Recognizing Antipatterns and Analyzing their Effects on Software Maintainability

Dénes Bán and Rudolf Ferenc

University of Szeged, Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
`{zealot,ferenc}@inf.u-szeged.hu`

Abstract. Similarly to design patterns and their inherent extra information about the structure and design of a system, antipatterns – or bad code smells – can also greatly influence the quality of software. Although the belief that they negatively impact maintainability is widely accepted, there are still relatively few objective results that would support this theory.
In this paper we show our approach of detecting antipatterns in source code by structural analysis and use the results to reveal connections among antipatterns, number of bugs, and maintainability. We studied 228 open-source Java based systems and extracted bug-related information for 34 of them from the PROMISE database. For estimating the maintainability, we used the ColumbusQM probabilistic quality model.
We found that there is a statistically significant, 0.55 Spearman correlation between the number of bugs and the number of antipatterns. Moreover, there is an even stronger, -0.62 reverse Spearman correlation between the number of antipatterns and code maintainability. We also found that even these few implemented antipatterns could nearly match the machine learning based bug-predicting power of 50 class level source code metrics.
Although the presented analysis is not conclusive by far, these first results suggest that antipatterns really do decrease code quality and can highlight spots that require closer attention.

Keywords: Antipatterns, Software maintainability, Empirical validation, OO design, ISO/IEC 25010, SQuaRE

1 Introduction

Antipatterns can be most simply thought of as the opposites of the more well-known design patterns [5]. While design patterns represent "best practice" solutions to common design problems in a given context, antipatterns describe a commonly occurring solution to a problem that generates decidedly negative consequences [2]. Also an important distinction is that antipatterns have a refactoring solution to the represented problem, which preserves the behavior of the code, but improves some of its internal qualities [4]. The widespread belief is that the more antipatterns a software contains, the worse its quality is.

Some research even suggests that antipatterns are symptoms of more abstract design flaws [9, 16]. However, there is little empirical evidence that antipatterns really decrease code quality.

We try to reveal the effect of antipatterns by investigating its impact on maintainability and its connection to bugs. For the purpose of quality assessment, we chose our ColumbusQM probabilistic quality model [1], which ultimately produces one number per system describing how "good" that system is. The antipattern-related information came from our own, structural analysis based extractor tool and source code metrics were computed using the Columbus CodeAnalyzer reverse engineering tool [3]. We compiled the types of data described above for a total of 228 open-source Java systems, 34 of which have corresponding class level bug numbers from the open-access PROMISE [13] database. With all this information we try to answer the following questions:

Research question 1. *What kind of relation exists between antipatterns and the number of known bugs?*

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

Research question 3. *Can antipatterns be used to predict future software faults?*

We obtained some promising results showing that antipatterns indeed negatively correlate with maintainability according to our quality model. Moreover, antipatterns correlate positively with the number of known bugs and also seem to be good attributes for bug prediction. However, these results are only a small step towards the empirical validation of this subject.

The rest of our paper is structured as follows. In Section 2 we highlight the related work. Then, in Section 3 we present our approach for extracting antipatterns and analyzing their relationship with bugs and maintainability. Section 4 summarizes the achieved empirical results. Next, Section 5 lists the possible threats to the validity of our work. Finally, we conclude the paper in Section 6.

2 Related Work

The most closely related research to our current work was done by Marinescu. In his publication in 2001 [11], he emphasized that the search for given types of flaws should be systematic, repeatable, scalable and language-independent. First, he defined a unified format for describing antipatterns and then a methodology for evaluating those. He showed this method in action using the GodClass and DataClass antipatterns and argued that it could similarly be done for any other pattern. To automate this process, he used his own TableGen software to analyze C++ source code – analogous to the Columbus tool in our case, – save its output to a database and extract information using standard queries.

In one of his works from 2004 [12], he was more concerned with automation and made declaring new antipatterns easier with "detection strategies." In these, one can define different filters for source metrics – limit or interval, absolute or relative – even with statistical methods that set the appropriate value by analyzing all values first and computing their average, mean, etc., to find outliers. Finally, these intermediate result sets can be joined by standard set operations like union, intersection, or difference. When manually checking the results he defined a "loose precision" value next to the usual "strict" one that did not label a match as a false positive if it *was* indeed faulty but not because of the searched pattern. His conclusion is an empirically 70% accurate tool that can be considered successful.

These "detection strategies" were extended with historical information by Rapu et al. [15]. They achieved this by running the above described analysis on not only the current version of their subject software system – Jun 3D graphical framework – but every fifth from the start. This way they could extract two more metrics: persistence that means how much of its "lifetime" was a given code element faulty, and stability that means how many times the code element changed during its life. The logic behind this was that e.g. a GodClass antipattern is dangerous only if it is not persistent – the error is due to changes, not part of the original design – or not stable – when it really disturbs the evolution of the system. With this method they managed to halve the candidates in need of manual checking in the case of the above example.

Trifu and Marinescu went further by assuming that these antipatterns are just symptoms of larger, more abstract faults [16]. They proposed to group several antipatterns – that may even occur together often – and supplemented them with contextual information to form "design flaws." Their main goal was to make antipattern recognition – and pattern recognition in general – more of a well-defined engineering task rather than a form of art.

Our work is similar to the ones mentioned above in that we also detect antipatterns by employing source code metrics and static analysis. But, in addition, we inspect the correlations of these patterns to the maintainability values of the subject systems and also consider bug-related information to more objectively prove the common belief that they are indeed connected.

In another approach, Khomh et al. [8] concentrated more on statistical methods and asked whether there is a connection between antipatterns and change-proneness. They used their own extractor tool "DECOR" and null-hypotheses to answer this question and concluded that antipatterns really increase change-proneness. We similarly use statistical methods – beside machine learning – but instead of change-proneness we concentrate on maintainability and bug information.

Lozano et al. [9] overviewed a broader sweep of related works and urged researchers to standardize their efforts. Apart from the individual harms antipatterns may cause, they aimed to find out that from exactly when in the life cycle of a software can an antipattern be considered "bad" and – not unlike [16] – whether these antipatterns should be raised to a higher abstraction level. In contrast to the historical information, we focus on objective metric results to shed light on the effect of antipatterns.

Also another approach by Mäntylä et al. [10] is to use questionnaires to reveal the subjective side of software maintenance. The different opinions of the participating developers could mostly be explained by demographic analysis and their roles in the company but there was a surprising difference compared to the metric based results. We, on the other hand, make these objective, metric based results our priority.

3 Approach

For analyzing the relationship between antipatterns, bugs, and maintainability we calculated the following measures for the subject systems:

- an absolute measure of maintainability per system (we used our ColumbusQM probabilistic quality model [1] to get this value).
- the total number of antipatterns per system.
- the total number of bugs per system.

For the third research question, we could compile an finer grained set of data – since the system-based quality attribute is not needed here:

- the total number of antipatterns related to each class in every subject system.
- the total number of bugs related to every class in every subject system.
- every class level metric for each class in every subject system.

The metric values were extracted by the Columbus tool [3], the bug number information comes from the PROMISE open bug database [13] and the pattern related metrics are calculated by our own tool described in Subsection 3.2.

3.1 Used Metrics

We used the following source code metrics for antipattern recognition and maintainability calculation:

- **AD** (**A**PI **D**ocumentation): ratio of the number of documented public members of a class or package to the number of all of its public members.
- **CBO** (**C**oupling **B**etween **O**bjects): The CBO metric for a class means the number of directly used different classes by the class.
- **CC** (**C**lone **C**overage): ratio of code covered by code duplications in the source code element to the size of the source code element, expressed in terms of the number of syntactic entities (statements, expressions, etc.).
- **CD** (**C**omment **D**ensity): ratio of the comment lines of the source code element (CLOC) to the sum of its comment (CLOC) and logical lines of code (LLOC).
- **CLOC** (**C**omment **L**ines **O**f **C**ode): number of comment and documentation code lines of the source code element; however, its nested, anonymous or local classes are not included.

- **LLOC** (**L**ogical **L**ines **O**f **C**ode): number of code lines of the source code element, without the empty and comment lines; its nested, anonymous or local classes are not included.
- **McCC** (**Mc**Cabe's **C**yclomatic **C**omplexity): complexity of the method expressed as the number of independent control flow paths in it.
- **NA** (**N**umber of **A**ttributes): number of attributes in the source code element, including the inherited ones; however, the attributes of its nested, anonymous or local classes (or subpackages) are not included.
- **NII** (**N**umber of **I**ncoming **I**nvocations): number of other methods and attribute initializations which directly call the method (or methods of a class).
- **NLE** (**N**esting **L**evel **E**lse-If): complexity expressed as the depth of the maximum "embeddedness" of the conditional and iteration block scopes in a method (or the maximum of these for the container class), where in the if-else-if construct only the first if instruction is considered.
- **NOA** (**N**umber **O**f **A**ncestors): number of classes, interfaces, enums and annotations from which the class is directly or indirectly inherited.
- **NOS** (**N**umber **O**f **S**tatements): number of statements in the source code element; however, the statements of its nested, anonymous or local classes are not included.
- **RFC** (**R**esponse set **F**or **C**lass): number of local (i.e. not inherited) methods in the class (NLM) plus the number of directly invoked other methods by its methods or attribute initializations (NOI).
- **TLOC** (**T**otal **L**ines **O**f **C**ode): number of code lines of the source code element, including empty and comment lines, as well as its nested, anonymous or local classes.
- **TNLM** (**T**otal **N**umber of **L**ocal **M**ethods): number of local (i.e. not inherited) methods in the class, including the local methods of its nested, anonymous or local classes.
- **Warning P1, P2 or P3**: number of different coding rule violations reported by the PMD analyzer tool, categorized into three priority levels.
- **WMC** (**W**eighted **M**ethods per **C**lass): The WMC metric for a class is the total of the McCC metrics of its local methods.

3.2 Mining Antipatterns

The whole process of analyzing the subject source files and extracting antipatterns is shown in Figure 1.

First, we convert the source code – through a language specific format and a linking stage – to the LIM model (**L**anguage **I**ndependent **M**odel), a part of the Columbus framework. It represents the information obtained from the static analysis of code in a more abstract, graph-like format. It has different types of nodes that correspond to e.g. classes, methods, attributes, etc. while different edges represent the connections between these.

From this data, the LIM2Metrics tool can compute various kinds of source code metrics – e.g. logical lines of code or number of statements in the *size* category, comment lines of code or API documentation in the *documentation*

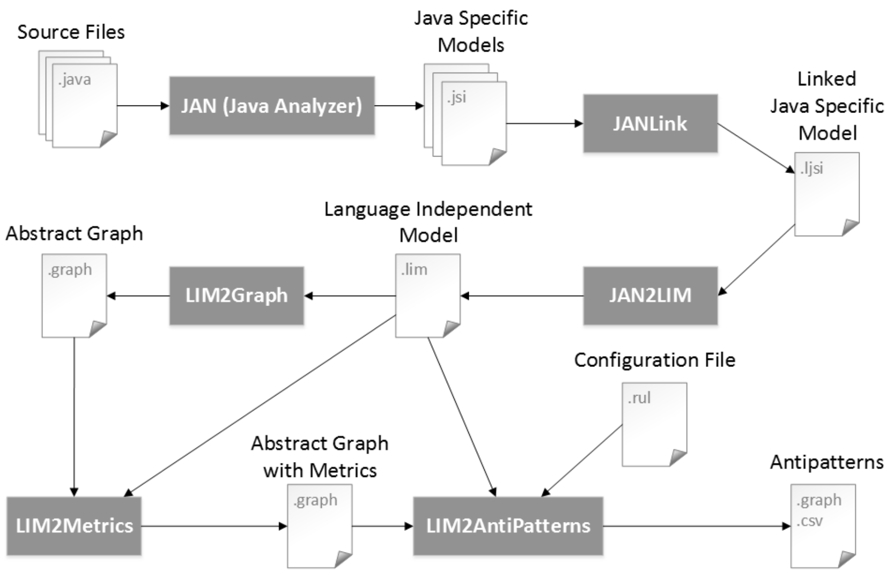

Fig. 1. The stages of the analysis

category and also different complexity, cohesion and inheritance metrics (see also Section 3.1). As these values are not a part of the LIM model, the output is pinned to an even more abstract graph – fittingly named "graph". This format literally only has "Nodes" and "Edges" but nodes can have dynamically attached and dynamically typed attributes. Since the LIM model is "strict" – e.g. it can only have statically typed attributes defined in advance – the "graph" format is more suitable as a target for the results of metrics, pattern matches, etc.

Each antipattern implementation can define one or more externally configurable parameters, mostly used for easily changeable metric thresholds. These come from an XML-style rule file – called RUL – that can handle multiple configurations and even inheritance. It can also contain language-aware descriptions and warning messages that will be attached to the affected graph nodes.

After all these preparations, our tool can be run. It is basically a single new class built around the Visitor design pattern [5] which is appropriate as it is a new operation defined for an existing data structure and this data structure does not need to be changed to accommodate the modification. It "visits" the LIM model and uses its structural information and the computed metrics from its corresponding graph nodes to identify antipatterns. It currently recognizes the 9 types of antipatterns listed below. They are described in greater detail by Fowler and Beck [4], here we just provide a short informal definition and explain how we interpreted them in the context of our LIM model. The parameters of the recognition are denoted with a starting **$** sign and can be configured in the

RUL file mentioned above. The referenced object-oriented source code metrics can be found in Subsection 3.1.

- **Feature Envy** (FE): A class is said to be envious of another class if it is more concerned with the attributes of that other class than those of its own. Interpreted as a method that accesses at least $MinAccess attributes and at least $MinForeign% of those belong to another class.
- **Lazy Class** (LC): A lazy class is one that does not do "much", only delegates its requests to other connected classes – i.e. a non-complex class with numerous connections. Interpreted as a class whose CBO metric is at least $MinCBO but its WMC metric is no more than $MaxWMC.
- **Large Class Code** (LCC): Simply put, a class that is "too big" – i.e. it probably encapsulates not just one concept or does too much. Interpreted as a class whose LLOC metric is at least $MinLLOC.
- **Large Class Data** (LCD): A class that encapsulates too many attributes, some of which might be extracted – along with the methods that more closely correspond to them – into smaller classes and be a part of the original class through aggregation or association. Interpreted as a class whose NA metric is at least $MinNA.
- **Long Function** (LF): Similarly to LCC, if a method is too long, it probably has parts that could – should – be separated into their own logical entities, thereby making the whole system more comprehensible. Interpreted as a method where either one of the LLOC, NOS or McCC metrics exceed $MinLLOC, $MinNOS or $MinMcCC, respectively.
- **Long Parameter List** (LPL): The long parameter list is one of the most recognized and accepted "bad code smells" in code. Interpreted as a function (or method) whose number of parameters is at least $MinParams.
- **Refused Bequest** (RB): If a class refuses to use its inherited members – especially if they are marked "protected," by which the parent expresses that descendants *should* most likely use it – then it is a sign that inheritance might not be the appropriate method of implementation reuse. Interpreted as a class that inherits at least one protected member that is not accessed by any locally defined method or attribute.
- **Shotgun Surgery** (SHS): Following the "Locality of Change" principle, if a method needs to be modified then it should not cause the need for many other – especially remote – modifications, otherwise one of those can easily be missed leading to bugs. Interpreted as a method whose NII metric is at least $MinNII.
- **Temporary Field** (TF): If an attribute only "makes sense" to a small percent of the container class then it – and its closely related methods – should be decoupled. Interpreted as an attribute that is only referenced by at most $RefMax% of the members of its container class.

3.3 ColumbusQM Software Quality Model

Our probabilistic software quality model [1] is based on the quality characteristics defined by the ISO/IEC 25010 standard [7]. 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 edges of the graph represent dependencies between an internal and an external or two external properties. In addition to the external nodes defined by the standard (black nodes) we introduced new ones (light gray nodes) and even kept those that were contained only in the old ISO/IEC 9126 standard (dark gray nodes). 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 calculate a so called "goodness value" (from the [0,1] interval) to each node in the ADG that expresses how good or bad (1 is the best) is the system regarding that quality attribute. We used the particular ADG presented in Figure 2 for assessing the maintainability of the selected subject systems. The informal definition of the referenced low-level metrics are described in Subsection 3.1.

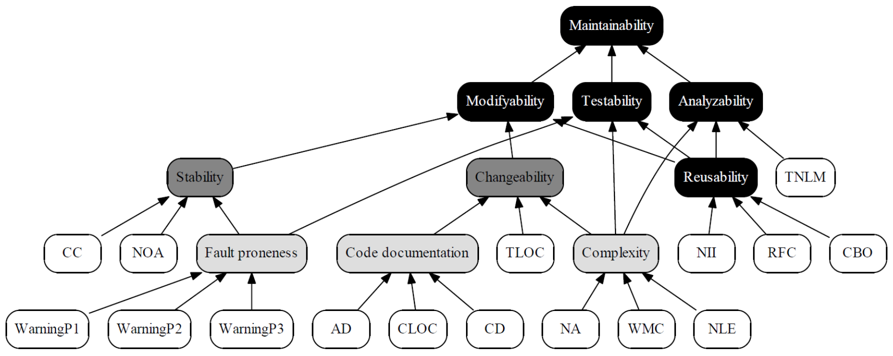

Fig. 2. The quality model used to calculate maintainability

3.4 PROMISE

PROMISE [13] is an open-access bug database whose purpose is to help software quality related research and make the referenced experiments repeatable. It contains the code of numerous open-source applications or frameworks and their corresponding bug-related data on the class level – i.e. not just system aggregates. We extracted this class level bug information for 34 systems from it that will be used for answering our third research question in Section 4.

3.5 Machine Learning

From the class level table of data we wanted to find out if the numbers of different antipatterns have any underlying structure that could help in identifying which system classes have bugs. 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 data – its training set – to be able to correctly label unclassified instances later. As empirically they perform best in similar cases, we chose decision trees as the method of analysis, specifically J48, an open-source implementation of C4.5 [14]. We preformed the actual learning with Weka [6].

4 Results

We analyzed the 228 subject systems and calculated the measures introduced in Section 3. For the first two research questions concerned with finding correlation, we compiled a system level database of the maintainability values, numbers of antipatterns and the numbers of bugs. As the bug-related data was available for every class in the corresponding 34 systems, we aggregated these values to fit in on a per system basis. The resulting dataset can be seen in Table 1, sorted in ascending order by the number of bugs.

After this, we performed correlation analysis on the collected data. Since we did not expect the relationship between the inspected values to be linear – only monotone –, we used the Spearman correlation. The Spearman correlation is in fact a “traditional” Pearson correlation, only it is carried out on the ordered ranks of the values, not the values themselves. This shows how much the two data sets “move together.” The extent of this matching movement is somewhat masked by the ranking – which can be viewed as a kind of data loss – but this is not that important as we are more interested in the *existence* of this relation rather than its type.

When we only considered the systems that had related bug information, we found that the total number of bugs and the total number of antipatterns per system has a Spearman correlation of **0.55** with a p-value below 0.001. This can be regarded as a significant relation that answers our **first research question** by suggesting that the more antipatterns there are in the source code of a software system, the more bugs it will likely contain. Intuitively this is to be expected but with this empirical experiment we are a step closer to being able to treat this belief as a fact. The discovered relation is, of course, not a one-to-one compliance but it illustrates the negative effect of antipatterns on the source code well.

If we disregard the bug information but expand our search to all the 228 analyzed systems, we can inspect the connection between the number of antipatterns and the computed maintainability values. Here we found that there is an even stronger, **-0.62** reverse Spearman correlation, also with a p-value less

Table 1. The extracted metrics of the analyzed systems

Name	**Nr. of Antipatterns**	**Maintainability**	**Nr. of Bugs**
jedit-4.3	2,351	0.47240	12
camel-1.0	685	0.62129	14
forrest-0.7	53	0.73364	15
ivy-1.4	709	0.49465	18
pbeans-2	105	0.48909	19
synapse-1.0	398	0.58202	21
ant-1.3	933	0.51566	33
ant-1.5	2,069	0.41340	35
poi-2.0	2,025	0.36309	39
ant-1.4	1,218	0.45270	47
ivy-2.0	1,260	0.44374	56
log4j-1.0	224	0.59301	61
log4j-1.1	341	0.56738	86
Lucene	3,090	0.47288	97
synapse-1.1	717	0.56659	99
jedit-4.2	1,899	0.46826	106
tomcat-1	5,765	0.30972	114
xerces-1.2	2,520	0.13329	115
synapse-1.2	934	0.55554	145
xalan-2.4	3,718	0.15994	156
ant-1.6	2,821	0.38825	184
velocity-1.6	614	0.43804	190
xerces-1.3	2,670	0.13600	193
jedit-4.1	1,440	0.47400	217
jedit-4.0	1,207	0.47388	226
lucene-2.0	1,580	0.47880	268
camel-1.4	2,136	0.62682	335
ant-1.7	3,549	0.38340	338
Mylyn	5,378	0.65841	340
PDE_UI	7,523	0.53104	341
jedit-3.2	1,017	0.47523	382
camel-1.6	2,804	0.62796	500
camel-1.2	1,280	0.63005	522
xalan-2.6	11,115	0.22621	625

than 0.001. Based on this observation, we can also answer our **second research question**: the more antipatterns a system contains the less maintainable it will be, meaning that it will most likely cost more time and resources to execute any changes. This result corresponds to the definitions of antipatterns and maintainability quite well as they suggest that antipatterns – the common solutions to design problems that seem beneficial but cause more harm than good on the long run – really do lower the expected maintainability – a value representing how easily a piece of software can be understood and modified, i.e. the "long run" harm.

This relation is visualized in Figure 3. The analyzed systems are sorted in the descending order of their contained antipatterns and the trend line of the maintainability value clearly shows improvement.

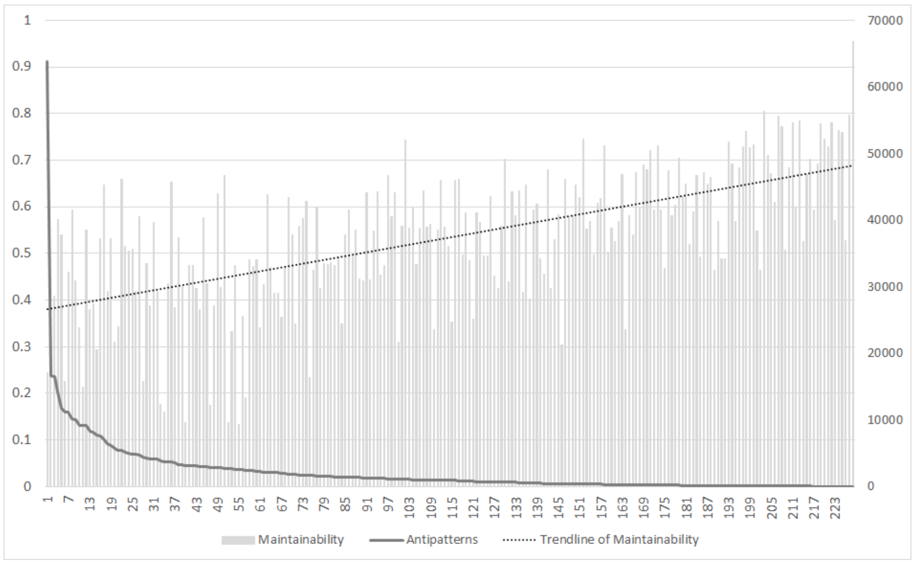

Fig. 3. The trend of maintainability in case of decreasing antipatterns

To answer our third and final research question, we could compile an even finer grained set of data. Here we retained the original, class level form of bug-related data and extracted class level source code metrics. We also needed to transform the antipattern information to correspond to classes so instead of aggregating them to a system level we kept class antipatterns unmodified while collecting method and attribute based antipatterns to their closest parent classes. Part of this data set is shown in Table 2.

The resulting class level data was largely biased because – as it is to be expected – more than 80% of the classes did not contain any bugs. We handled this problem by subsampling the bugless classes in order to create a normally distributed starting point. Subsampling means that in order to make the results

Table 2. Part of the compiled class level data set

System	Name	Bugs	LLOC	TNOS	...	ALL	FE	LC	LCC	LCD	LF	LPL	RB	SHS	TF
ant-1.3	org.apache.tools.ant.AntClassLoader	2	230	124	...	4	0	0	0	0	1	0	0	0	3
ant-1.3	org.apache.tools.ant.BuildEvent	0	51	21	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.BuildException	0	63	27	...	6	0	0	0	0	0	0	0	6	0
ant-1.3	org.apache.tools.ant.DefaultLogger	2	74	30	...	7	0	0	0	0	0	0	6	0	1
ant-1.3	org.apache.tools.ant.DesirableFilter	0	20	11	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.DirectoryScanner	0	472	353	...	25	0	0	0	0	3	0	19	1	2
ant-1.3	org.apache.tools.ant.IntrospectionHelper	2	229	142	...	2	0	0	0	0	1	0	0	0	1
ant-1.3	org.apache.tools.ant.Location	0	29	13	...	1	0	0	0	0	0	0	0	0	1
ant-1.3	org.apache.tools.ant.Main	1	364	254	...	3	0	0	0	0	2	0	0	0	1
ant-1.3	org.apache.tools.ant.NoBannerLogger	0	21	8	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.PathTokenizer	0	37	16	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.Project	1	710	406	...	37	0	0	1	0	2	0	0	9	25
ant-1.3	org.apache.tools.ant.ProjectHelper	3	151	267	...	2	0	0	0	0	1	0	0	0	1
ant-1.3	org.apache.tools.ant.RuntimeConfigurable	2	45	18	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.Target	1	103	42	...	1	0	0	0	0	0	0	0	1	0
ant-1.3	org.apache.tools.ant.Task	0	64	19	...	15	0	0	0	0	0	0	8	6	1
ant-1.3	org.apache.tools.ant.TaskAdapter	0	25	13	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.taskdefs.Ant	0	133	87	...	0	0	0	0	0	0	0	0	0	0
ant-1.3	org.apache.tools.ant.taskdefs.AntStructure	0	179	134	...	1	0	0	0	0	1	0	0	0	0
ant-1.3	org.apache.tools.ant.taskdefs.Available	0	101	46	...	0	0	0	0	0	0	0	0	0	0

of the machine learning experiment more representative, we used only part of the data related to those classes that did not contain bugs. We created a *sub*set by randomly *sampling* – hence the name – from the bugless classes so that their number becomes equal to the "buggy" classes. This way the learning algorithm could not achieve a precision higher than 50% just by choosing one class over the other. We then applied the J48 decision tree – mentioned in Section 3 – in three different configurations:

- using only the different antipattern numbers as predictors
- using only the object-oriented metrics extracted by the Columbus tool as predictors
- using the attributes of both categories

Every actual learning experiment was performed with a ten-fold cross validation. We tried to calibrate the built decision trees to have around 50 leaf nodes and around 100 nodes in total. This is an approximately good compromise between under- and overlearning the training data. The results are summarized in Table 3.

Table 3. The results of the machine learning experiments

Method	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Measure**
Antipatterns	0.658	0.342	0.670	0.658	0.653
Metrics	0.711	0.289	0.712	0.711	0.711
Both	0.712	0.288	0.712	0.712	0.712

These results clearly show that although antipatterns are – for now – inferior to OO metrics in this field, even a few patterns concerned with single entities only can approximate their bug predicting powers quite well. We note that it is expected that the "Both" category does not improve upon the "Metrics" category because in most cases – as of yet – the implemented antipatterns can be viewed as predefined thresholds on certain metrics. With this we answered our **third research question** as antipatterns can already be considered valuable bug predictors and with more implemented patterns – spanning multiple code entities – and heavier use of the contextual structural information they might even overtake them.

5 Threats to Validity

Similarly to most works, our approach also has some threats to validity. First of all, when dealing with recognizing antipatterns, or any patterns, the accuracy of mining is always a question. To make sure that we really extract the patterns we want, we created small, targeted source code tests that checked the structural requirements and metric thresholds of each pattern. To be also sure that we want to match the patterns we *should* – i.e. those that are most likely to really have a bad effect on the quality of the software –, we only implemented antipatterns that are well-known and well-documented in the literature. This way the only remaining threat factor is the interpretation of those patterns to the LIM model.

Then there is the concern of choosing the correct thresholds for certain metrics. Although they are easily configurable – even before every new inspection –, in order to have results we could correlate and analyze, we had to use some specific thresholds. These were approximated by expert opinions taking into consideration the minimum, maximum and average values of the corresponding metrics. This method can be further improved by implementing statistics based dynamic thresholds which is among our future goals.

Another threat to validity is using our previously published quality model for calculating maintainability values. Although we have done many empirical validations of our probabilistic quality model in previous works, we cannot state that the used maintainability model is perfect. Moreover, as the ISO/IEC 25010 standard does not define 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. These factors are possible threats to validity, but our results and continuous empirical validation of the model proves its applicability and usefulness.

Our results also depend on the assumption that the bug-related values we extracted from the PROMISE database are correct. If they are inaccurate that means that our correlation results with the number of bugs are inaccurate too. But as many other works make use of these data, we consider the possibility of this negligible.

Finally, we have to face the threat that our data is biased or that the results are coincidental. We tried to combat these factors by using different kinds of subject systems, balancing our training data to a normal class distribution before the machine learning procedure and considering only statistically significant correlations.

6 Conclusions

In this paper we presented an empirical analysis of exploring the connection between antipatterns, the number of known bugs and software maintainability. First, we briefly explained how we implemented a tool that can match antipatterns on a language independent model of a system. We then analyzed more than 200 open-source Java systems, extracted their object-oriented metrics and antipatterns, calculated their corresponding maintainability values using our probabilistic quality model and even collected class level bug information for 34 of them. By correlation analysis and machine learning methods we were able to draw interesting conclusions.

On a system level scale, we found that in the case of the 34 systems that also had bug-related information there is a significant positive correlation between the number of bugs and the number of antipatterns. Also, if we disregarded the bug data but expanded our search to all 228 analyzed systems to concentrate on maintainability, the result was an even stronger negative correlation between the number of antipatterns and maintainability. This further supports what one would intuitively think considering the definitions of antipatterns, bugs, and quality.

Another interesting result is that the mentioned 9 antipatterns in themselves can quite closely match the bug predicting power of more than 50 class level object-oriented metrics. Although they – as of yet – are inferior, with further patterns that would span over source code elements and rely more heavily on the available structural information, this method has the potential to overtake simple metrics in fault prediction.

As with all similar works, ours also has some threats to its validity but we feel that it is a valuable step towards empirically validating that antipatterns really do hurt software maintainability and can highlight points in the source code that require closer attention.

Acknowledgments

This research was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP-4.2.4.A/2-11/1-2012-0001 “National Excellence Program”, and by the Hungarian national grant GOP-1.1.1-11-2011-0006.

References

1. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A probabilistic software quality model. In: Software Maintenance (ICSM), 2011 27th IEEE International Conference on. pp. 243–252 (2011)
2. Brown, W.J., Malveau, R.C., McCormick, III, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., New York, NY, USA (1998)
3. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus-reverse engineering tool and schema for c++. In: Software Maintenance, 2002. Proceedings. International Conference on. pp. 172–181. IEEE (2002)
4. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley object technology series, Addison-Wesley (1999)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
6. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. 11(1) (Nov 2009)
7. ISO/IEC: ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. ISO/IEC (2005)
8. Khomh, F., Di Penta, M., Guéhéneuc, Y.G.: An exploratory study of the impact of code smells on software change-proneness. In: Reverse Engineering, 2009. WCRE ’09. 16th Working Conference on. pp. 75–84 (2009)

9. Lozano, A., Wermelinger, M., Nuseibeh, B.: Assessing the impact of bad smells using historical information. In: Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting. pp. 31–34. IWPSE '07, ACM (2007)
10. Mäntylä, M., Vanhanen, J., Lassenius, C.: Bad smells - humans as code critics. In: Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. pp. 399–408 (2004)
11. Marinescu, R.: Detecting design flaws via metrics in object-oriented systems. In: In Proceedings of TOOLS. pp. 173–182. IEEE Computer Society (2001)
12. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: In Proc. IEEE International Conference on Software Maintenance (2004)
13. Menzies, T., Caglayan, B., He, Z., Kocaguneli, E., Krall, J., Peters, F., Turhan, B.: The promise repository of empirical software engineering data (June 2012), `http://promisedata.googlecode.com`
14. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
15. Rapu, D., Ducasse, S., Girba, T., Marinescu, R.: Using history information to improve design flaws detection. In: Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on. pp. 223–232 (2004)
16. Trifu, A., Marinescu, R.: Diagnosing design problems in object oriented systems. In: Proceedings of the 12th Working Conference on Reverse Engineering. pp. 155–164. WCRE '05, IEEE Computer Society (2005)