. Context: {@link SelectionTool}; State: {@link DragTracker},
 * {@link HandleTracker}, {@link SelectAreaTracker}.
 *
 */
```

Listing 3.1: The source design pattern documentation

The text parser processed the two types of pattern comments that appeared in the source - the above listing being one of them. Then - using regular expressions - it obtained the names of the patterns and a list of the participants from the *@link* directives. This list contained the name of both the roles - e.g. *Context* for the Strategy pattern - and the classes that fit them - *SelectionTool*. After this, all fully qualified name references were trimmed, - *foo.Bar* became *Bar* - because of the inconsistent naming conventions. Then the lists were alphabetically ordered, converted to a unique string form and added to a set in order to avoid pattern instance duplication even if a pattern was documented in more than one of its participants' codes. Finally we ran the parser on all relevant revisions of JHotDraw7. The extracted data can be seen in Listing 3.2:

This information was sufficient for us to track the changes because even if the number of pattern instances was the same in two consecutive revisions we could still check whether the patterns themselves - or even their participants - have changed or not.

```
...
Chain of responsibility [ DragTracker->Handler , Handle->Handler , SelectionTool->Client ]
Composite [ CompositeFigure->Composite , Figure->Component ]
Decorator [ BezierFigure->Component , LineDecoration->Decorator ]
Decorator [ DecoratedFigure->Component , Figure->Decorator ]
Framework [ Application->Contract , ApplicationModel->Contract , View->Contract ]
...
Strategy [ DragTracker->State , HandleTracker->State , SelectAreaTracker->State ,
          SelectionTool->Context ]
...
```

Listing 3.2: The extracted design pattern information

We followed this step with a manual verification wherever the text parser indicated a change. There were some cases where the sudden increase in pattern numbers were only due to the - late - appearance of documentation and the involved classes were already a part of the system. We later disregarded these anomalies when determining the correlation in question.

3.3 Machine learning

From the data in Listing 3.2 we could easily compile the list of classes that participate in at least one design pattern. We then added this information as a new column to the already present metrics of a specific revision - 590, chosen for its highest pattern line density. For method level metrics, we used the heuristic that a method is in a design pattern if its enclosing class is in one. This way we had two separate data sets where the new column was the classifier attribute and every other metrics could be used as predictors.

From these two tables we wanted to find out if they have any underlying structure that could be helpful in detecting pattern instances. This is exactly the type of problem machine learning is concerned with. Machine learning is a branch of artificial intelligence that tries to predict the workings of the mechanism that generated its input empirical data - its training set - to be able to correctly label later unclassified instances. There are many approaches for implementing such an algorithm. As this is a textbook example of a classification task, we chose the most common type of solution for it: decision trees.

Decision tree - also known as *classification tree* or *regression tree* - based learning uses a tree structure to map observations about a given item to the conclusions of the

item's target class. In these trees, leaves represent class labels and branches represent how - by what decisions - we can get there. One powerful and popular decision tree building algorithm is the *C4.5* [25], an improved version of the earlier *ID3* [24].

For the actual machine learning process, we used Weka¹, a Java based open source data mining software. It implements lots of machine learning methods from which we chose the *J48*, Weka's version of the *C4.5* algorithm mentioned above.

¹<http://www.cs.waikato.ac.nz/ml/weka/>

Chapter 4

Results

We analyzed all the 779 revisions of the JHotDraw 7 subversion branch¹ and calculated the measures introduced in Chapter 3. The documentation of design patterns is introduced in revision 522, therefore the empirical evaluation has been performed on 258 revisions (between revision 522 and 779). Some basic properties of the starting, ending and most pattern-dense revision of the analyzed JHotDraw system can be seen in Table 4.1.

Revision (<i>r</i>)	Lines of code	Nr. of classes	Nr. of methods	$PI n_r$	$\frac{PCL_r}{TNCL}$	$PDens_r$
522	72472	630	6117	45	11.587%	16.915%
590	72385	635	6129	52	12.756%	17.826%
779	81686	685	6573	54	13.285%	16.758%

Table 4.1: Basic properties of JHotDraw 7 system

Figure 4.1 depicts the ratio of classes playing a role in some pattern instances and the total number of classes in the system. We note that the deliberately applied design patterns cause a very high ratio (about 13% of the classes are part of a design pattern).

¹<https://jhotdraw.svn.sourceforge.net/svnroot/jhotdraw/trunk/jhotdraw7/>

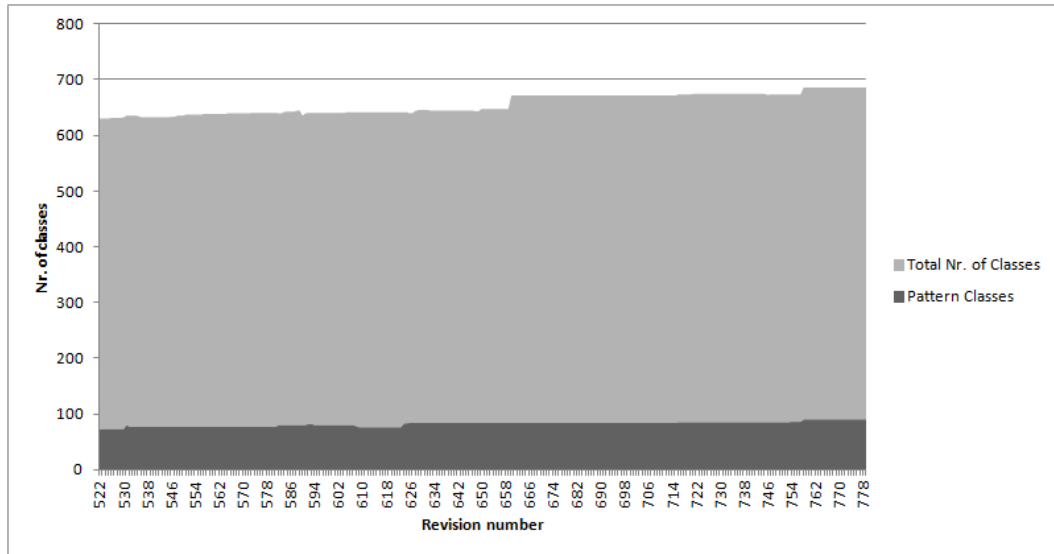


Figure 4.1: The number of pattern classes compared to the total number of classes

4.1 Effect of design pattern changes

To be able to answer our first research question we have analyzed those particular revisions where the number of design pattern instances has changed. After filtering out the changes that does not introduce or remove real pattern instances (e.g.: comments are added to an already existing pattern instance) five revisions have remained. We also checked that these change sets do not contain a lot of source code that is not related to patterns. We did this by parsing the locations of the changes and compared them against the list of pattern-related classes because each source file corresponds to exactly one class. Then we computed the relevant change ratio as the number of lines that did not match divided by the total length of the *diff*. It is important to be able to clearly distinguish the effect of design pattern changes to maintainability compared to other changes that could be deliberate refactorings, thereby improving quality while being completely unrelated to design patterns. In all five cases more than 90% of the code changes are related to the pattern implementations - either modifying existing code or adding new source files that are responsible for a participant of a new instance.

The tendency of different quality attributes in these revisions can be seen in Table 4.2.

In four out of five cases there was growth in the pattern instance numbers. In all of these four cases every ISO/IEC 9126 quality characteristic (including the maintainability)

Revision (r)	Pattern	Pattern Line Density ($PDens_r$)	Maintain- ability (M_r)	Test- ability	Analyz- ability	Stability	Change- ability
531	+3	↗	↗	↗	↗	↗	↗
574	+1	↗	↗	↗	↗	↗	↗
609	-1	↘	—	—	—	—	—
716	+1	↘	↗	↗	↗	↗	↗
758	+1	↗	↗	↗	↗	↗	↗

Table 4.2: The tendency of the quality attributes in case of design pattern changes

increased compared to the previous revision. This is true even for revision 716 where the pattern line ratio decreased despite the addition of a design pattern. In case of revision 609 a *Framework* pattern has been removed but the quality characteristics have remained unchanged. This is not so surprising since this pattern (which is not part of the GoF patterns) consists of a simple interface. Therefore its removal does not have any effect on the low level source code metrics which our maintainability model is based on.

As we have shown in one of our previous works [3] a system's maintainability does not improve during development without applying explicit refactorings. Therefore, the application of design patterns can be seen as applying refactorings on the source code. These results support the hypothesis that design patterns do have a traceable impact on maintainability. In addition, our empirical analysis on JHotDraw indicates that this impact is positive.

4.2 Correlation of $PDens_r$ and M_r

To answer our second research question and shed light on the relationship between design pattern density and maintainability we performed a correlation analysis on pattern line density ($PDens_r$) and maintainability (M_r). We chose pattern line density instead of pattern instance or pattern class density because it is the finest grained measure showing the amount of source code related to any pattern instances. Figure 4.2 depicts the tendencies of pattern line density and maintainability. It can be seen that the two curves

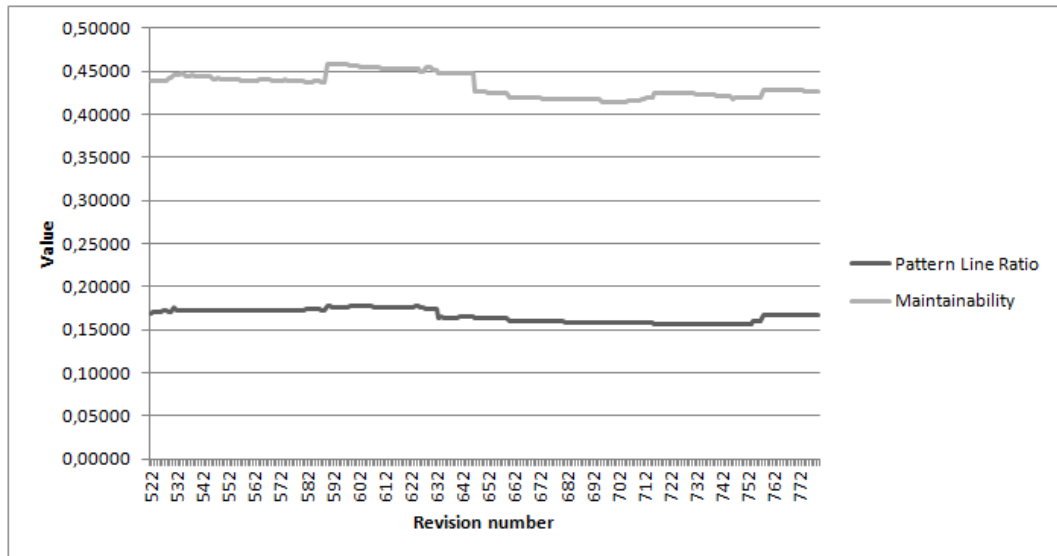


Figure 4.2: The tendencies of pattern line density and maintainability

have a similar shape meaning that they move very much together. The Pearson correlation analysis of the entire data set (from revision 522 to 779) shows the same result, the pattern line density and maintainability has a **0.89** correlation. This result may indicate that there is a strong relation between the rate of design patterns in the source code and the maintainability. However, this is still a surmise and we cannot generalize the results without performing a large number of additional empirical evaluations.

4.3 Pattern recognition from metrics

Having run both the pattern parser tool and the Columbus toolset on the relevant revisions of JHotDraw7 source code, we obtained system, class and method-level metrics as well as the pattern information. We needed one specific revision for this type of testing, so we chose number 590 because this revision had the highest value of pattern line density - namely 17.826%. We then preprocessed this data by dividing it to separate class and method metrics tables, removing unneeded entries such as name or original code position and adding a new column which contained "true" if the given object was associated with any of the recognised design patterns, "false" otherwise. The last association was determined by simple «pattern participant» to «class name» matching for the classes and «pattern participant» to «enclosing class of method» matching for methods. This pairing

gave the heuristic that every method is in a pattern whose parent class is in one. This - although not guaranteed to be correct in every case - is the closest approximation short of manually examining every method of every pattern-related class. We also added the separately computed high level method quality metric to the method table as another predictor. Either utilizing or ignoring it lead to the two distinct method based searches described below.

After preparing these two tables, we performed the machine learning analysis of both class and method metrics using the algorithm described in Section 3.3. The important values provided by the output are:

- **Accuracy** - The number - and ratio - of correctly and incorrectly classified instances belonging to a given class
- **TP Rate** - True Positive Rate, the number of correctly identified instances / the cardinality of the given class, equivalent to **Recall**
- **FP Rate** - False Positive Rate, the number of incorrectly identified instances / the cardinality of all the other classes except the given class
- **Precision** - Proportion of the examples which are truly in the given class / total number of instances classified as the given class
- **F-Measure** - $2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$, a balanced combined measure for precision and recall
- **ROC Area** - The area under the Receiver Operating Characteristic curve, a plot of the true positive rate against the false positive rate
- **Class** - The class the other measures on the line refer to

Every available metric was used as a predictor and the only two classes were whether the object is included in a pattern or not. We expected the algorithm to be able to label a majority of the cases correctly since code metric and design patterns both mean to express the same general idea and therefore should be interoperable to a degree.

The prediction from class metrics yielded a 89.3% accurate result as the decision tree was able to correctly classify 637 instances from the total number of 713. The more

important statistics of the search are presented in Table 4.3 while the full decision tree is included as Figure A.1 in Appendix A.

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.945	0.526	0.936	0.945	0.94	0.717	false
	0.474	0.055	0.514	0.474	0.493	0.717	true
Weighted Avg.	0.893	0.474	0.89	0.893	0.892	0.717	

Table 4.3: The results of machine learning on class metrics

After this we ran the same analysis on the method related metrics - once with the high-level method quality metric described in Section 3.1 and once without it. In the first case the results show 86.7% accuracy since the classification based on the built decision tree was correct in 5894 out of 6797 cases. In the second case the accuracy is 86.6% with 5885 true positives. However, the difference of only 0.1% is to be expected because the method quality metric is derived from the other metrics, therefore not independent and gives almost no new classifying power to the decision tree. Again, the more important information related to the analysis is available in Table 4.4 and Table 4.5 and the latter full search tree is included as Figure A.2 in Appendix A.

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.957	0.582	0.892	0.957	0.923	0.718	false
	0.418	0.043	0.658	0.418	0.511	0.718	true
Weighted Avg.	0.867	0.493	0.853	0.867	0.855	0.718	

Table 4.4: The results of machine learning on method metrics **with** the quality attribute

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.955	0.58	0.892	0.955	0.922	0.721	false
	0.42	0.045	0.649	0.42	0.51	0.721	true
Weighted Avg.	0.866	0.491	0.852	0.866	0.854	0.721	

Table 4.5: The results of machine learning on method metrics **without** the quality attribute

From these results - despite the small amount of data and this research being only the first in hopefully many more empirical validations on the subject to come - we conclude that objective source code metrics may be useful in not only indicating software quality but also in finding design pattern instances.

Chapter 5

Threats to Validity

Similarly to most of the works, our approach also has some threats to validity. First of all, when dealing with design patterns, the accuracy of mining is always a question. As there are no provably perfect pattern miner tools, we chose our subject system to be a special one, having all design pattern instances documented by its authors. This way we can be sure that all (intentionally placed) design patterns are recognized and no false positive instances are introduced. Of course it is still possible that some pattern comments are missing or our text parser introduces false instances. We reduced this effect by manually inspecting the results of our text parser as well as the source code of JHotDraw.

Another threat to validity is using our previously published quality model for calculating maintainability values. Although we have done some empirical validation on our probabilistic quality model in our previous work, we cannot state that the used maintainability model is fully validated. Moreover, as the ISO/IEC 9126 standard is not defining the low-level metrics, the results can vary depending on the quality model's settings (chosen metrics and weights given by professionals). It is also very important to have a source code metrics repository with a large enough number of systems to get an objective absolute measure for maintainability. The maintainability model uses only source code metrics and no process metrics at all. These factors are possible threats to validity, but our first results and continuous empirical validation of the maintainability model proves its applicability and usefulness.

Finally, the small number of design pattern changes and the fact that less than 300 revisions of one system have been evaluated threatens the generality of results. It might

be possible that the explored relationship between design patterns and maintainability is just a byproduct of other factors. Our analysis is only a first step towards the empirical analysis of this relation. Nonetheless, these first results are already valuable and support the common belief that design patterns do have a positive impact on maintainability.

Chapter 6

Conclusions

In this paper we presented an empirical analysis of exploring the connection between design patterns and software maintainability. We analyzed nearly 300 revisions of JHot-Draw 7 and calculated the maintainability values with our probabilistic quality model and mined the design pattern instances parsing the comments in the source code. Examining the maintainability values where changes in the number of pattern instances happened and by correlation analysis of design pattern density and maintainability we were able to draw some conclusions.

Every ISO/IEC 9126 quality characteristics (including the maintainability) increased with the number of pattern instances. Since there were no other changes in the code it indicates that the quality attributes increased due to the introduced patterns. Hence, we could observe a traceable positive impact of design patterns to maintainability of the subject system.

Another interesting result is that the pattern line density and maintainability values have a very similar tendency. The Pearson correlation analysis of the data sets showed that there is a strong relation between the rate of design patterns in the source code and its maintainability. These facts strengthen the common assumption that using design patterns improve the maintainability of the source code. However, these results should be handled with caution. We analyzed only one system and a relatively few number of pattern instance changes. We are far from drawing some general conclusions based on this; our work should be considered as a first step towards the empirical validation of the relation between design patterns and software maintainability.

As a future work we plan to repeat the presented analysis on a large number of revisions of other subject systems. Furthermore, we want to involve some of the well-known design pattern miner tools into matching design patterns. One of the other directions of the future work could be to analyze the maintainability on a finer level. Instead of comparing the pattern density and the system level maintainability we could calculate the maintainability measures for every class (or even method) in the system. Then we could compare the maintainability values of those program elements that play some role in a design pattern to all the other elements. Our hypothesis is that the program elements taking part in some design patterns have a significantly larger maintainability value in average compared to other elements.

Acknowledgements

I would like to thank Péter Hegedűs and Dr. Rudolf Ferenc for their continuing support and guidance.

Bibliography

- [1] Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., Di Penta, M.: An Empirical Study on the Evolution of Design Patterns. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. pp. 385–394. ESEC-FSE '07, ACM, New York, NY, USA (2007)
- [2] Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A Probabilistic Software Quality Model. In: Proceedings of the 27th IEEE International Conference on Software Maintenance. pp. 368–377. ICSM 2011, IEEE Computer Society, Williamsburg, VA, USA (2011)
- [3] Bakota, T., Hegedűs, P., Ladányi, G., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A Cost Model Based on Software Maintainability. In: Proceedings of the 28th IEEE International Conference on Software Maintenance. ICSM 2012, IEEE Computer Society, Williamsburg, VA, USA (2012)
- [4] Bansiya, J., Davis, C.: A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering* 28, 4–17 (2002)
- [5] Dong, J., Lad, D.S., Zhao, Y.: DP-Miner: Design Pattern Discovery Using Matrix. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. pp. 371–380. ECBS '07, IEEE Computer Society, Washington, DC, USA (2007)
- [6] Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus – Reverse Engineering Tool and Schema for C++. In: Proceedings of the 18th International Confer-

- ence on Software Maintenance (ICSM 2002). pp. 172–181. IEEE Computer Society (Oct 2002)
- [7] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley Pub Co (1995)
- [8] Heitlager, I., Kuipers, T., Visser, J.: A Practical Model for Measuring Maintainability. Proceedings of the 6th International Conference on Quality of Information and Communications Technology pp. 30–39 (2007)
- [9] Hsueh, N.L., Wen, L.C., Ting, D.H., Chu, W., Chang, C.H., Koong, C.S.: An Approach for Evaluating the Effectiveness of Design Patterns in Software Evolution. In: IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW). pp. 315 –320 (July 2011)
- [10] Huston, B.: The Effects of Design Pattern Application on Metric Scores. Journal of Systems and Software pp. 261–269 (2001)
- [11] Khomh, F., Guéhéneuc, Y.G.: Do Design Patterns Impact Software Quality Positively? In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering. pp. 274–278. CSMR '08, IEEE Computer Society, Washington, DC, USA (2008)
- [12] Krein, J., Pratt, L., Swenson, A., MacLean, A., Knutson, C., Eggett, D.: Design Patterns in Software Maintenance: An Experiment Replication at Brigham Young University. In: Second International Workshop on Replication in Empirical Software Engineering Research (RESER 2011). pp. 25 –34 (Sept 2011)
- [13] McNatt, W.B., Bieman, J.M.: Coupling of Design Patterns: Common Practices and Their Benefits. In: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development. pp. 574–579. COMPSAC '01, IEEE Computer Society, Washington, DC, USA (2001)
- [14] Ng, T.H., Cheung, S.C., Chan, W.K., Yu, Y.T.: Do Maintainers Utilize Deployed Design Patterns Effectively? In: Proceedings of the 29th international conference

- on Software Engineering. pp. 168–177. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007)
- [15] Prechelt, L., Unger, B., Tichy, W., Brössler, P., Votta, L.: A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering* 27, 1134–1144 (2001)
- [16] Venners, B.: How to Use Design Patterns - A Conversation With Erich Gamma, Part I (2005), <http://www.artima.com/lejava/articles/gammadp.html>
- [17] Vokáč, M.: Defect Frequency and Design Patterns: an Empirical Study of Industrial Code. *IEEE Transactions on Software Engineering* 30(12), 904 – 917 (Dec 2004)
- [18] Vokáč, M., Tichy, W., Sjøberg, D.I.K., Arisholm, E., Aldrin, M.: A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns - A Replication in a Real Programming Environment. *Empirical Software Engineering* 9(3), 149–195 (Sep 2004)
- [19] Wendehals, L.: Improving Design Pattern Instance Recognition by Dynamic Analysis. In: *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA)*, Portland, USA (2003)
- [20] Wendorff, P.: Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In: *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*. pp. 77–. CSMR '01, IEEE Computer Society, Washington, DC, USA (2001)
- [21] Antoniol, G., Fiutem, R., Cristoforetti, L.: Using metrics to identify design patterns in object-oriented software. In: *PROC. IEEE-CS SOFTWARE METRICS SYMP.*, pages 23–34, (1998)
- [22] Brito e Abreu, F., Melo, W.: Evaluating the impact of object-oriented design on software quality. In: *IEEE International Symposium on Software Metrics*, 0:90, (1996)

- [23] Prechelt, L., Unger-Lamprecht, B., Philippsen, M., and Tichy, M.: Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, (June 2002)
- [24] Quinlan, J. R.: Induction of decision trees. *Machine Learning*, 1:81–106, (1986)
- [25] Quinlan, J. R.: *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, (1993)

Appendix A

Decision trees

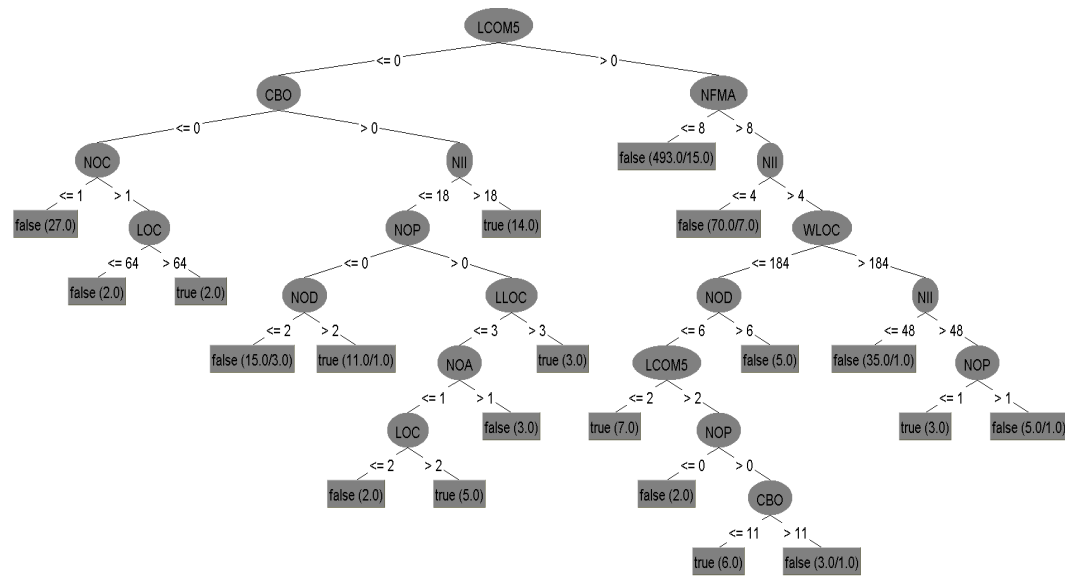


Figure A.1: Class decision tree

