Acta Cybernetica **23** (2017) 471–490.

The Connection between Antipatterns and Maintainability in Firefox

Dénes Bán[a]

Abstract

The notion that antipatterns have a detrimental effect on source code maintainability is widely accepted, but there is relatively little objective evidence to support it. We seek to investigate this issue by analyzing the connection between antipatterns and maintainability in an empirical study of Firefox, an open source browser application developed in C++.

After extracting antipattern instances and maintainability information from 45 revisions, we looked for correlations to uncover a connection between the two concepts. We found statistically significant negative values for both Pearson and Spearman correlations, most of which were under -0.65. These values suggest there are strong, inverse relationships, thereby supporting our initial assumption that the more antipatterns the source code contains, the harder it is to maintain.

Lastly, we combined these data into a table applicable for machine learning experiments, which we conducted using Weka [10] and several of its classifier algorithms. All five regression types we tried had correlation coefficients over 0.77 and used mostly negative weights for the antipattern predictors in the models we constructed.

In conclusion, we can say that this empirical study is another step towards objectively demonstrating that antipatterns have an adverse effect on software maintainability.

Keywords: static analysis, source code metrics, antipatterns, maintainability, correlation, machine learning

1 Introduction

In the area of source code analysis, there are many topics that are intensively studied. One of them is pattern recognition. Antipatterns are common solutions to frequently occurring problems which are supposed to incur decidedly negative consequences. However, even for the most widespread and universally accepted antipatterns, there is no substantial objective evidence that confirms their detrimental effects. To address this, we propose an empirical study intended to improve

[a]University of Szeged, E-mail: zealot@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.3

our understanding of the connection between antipatterns and source code maintainability.

As our subject systems, we selected 45 evenly distributed sample revisions taken from the `master` and `electrolysis` branches of Firefox between 2009 and 2010 – approximately one revision every two weeks. These revisions provided the basis for both antipattern detection and maintainability assessment. We extracted the occurrences of 9 different antipattern types and summed the number of matches by type. We also divided these sums by the total number of logical lines of the subject system for each revision to create new, system-level antipattern density predictor metrics.

Next, we computed corresponding maintainability values using a C++ specific quality model that calculates increasingly more abstract source code characteristics by performing a weighted aggregation of lower level metrics according to the ISO/IEC 25010 standard [14]. Its final result is a number between 0 and 1, which indicates the maintainability of the source code. Moreover, we adapted versions of the independent Maintainability Index [5] to get a secondary quality indicator.

With these data available, we attempted to answer the following two research questions:

- **RQ1: How does the number of antipatterns in a given system correlate with its maintainability?**
- **RQ2: Can the antipattern instances of a system be used to predict its maintainability?**

The paper is structured as follows. In Section 2, we list some related work, then in Section 3 we elaborate on our methodology. In Section 4, we discuss the results we obtained, then in Section 5 we overview some factors that might threaten the validity of these results. Lastly, in Section 6 we draw some pertinent conclusions and outline our plans for future work.

2 Related Work

Maintainability Trying to quantify complex software systems with a single maintainability index is not a new idea. Peercy [20] attempted to characterize subject systems using questionnaires as early as 1981. This, however, was a manual and mostly subjective effort.

Automatic source code analysis and metric extraction later led to metric-based maintainability models. One of the earlier – and more well-known – ones is the Maintainability Index metric (MI) published by Coleman et al. [5], which is a predefined formula that uses specific source code metrics to provide its result. Since it is still widely used to this day, we also included it in our investigations.

With the publication of the ISO/IEC 9126 framework [13], the expected structure and aspects of quality (and maintainability) models were more formally defined. It prescribes how to perform a weighted aggregation of objective, low-level

source code characteristics so it can obtain increasingly abstract values, thereby providing a high-level overview of the whole system. This aggregation is simply visualized by a graph whose leaf nodes are the source code metrics and the most abstract characteristic (in our case, the maintainability) is the root node. An example of this approach in practice is given by Antonellis et al. [2]. Similar to our approach, they also use expert opinion-based graph weighting, but they achieve it by using a technique called Analytical Hierarchical Processing. They conclude that this method helps domain experts to find connections between individual metrics and global maintainability as well as identify problematic areas.

Another example of the ISO/IEC 9126 framework in action is the probabilistic quality model published by Bakota et al. [3]. It also aggregates low-level metrics to arrive at the more abstract maintainability but instead of concrete "goodness values", it makes use of "goodness functions", and the leaf nodes of the dependency graph are treated as random variables. These goodness functions are built by analyzing a benchmark containing over 100 subject systems.

Our main approach uses the ISO/IEC 25010 [14] standard (a successor to ISO/IEC 9126) combined with the source code normalization method presented in [21]. While the model built there was meant for parallelization, not maintainability, the same principles apply. This notion is elaborated on in Section 3.3.

Antipatterns The two antipattern detection strategies closest to our own were published by Marinescu [18] and Moha et al. [19]. These studies both use source code metrics and threshold analysis and they both feature externally parametrized antipattern rules – but in these cases, the structure of the pattern is also customizable. Moreover, Moha et al. also utilize non-metric based, structural or even lexical cues which, although they cannot be so easily customized, have also been incorporated into our approach.

If preexisting benchmarks with known antipattern occurrences are available, machine learning becomes a viable option. Khomh et al. [15] built on the methodology of Moha et al. by making the decisions among parts of a complex ruleset more fuzzy with Bayesian networks. Another example was published by Maiga et al. [17], where they used Support Vector Machines to train models based on source code metrics to recognize antipattern instances. Here, however, we build machine learning models just to analyze the connection between the precomputed antipatterns and the maintainability of a given system.

In yet another approach, Stoianov and Şora [23] reduced pattern recognition to the resolution of logical predicates using Prolog. While this may seem radically different, there are similarities with our technique if we treat our metric thresholds and structural checks as the predicates and the programmatic source code traversal as Prolog's internal resolution process.

The Connections between Antipatterns and Maintainability As we mentioned earlier, little research has been done so far on finding an explicit connection between antipatterns and maintainability. One of these is our previous study [4],

where the two concepts were inversely related, while antipatterns were proportionately related to program faults (or bugs). Another is an investigation by Fontana and Maggioni [8] where they assume the connection and use antipatterns as well as source code metrics to evaluate software quality. Yet another is an empirical study by Yamashita and Moonen [25] where, after the refactoring of 4 Java systems, they conclude that antipatterns could provide experts and developers with more insights into maintainability than source code metrics or subjective judgment alone; however, a combined approach is suggested.

If we broaden our search from maintainability to include other concepts, antipatterns have been linked (among others) to:

- Comprehension by Abbes et al. [1], who concluded that, although single instances can be managed, multiple antipattern occurrences could have a significant impact and should be avoided,
- Class change- and fault-proneness by Khomh et al. [16], who concluded that classes participating in antipatterns are more change- and fault-prone, and
- Unit testing effort by Sabane et al. [22], who concluded that antipattern classes require substantially more test cases and should be tested with additional care.

On the other hand, if we just focus on maintainability, it has been positively linked to design patterns by Hegedűs et al. [11], refactorings by Szőke et al. [24], and version history metrics by Faragó et al. [6].

3 Methodology

The sequence of steps we took in order to answer our research questions is depicted in Figure 1.

The grey circles represent the different artifacts that exist between the steps of the process, while the white rectangles – which are explored in their own subsections – are the steps themselves.

3.1 Analysis

The analysis was conducted using a shell script that enumerated the 45 Firefox revisions, checked out the corresponding repositories (if not yet available) and updated them to the correct commit before initiating a build sequence. The core of the analysis was performed using the SourceMeter tool [7] developed at the Software Engineering Department of the University of Szeged.

It should be mentioned here that apart from the simple build script of `make -f client.mk`, our custom analysis configuration contained filters to skip the results of every command that matched the word "conftest" (a so-called hard filter) and to later skip any source code elements whose source code path information matched the filters described in Listing 1 (a so-called soft filter). These filters were obtained

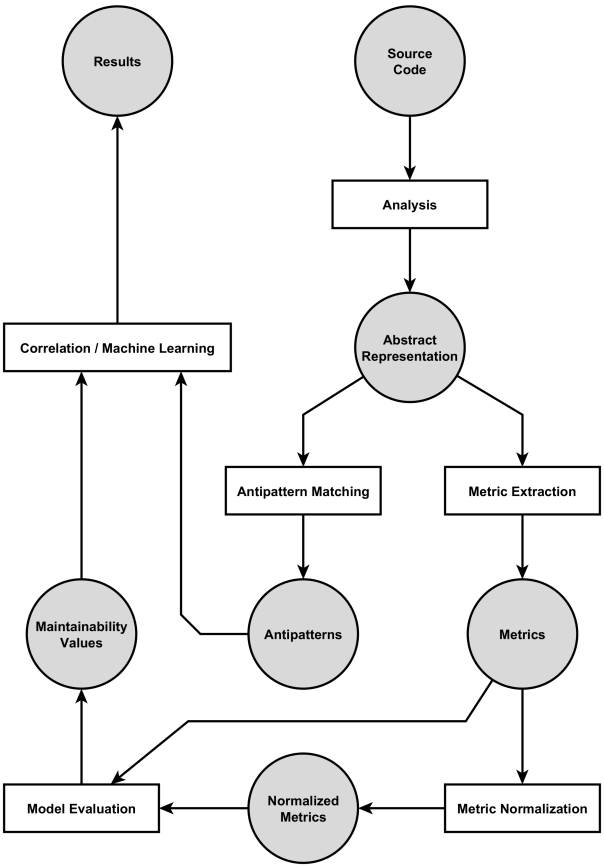

Figure 1: The methodology step sequence

via manual analysis of the Firefox repositories, pinpointing irrelevant or 3rd party code.

The lines in Listing 1 are applied in the order shown, allowing or disallowing a path based on the starting `+` or `-` character. So, for example, the first two lines mean that everything is filtered except for any content coming from the "repos" directory.

It should be added that the 45 Firefox revisions we selected are a subset of the Green Mining Dataset collected by Abram Hindle [12], as it is also our intention to relate antipatterns and software quality to energy and power consumption in the future.

Listing 1: The filter file for the analysis

```
-/
+/path/to/repos/
-/config/
-/testing/
-/build/
-/media/
-/security/
-/db/
-/jpeg/
-/modules/
+/path/to/repos/.*/modules/plugin/
+/path/to/repos/.*/modules/staticmod/
```

3.2 Metric Extraction

After performing our analysis, we extracted the metrics of the global namespace, which represent an aggregated, top-level view of the subject system. These metrics are the following:

- **HVOL** (**H**alstead **VOL**ume): if we let η_1 denote the number of distinct operators, η_2 the distinct operands, N_1 the total number of operators and N_2 the total number of operands, then $HVOL = N_1 + N_2 \cdot log_2(\eta_1 + \eta_2)$. From a C++ perspective, we will treat unary and binary operators (both arithmetic, increment, comparison, boolean, assignment, bitwise, shift and compound), keywords (e.g., return, sizeof, if, else, etc.), brackets, braces, parentheses, semicolons and pointer asterisks as operators, while the corresponding types, names, members, constants and literals will be treated as operands. Although this metric is usually used for single methods, it can be easily generalized to the system level.

- **TCBO** (**T**otal **C**oupling **B**etween **O**bjects): the CBO metric for a class means the number of different classes that are directly used by the class. Usage, among others, includes method calls, parameters, instantiations and attribute accesses as well as returnable and throwable types. **TCBO** is an aggregation of class-level CBOs to the system level, while **AvgCBO** (Average CBO) is defined as the ratio $TCBO/TNCL$.

- **TLCOM5** (**T**otal **L**ack of **CO**hesion in **M**ethods 5): for a class, LCOM5 measures the lack of cohesion, and it is interpreted as how many coherent classes the class could be split into. It is calculated by taking a non-directed graph, where the nodes are the implemented local methods of the class and there is an edge between two nodes if and only if a common attribute or abstract method is used or a method invokes another. The value of the metric

is the number of connected components in the graph not counting those which contain only constructors, destructors, getters or setters. TLCOM5 is the sum of LCOM5s, while **AvgLCOM5** (Average LCOM5) is defined as the ratio $TLCOM5/TNCL$.

- **TRFC** (**T**otal **R**esponse set **F**or **C**lass): for a class, RFC is the number of local (i.e. not inherited) methods in the class plus the number of directly invoked other methods by its methods or attribute initializations. For the system, TRFC is the aggregated sum of RFCs, while **AvgRFC** (Average RFC) is defined as the ratio $TRFC/TNCL$.
- **TWMC** (**T**otal **W**eighted **M**ethods per **C**lass): the WMC metric for a class is the total of the McCC (McCabe's Cyclomatic Complexity) metrics of its local methods. For the system, TWMC is the sum of all WMCs, while **AvgWMC** (Average WMC) is defined as the ratio $TWMC/TNCL$.
- **TAD** (**T**otal **A**PI **D**ocumentation): the ratio of the number of documented public members of the system over the number of all of its public members.
- **TCD** (**T**otal **C**omment **D**ensity): the ratio of the comment lines of the system (TCLOC) over the sum of its comment (TCLOC) and logical lines of code (TLLOC).
- **TCLOC** (**T**otal **C**omment **L**ines **O**f **C**ode): the number of comment and documentation code lines in the system, where comment lines are lines that have either a block or a line comment, while a documentation comment line is a line that has (at least part of) a comment that is syntactically directly in front of a member. Note that a single line can be both a logical line *and* a comment line if it has both code and at least one comment.
- **TLLOC** (**T**otal **L**ogical **L**ines **O**f **C**ode): the number of code lines of the system, without the empty and purely comment lines.
- **TNA** (**T**otal **N**umber of **A**ttributes): the number of attributes in the system.
- **TNCL** (**T**otal **N**umber of **C**lasses): the number of classes in the system.
- **TNEN** (**T**otal **N**umber of **E**nums): the number of enums in the system.
- **TNIN** (**T**otal **N**umber of **I**nterfaces): the number of interfaces in the system. Note that although C++ lacks language support for the concept, we will treat classes with only pure virtual methods as interfaces.
- **TNM** (**T**otal **N**umber of **M**ethods): the number of methods in the system.
- **TNPKG** (**T**otal **N**umber of **P**ac**K**a**G**es): the number of namespaces in the system. Note that the word "package" here refers to a generalized object-oriented container concept which, in C++, directly maps to namespaces.

- **TNOS** (**T**otal **N**umber **O**f **S**tatements): the number of statements in the system.

It should be mentioned that the SourceMeter tool [7] did not have native support for some of the system-level metrics, including the Total and Average versions of CBO, WMC, LCOM5 and RFC, along with the aggregated Halstead Volume. The implementation of these computations was performed specifically for this study.

3.3 Metric Normalization

The metrics we have calculated so far may be viewed as complete from the perspective of the subject systems, but they cannot be related. They are, in a sense, absolute metric values and we have no way to tell, for instance, what an average WMC of 49.6 or a comment density of .31 *means* compared to each other. For this reason, it is desirable to normalize each metric value to the $[0, 1]$ interval using empirical cumulative distribution functions (or ECDFs). This method produces relative numeric values which indicate the ratio of how many of the available data points are smaller than a certain metric. These values are relative because they depend on the context they were evaluated in.

Let $(v_1, v_2, \ldots, v_n)$ be independent and identically distributed random variables with a common distribution function. The empirical distribution function is $\widehat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} I(v_i \leq x)$, where I is the indicator function; namely, $I(v_i \leq x) = 1$ if $v_i \leq x$ and 0 otherwise. For example, the empirical distribution function of variables 1, 1, 1, 1, 2, 2, 4, 4, 5, 5, 6, 6, 8, 9, 13, and 15 can be seen in Figure 2.

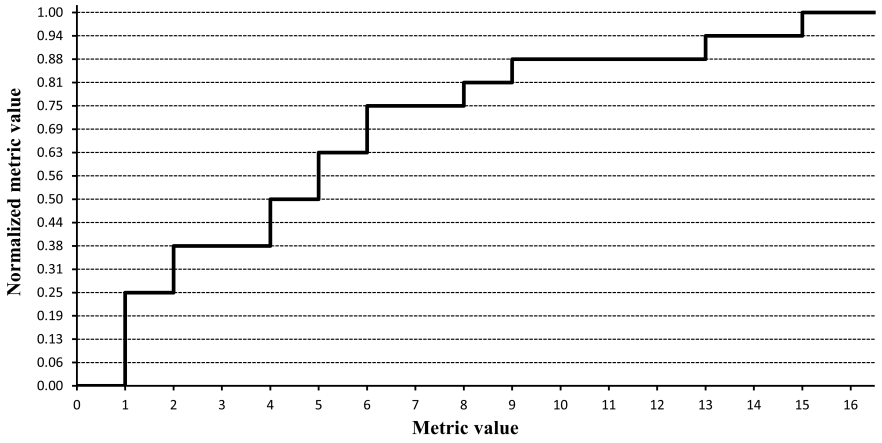

Figure 2: An example Empirical Cumulative Distribution Function [21]

Note that these normalized metrics will be greater for greater absolute inputs and smaller for smaller ones. However, the majority of our examined metrics are "the smaller the better", hence to facilitate a simpler mental model where values

closer to 1 mean *better* quality, we decided to invert this relation. The exceptions to this inversion – or, the "the bigger the better" metrics – are all documentation-related, namely TAD, TCD and TCLOC.

3.4 Antipatterns

For the automatic recognition of antipattern instances, we used our previously published static source code analyzer tool [4]. It currently recognizes the 9 types of antipatterns listed below. They are described in greater detail by Fowler and Beck [9], and here we will just provide a short informal definition and explain how we interpreted them in the context of our model.

Each antipattern implementation can define one or more externally configurable parameters, mostly used for easily adjustable metric thresholds. These are denoted by a starting **$** sign and their default values are listed in Table 1. Several of the object-oriented source code metrics referenced below coincide with the extracted metrics described in Section 3.2, while others are briefly explained in place.

- **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. It is 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", just delegates its requests to other connected classes – i.e., a non-complex class with numerous connections. It is 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 it does too much. It is 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 might be a part of the original class through aggregation or association. It is interpreted as a class whose NA metric is at least **$**MinNA.

- **Long Function** (LF): Similar to LCC, if a method is too long, it probably has parts that could (or should) be separated into their own logical entities, thereby making the whole system more comprehensible. It is interpreted as a method where the LLOC, NOS or McCC metric exceeds **$**MinLLOC, **$**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. It is 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," through which the parent states that descendants *should* most likely use it – then it is a sign that inheritance might not be the appropriate method of implementation reuse. It is 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 a demand for many other – especially remote – modifications, otherwise one of those can easily be missed leading to bugs. It is interpreted as a method whose NII (Number of Incoming Invocations, i.e., the number of the different methods or attribute initializations where this method is called) metric is at least **$**MinNII.

- **Temporary Field** (TF): If an attribute only "makes sense" to a small percentage of the container class then it – and its closely related methods – should be decoupled. It is interpreted as an attribute that is only referenced by at most **$**RefMax% of the members of its container class.

It should be mentioned that in addition to these single antipatterns, we collected a "SUM" value, which is – not surprisingly – defined as the sum of all types of antipatterns in the given subject system. Also, we calculated densities for each absolute antipattern listed above, meaning that for every "AP" antipattern there is an "AP_DENS" metric available, computed as the ratio $AP/TLLOC$.

Table 1: Antipattern default thresholds

Antipattern	**Parameter**	**Value**
FE	MinAccess	5
FE	MinForeign%	80%
LC	MinCBO	5
LC	MaxWMC	10
LCC	MinLLOC	500
LCD	MinNA	30
LF	MinLLOC	80
LF	MinNOS	80
LF	MinMcCC	10
LPL	MinParams	7
SHS	MinNII	10
TF	RefMax%	10%

3.5 Maintainability Models

In order to assess the maintainability of the systems we analyzed, we created an expert opinion-based maintainability model according to the ISO/IEC 25010 standard [14]. The standard states that Maintainability should be the weighted aggregation of 5 subcharacteristics, these being Analysability, Modifiability, Testability, Modularity, and Reusability. The weights of how they are to be aggregated – and how they themselves are computed from source code metrics – were derived from the results of a poll.

First, the 10 experts – each of whom is an academic or industrial professional with at least 5 years of experience in software engineering – had to distribute 100 points among the source code metrics (listed in Section 3.2) for each subcharacteristic, to express how much they think that metric affects the given subcharacteristic. The results of this step are summarized in Table 2.

Table 2: The results of the subcharacteristic votes

ID	Analysability	Modifiability	Testability	Modularity	Reusability
HVOL	25	26.6	25.7	11	13
AvgCBO	22	29.6	24.5	43	33
AvgLCOM5	6.5	4.7	7.4	8.5	7.5
AvgRFC	1.5	3	3.3	15	7.5
AvgWMC	10	10	10.8	5.5	9
TAD	7.3	7	2.5	3.3	7.9
TCBO	1	1.3	3.6	0	1
TCD	4.1	1.5	1.5	0	2
TCLOC	0.3	0.5	0	0	4.5
TRFC	0	0.5	1	0.5	0.5
TWMC	1	1	0	0	0
TLLOC	14.5	9.2	8.5	0	2.4
TNA	0	0	0.2	0	0
TNCL	5	3	5	0	0
TNM	1.4	1.3	3.1	0	0
TNOS	0	0	1	0	0
TNPA	0	0	0	0	1.4
TNPCL	0	0	0	4.6	2.4
TNPEN	0	0	0	0.4	1.3
TNPIN	0	0	0	5.7	3.5
TNPKG	0.4	0.8	0	0	0.2
TNPM	0	0	1.9	2.5	2.9

Next, they had to distribute another 100 points among the subcharacteristics themselves, expressing how much each of them affects the overall Maintainability. The results of this step are summarized in Table 3.

Table 3: The results of the Maintainability votes

Subcharacteristic	**Maintainability**
Analysability	28.5
Modifiability	26.5
Testability	14.2
Modularity	17.1
Reusability	13.7

Given these weights – and later dividing by 100 – we were able to obtain system-level Maintainability values for each of the given subject revisions in the $[0, 1]$ interval.

In addition, as mentioned in Section 1, we computed the two "traditional" Maintainability Index metrics [5], interpreting them using our static source code metrics as:

$$MI = 171 - 5.2 \cdot ln(HVOL) - 0.23 \cdot TWMC - 16.2 \cdot ln(TLLOC)$$

and

$$MI_2 = 171 - 5.2 \cdot log_2(HVOL) - 0.23 \cdot TWMC \\ - 16.2 \cdot log_2(TLLOC) + 50 \cdot \sin(\sqrt{2.4 \cdot TCD})$$

We also calculated their modified counterparts (MI^* and $MI_2{}^*$), where we changed the Total WMC values to their corresponding averages. We did so to scale each part of the sum to the same magnitude because complexity (WMC) is the only component not inside a logarithm or sine and the TWMC values dominated every other term of the formulas.

3.6 Correlations and Machine Learning

Correlation is a statistical relationship between two sets of data denoting the strength of their dependence. Its values can range from +1 (strong relationship) to -1 (strong inverse relationship). We tested both Pearson's (which expresses linear dependence) and Spearman's correlation (which is a Pearson's correlation performed on the rankings of the original data).

Regression analysis is another statistical approach for estimating the relationships among variables. It seeks to predict how the typical value of the dependent

variable changes when any one of the independent variables change. It achieves this by providing an estimate for the dependent variable from a continuous interval. Its most important performance measure is its correlation coefficient, expressing how well the predicted values follow the tendency of the real value of the dependent variable.

In our current case, the dependent variable was one of the maintainability metrics (our ISO/IEC 25010-based Maintainability or one of the previously mentioned MI versions), while the independent variables were the counts and densities of the different antipattern types.

In this empirical study, we evaluated each of the following regression types: linear regression, multilayer perceptron, reduced error pruning tree, M5P tree and sequential minimal optimization regression.

4 Results

After all these preliminaries, we are now ready to address our two research questions.

4.1 Correlation Results

To address **RQ1**, we decided to calculate the Pearson and Spearman correlations between each antipattern and maintainability measure pair, summarized in Table 4 and Table 5, respectively. Note that a single star suffix (*) means that the correlation is statistically significant at the .05 level, while a double star (**) means a significance at the .01 level. Also, to help in quickly parsing these tables, any cell where the correlation coefficient is either positive or non-significant was marked in a light gray background, and a darker gray when it is significantly positive (the worst case from our perspective).

As these tables clearly show, most antipattern-maintainability pairs have a strong, significant inverse connection. There are a few marked correlations, mainly for Modularity and Reusability, but even in these cases the non-significant values are still negative, while the positive values are non-significant and weak. We highlight the correlations between the SUM and SUM_DENS antipatterns and our final Maintainability measure as these represent most closely the overall effects of antipatterns on maintainability. The corresponding values are -.658 and -.692 for Pearson, and -.704 and -.678 for Spearman correlation, respectively. Thus, in response to the first research question we conclude – based on these empirical findings – that there is a strong, inverse relationship between the number of antipatterns in a system and its maintainability. This supports our initial assumption that the more antipatterns the source code contains, the harder it is to maintain.

Table 4: Pearson correlations between different antipatterns and maintainability measures

Antipattern	MI	MI_2	MI^*	$MI_2{}^*$	**Analysability**	**Modifiability**	**Testability**	**Modularity**	**Reusability**	**Maintainability**
FE	-.985**	-.985**	-.857**	-.796**	-.830**	-.738**	-.785**	-.141	-.311*	-.657**
FE_DENS	-.659**	-.659**	-.303*	-.219	-.548**	-.637**	-.679**	-.538**	-.570**	-.661**
LC	-.825**	-.825**	-.933**	-.968**	-.732**	-.502**	-.519**	.274	.023	-.371*
LC_DENS	-.749**	-.749**	-.886**	-.943**	-.666**	-.425**	-.435**	.327*	.075	-.297*
LCC	-.987**	-.987**	-.864**	-.822**	-.862**	-.758**	-.791**	-.133	-.326*	-.674**
LCC_DENS	-.670**	-.670**	-.296*	-.249	-.624**	-.700**	-.711**	-.563**	-.638**	-.722**
LCD	-.768**	-.768**	-.555**	-.438**	-.531**	-.554**	-.611**	-.290	-.314*	-.526**
LCD_DENS	-.662**	-.662**	-.424**	-.298*	-.422**	-.483**	-.541**	-.344*	-.325*	-.477**
LF	-.988**	-.988**	-.883**	-.830**	-.885**	-.782**	-.830**	-.174	-.372*	-.710**
LF_DENS	-.820**	-.820**	-.530**	-.455**	-.783**	-.818**	-.866**	-.566**	-.688**	-.835**
LPL	-.961**	-.961**	-.931**	-.872**	-.781**	-.643**	-.704**	.014	-.160	-.544**
LPL_DENS	-.864**	-.864**	-.741**	-.649**	-.652**	-.594**	-.673**	-.157	-.253	-.542**
RB	-.985**	-.985**	-.852**	-.788**	-.820**	-.736**	-.783**	-.165	-.328*	-.662**
RB_DENS	-.930**	-.930**	-.714**	-.634**	-.757**	-.734**	-.786**	-.317*	-.435**	-.695**
SHS	-.988**	-.988**	-.850**	-.778**	-.837**	-.767**	-.811**	-.212	-.375*	-.698**
SHS_DENS	-.889**	-.889**	-.634**	-.538**	-.745**	-.771**	-.815**	-.450**	-.548**	-.754**
SUM	-.982**	-.982**	-.869**	-.791**	-.814**	-.735**	-.785**	-.164	-.319*	-.658**
SUM_DENS	-.910**	-.910**	-.712**	-.606**	-.731**	-.729**	-.783**	-.342*	-.438**	-.692**
TF	-.955**	-.955**	-.835**	-.740**	-.777**	-.723**	-.771**	-.199	-.330*	-.651**
TF_DENS	-.882**	-.882**	-.711**	-.592**	-.691**	-.695**	-.747**	-.317*	-.397**	-.653**

Table 5: Spearman's correlations between different antipatterns and maintainability measures

Antipattern	MI	MI_2	MI^*	MI_2^*	**Analysability**	**Modifiability**	**Testability**	**Modularity**	**Reusability**	**Maintainability**
FE	-.985**	-.985**	-.853**	-.809**	-.852**	-.749**	-.806**	-.161	-.370*	-.652**
FE_DENS	-.535**	-.535**	-.257	-.258	-.440**	-.532**	-.561**	-.550**	-.595**	-.609**
LC	-.757**	-.757**	-.936**	-.920**	-.755**	-.538**	-.572**	.224	-.044	-.381**
LC_DENS	-.542**	-.542**	-.777**	-.854**	-.517**	-.276	-.307*	.372*	.123	-.133
LCC	-.985**	-.985**	-.873**	-.839**	-.871**	-.754**	-.811**	-.131	-.354*	-.651**
LCC_DENS	-.482**	-.482**	-.213	-.258	-.439**	-.543**	-.550**	-.590**	-.655**	-.636**
LCD	-.731**	-.731**	-.484**	-.445**	-.509**	-.551**	-.590**	-.303*	-.365*	-.528**
LCD_DENS	-.626**	-.626**	-.355*	-.344*	-.373*	-.431**	-.474**	-.299*	-.318*	-.428**
LF	-.991**	-.991**	-.849**	-.800**	-.902**	-.821**	-.866**	-.242	-.453**	-.728**
LF_DENS	-.874**	-.874**	-.622**	-.608**	-.824**	-.837**	-.876**	-.500**	-.671**	-.821**
LPL	-.952**	-.952**	-.926**	-.856**	-.851**	-.696**	-.750**	.019	-.219	-.560**
LPL_DENS	-.904**	-.904**	-.707**	-.670**	-.715**	-.654**	-.708**	-.184	-.338*	-.580**
RB	-.976**	-.976**	-.819**	-.793**	-.829**	-.734**	-.794**	-.167	-.375*	-.646**
RB_DENS	-.911**	-.911**	-.706**	-.728**	-.735**	-.674**	-.730**	-.219	-.396**	-.614**
SHS	-.985**	-.985**	-.820**	-.768**	-.884**	-.827**	-.871**	-.291	-.487**	-.747**
SHS_DENS	-.907**	-.907**	-.694**	-.698**	-.787**	-.773**	-.812**	-.370*	-.538**	-.726**
SUM	-.978**	-.978**	-.806**	-.754**	-.847**	-.786**	-.834**	-.250	-.444**	-.704**
SUM_DENS	-.895**	-.895**	-.674**	-.641**	-.732**	-.726**	-.768**	-.332*	-.476**	-.678**
TF	-.945**	-.945**	-.746**	-.704**	-.785**	-.746**	-.796**	-.277	-.445**	-.681**
TF_DENS	-.843**	-.843**	-.595**	-.543**	-.675**	-.704**	-.739**	-.378*	-.484**	-.665**

4.2 Machine Learning Results

To answer **RQ2**, we compiled ten tables applicable for machine learning experiments – one for each maintainability measure. These contained every antipattern type as predictors and the values for their chosen maintainability measures as targets for prediction. We then ran these tables through all five regression techniques mentioned in Section 3.6 to see how well they worked in practice. The resulting models were later tested with a 10-fold cross-validation, and the corresponding correlation coefficients are shown in Table 6.

Table 6: Correlation coefficients of the machine learning models

	Linear Reg.	**MLP**	**REPTree**	**M5P**	**SMO Reg.**
MI	.9991	.9969	.9079	.9983	.9993
MI*	.9825	.9968	.8695	.9727	.9971
MI2	.9991	.9969	.9635	.9983	.9993
MI2*	.9864	.9689	.9033	.9799	.9858
Analysability	.8210	.9085	.7632	.9097	.9151
Modifiability	.8082	.9223	.7286	.8138	.8348
Testability	.8637	.9547	.8564	.8874	.8903
Modularity	.9082	.8915	.7461	.7589	.8757
Reusability	.8247	.8927	.6777	.6222	.8455
Maintainability	.8513	.9318	.7619	.8179	.8556

The high values of these coefficients suggest an affirmative answer to our second research question: antipatterns *can* be valuable predictors for maintainability assessment. The models we built weight the antipattern predictors with mostly negative values, but there are numerous positive instances as well. Further analysis of the structure of the models in the case of the Maintainability target revealed that some antipatterns *consistently* appear with negative weights more often than others. Moreover, this ordering of importance largely coincides with the above correlation magnitudes.

4.3 Lessons Learned

The most obvious lesson learned, based on these results, is the measurable detrimental effect of antipatterns on maintainability. Moreover, the conclusion we drew from the correspondence between correlation values and negative model weights is that there could also be an *order of importance* among the antipatterns studied here.

The most important ones to avoid appear to be Long Functions, Large Class Codes and Shotgun Surgeries. The frequently suggested refactorings for the first

two antipatterns are "Extract Method" and "Extract Class", respectively. As for Shotgun Surgery, the main goal is to reduce coupling by moving or extracting methods or fields, or even identifying a common superclass.

Refused Bequests and Temporary Fields seem less dangerous. The former can be fixed with "Replace Inheritance with Delegation" or by extracting an even more abstract superclass to house just the common members, while the latter is often corrected with "Extract Class" – which can coincide with extracting a method object.

And finally, Long Parameter Lists, Feature Envies, Lazy Classes and Large Class Data instances can be more easily tolerated. However, these can also be eliminated using techniques given in [9]. Long Parameter Lists have "Preserve Whole Object" or "Introduce Parameter Object"; Feature Envy has "Move Method" or "Extract Method"; Lazy Classes may vanish if their functionality is inlined or their connections are introduced to each other without the middle man; and lastly, Large Class Data can be solved – again – with "Extract Class".

The key point of these observations is that developers should concern themselves more forcefully with the organization of source code, and not just its behavior, since the work they put in in advance seems to lead to an easier maintenance phase, while the performance overhead introduced by the extra classes and methods is negligible.

5 Threats to Validity

There are a few aspects that might possibly threaten the validity of our results. One is that the antipattern matches might not be correct. While finding antipattern instances is far from being a solved problem, we tried to acquire reliable statistics by implementing widely recognized antipatterns with usual/recommended threshold values and previously published tooling support.

Imprecise maintainability scores could also skew our results. To combat this, we decided to utilize static, independent source code metrics and expert opinion-based weight determination, all the while adhering to the guidelines of an international standard.

To ensure that the connections we uncover were not just coincidental, we only included statistically significant correlations in this study. The connections could also be attributed to the fact that both the maintainability scores and the antipattern instances are – at least partially – based on the same static source code metrics. Despite the overlap, there are important differences, because the two concepts do not rely on the same aggregation level of metrics (method/class or system level) and antipatterns heavily incorporate other structural cues as well. We would also argue that the results *could* be meaningful even if the base set of metrics were identical, given that the mapping of concepts to metrics is plausible.

Lastly, the generalizability of these findings could be largely affected by the number of subject systems analyzed. Although a benchmark made from 45 versions of such a huge and complex software system can hardly be regarded as small, we intend to include more revisions and different applications as well.

6 Conclusions and Future Work

In this study, we analyzed 45 revisions of Firefox and calculated static source code metrics for each of them. Using these metrics, specific threshold parameters and structural information, we matched 9 types of antipatterns and their respective densities in each revision. Also utilizing these metrics, we calculated maintainability values based on the ISO/IEC 25010 software quality framework. After correlating these two sets of data, we found statistically significant inverse relationships, which we consider another step towards objectively demonstrating that antipatterns have an adverse effect on software maintainability. Moreover, our machine learning experiments indicated that regression techniques can attain high precision in predicting maintainability from antipattern information alone, suggesting that antipatterns can be valuable besides – or even instead of – static source code metrics in software maintainability assessment.

A possible next step for this investigation might be to analyze more Firefox revisions or include other C++ subject systems in another empirical study. Also, our selected Firefox revisions have runtime, power consumption and energy efficiency measurements as part of the Green Mining Dataset [12], and this provides us with the chance to relate antipatterns or maintainability to those concepts, too. We plan to calculate maintainability values at the class level as well (instead of at the system level), thereby – hopefully – gaining a more fine-grained view of how antipattern and non-antipattern classes relate to maintainability.

References

[1] Abbes, Marwen, Khomh, Foutse, Gueheneuc, Yann-Gael, and Antoniol, Giuliano. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Antonellis, P, Antoniou, D, Kanellopoulos, Y, Makris, C, Theodoridis, E, Tjortjis, C, and Tsirakis, N. A data mining methodology for evaluating maintainability according to iso/iec-9126 software engineering–product quality standard. *Special Session on System Quality and Maintainability-SQM2007*, 2007.

[3] Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., and Gyimóthy, T. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243–252, 2011.

[4] Bán, Dénes and Ferenc, Rudolf. *Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability*, pages 337–352. Springer International Publishing, Cham, 2014.

[5] Coleman, D., Ash, D., Lowther, B., and Oman, P. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug 1994.

[6] Faragó, C., Hegedus, P., Ladányi, G., and Ferenc, R. Impact of version history metrics on maintainability. In *2015 8th International Conference on Advanced Software Engineering Its Applications (ASEA)*, pages 30–35, Nov 2015.

[7] Ferenc, Rudolf, Langó, László, Siket, István, Gyimóthy, Tibor, and Bakota, Tibor. Source meter sonar qube plug-in. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 77–82, Washington, DC, USA, 2014. IEEE Computer Society.

[8] Fontana, F. A. and Maggioni, S. Metrics and antipatterns for software quality evaluation. In *Software Engineering Workshop (SEW), 2011 34th IEEE*, pages 48–56, June 2011.

[9] Fowler, M. and Beck, K. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley object technology series. Addison-Wesley, 1999.

[10] Hall, Mark, Frank, Eibe, Holmes, Geoffrey, Pfahringer, Bernhard, Reutemann, Peter, and Witten, Ian H. The WEKA Data Mining Software: An Update. In *SIGKDD Explorations*, volume 11, pages 10–18. ACM, June 2009.

[11] Hegedűs, Péter, Bán, Dénes, Ferenc, Rudolf, and Gyimóthy, Tibor. *Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability*, pages 138–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[12] Hindle, Abram. Green mining: A methodology of relating software change to power consumption. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 78–87. IEEE, 2012.

[13] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[14] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.

[15] Khomh, F., Vaucher, S., Guéhéneuc, Y. G., and Sahraoui, H. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314, Aug 2009.

[16] Khomh, Foutse, Penta, Massimiliano Di, Guéhéneuc, Yann-Gaël, and Antoniol, Giuliano. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[17] Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y. G., and Aimeur, E. Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475, Oct 2012.

[18] Marinescu, Radu. Detection strategies: Metrics-based rules for detecting design flaws. In *In Proc. IEEE International Conference on Software Maintenance*, 2004.

[19] Moha, N., Guéhéneuc, Y. G., Duchien, L., and Meur, A. F. Le. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Jan 2010.

[20] Peercy, D. E. A software maintainability evaluation methodology. *IEEE Transactions on Software Engineering*, SE-7(4):343–351, July 1981.

[21] Rudolf Ferenc et al. *REPARA deliverable D7.4: Maintainability models of heterogeneous programming models.* 2015.

[22] Sabane, A., Penta, M., Antoniol, G., and Guéhéneuc, Y. G. A study on the relation between antipatterns and the cost of class unit testing. In *Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR*, March 2013.

[23] Stoianov, A. and Şora, I. Detecting patterns and antipatterns in software using prolog rules. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 253–258, May 2010.

[24] Szőke, G., Nagy, C., Hegedűs, P., Ferenc, R., and Gyimóthy, T. Do automatic refactorings improve maintainability? an industrial case study. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 429–438, Sept 2015.

[25] Yamashita, Aiko and Moonen, Leon. Do code smells reflect important maintainability aspects? pages 306–315. IEEE, September 2012.