Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects

Rodrigo Morales*, Shane McIntosh†, Foutse Khomh*
*SWAT, Polytechnique Montréal, Canada; {rodrigo.morales, foutse.khomh}@polymtl.ca
†School of Computing, Queen's University, Canada; mcintosh@cs.queensu.ca

Abstract—**Code review is the process of having other team members examine changes to a software system in order to evaluate its technical content and quality. A lightweight variant of this practice, often referred to as Modern Code Review (MCR), is widely adopted by software organizations today. Previous studies have established a relation between the practice of code review and the occurrence of post-release bugs. While the prior work studies the impact of code review practices on software release quality, it is still unclear what impact code review practices have on software design quality. Therefore, using the occurrence of 7 different types of anti-patterns (i.e., poor solutions to design and implementation problems) as a proxy for software design quality, we set out to investigate the relationship between code review practices and software design quality. Through a case study of the Qt, VTK and ITK open source projects, we find that software components with low review coverage or low review participation are often more prone to the occurrence of anti-patterns than those components with more active code review practices.**

I. Introduction

Software system design is critical. Studies suggest that defects relating to system design are orders of magnitude more expensive to fix than those introduced during its implementation [1–4]. Indeed, neglecting to carefully design a software system may result in a highly complex implementation that is difficult to maintain. Moreover, a complex implementation requires a considerable investment of effort to restructure.

To improve the design of software systems, software designers use design patterns [5] (i.e., reusable solutions to recurring design problems) and avoid anti-patterns [6] (i.e., poor solutions to design and implementation problems). In fact, Khomh et al. [7] find that there is a strong correlation between the occurrence of anti-patterns and the change-proneness of source code files. Moreover, Taba et al. [8] and D'Ambros et al. [9] find that source code files that contain anti-patterns tend to be more defect-prone than other source code files.

Like other complex systems, software systems age [10]. As they age, software systems tend to grow in complexity and degrade in effectiveness [11], unless the quality of the systems is controlled and continually improved. Even good design solutions (e.g., design patterns) tends to decay into anti-patterns as systems age [12, 13]. When the design of a system is poor, changes to it often degrade the quality of the system. Aging systems are often defect-prone, and the cost for removing these defects is high [10]. Moreover, the length of time that an anti-pattern can remain in a system is variable, though there is evidence to suggest that they tend to linger for several releases [14, 15].

According to a recent study at Microsoft [16], practitioners expect that *code review*, i.e., the practice of having other team members critique changes to a software system, will help to combat design decay. Prior work has shown that much of the discussion of a code review relates to the evolvability of a module rather than the behaviour of a change [17, 18]. However, the relationship between code review practices and software design quality remains largely unexplored.

In this paper, we study the impact that code review practices have on software design quality. To quantify code review practices, similar to our prior work [19], we use *code review coverage*, i.e., the proportion of changes that have been code reviewed, and *code review participation*, i.e., the degree of reviewer involvement in the code review process. Furthermore, we use the occurrences of 7 well-known anti-patterns (i.e., Code Duplication, Blob, Data Class, Data Clumps, Feature Envy, Schizophrenic Class and Tradition Breaker) as a proxy for software design quality. Using our code review metrics to complement of popular product and process metrics, we train regression models that explain the occurrences of anti-patterns. Through a case study of Qt, VTK and ITK open source projects, we address the following two research questions:

(RQ1) Is there a relationship between code review coverage and the incidence of anti-patterns?
We find that code review coverage has a negative impact on the occurrence of anti-patterns. However, review coverage provides a statistically significant amount of explanatory power in only two of the four studied systems, suggesting that review coverage alone is not enough to ensure a low rate of design issues.

(RQ2) Is there a relationship between code review participation and the incidence of anti-patterns?
Participation during the code review process is also associated with components that have fewer anti-patterns. Indeed, according to our models, there is a high correlation between review participation metrics and the occurrence of anti-patterns.

Paper organization. The remainder of this paper is organized as follows. Section II provides background details and discusses the related work on anti-patterns and code reviews. Section III describes our case study design, while Section IV presents the results. Section V discloses the threats to the validity of our study. Finally, Section VI draws our conclusions and lays out directions for future work.

II. Background and Related Work

In this section, we describe anti-patterns, the code review process, and discuss the related work.

Anti-patterns—such as those defined by Brown et al. [20]—have been proposed to embody poor design choices. These anti-patterns stem from experienced software developers' expertise and are reported to negatively impact systems by making classes more change-prone [21] and defect-prone [7, 8]. They are opposite to design patterns [22], *i.e.*, they identify "poor" solutions to recurring design problems. For example, Brown et al. define 40 anti-patterns that describe the most common recurring pitfalls in the software industry [20]. Anti-patterns are generally introduced in software systems by developers not having sufficient experience in solving a particular problem, or having misapplied some design patterns. Coplien and Harrison [23] described an anti-pattern as "something that looks like a good idea, but which back-fires badly when applied". A common anti-pattern is the Blob, a.k.a., God Class. The Blob is a large and complex class that centralises the behaviour of a portion of a system and only uses other classes as data holders, i.e., data classes. The main characteristic of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields or accessors to their fields. Another example anti-pattern is the MessageChain, which occurs when a class uses a long chain of method invocations to realise (at least) one of its functionalities, causing the class to become coupled to all of the classes involved in this traversal. As a result, a change to any of the intermediate relationships will require a change to the class. Khomh et al. [7] investigated MessageChains in ArgoUML, Eclipse, Mylyn, and Rhino and found them to be consistently related to high defect and change rates.

The literature related to anti-patterns generally falls into three categories: (1) the detection of anti-patterns (e.g., [24, 25]); (2) the evolution of anti-patterns in software systems (e.g., [15, 26, 27]) and their impact on software quality (e.g., [7, 28, 29]); and (3) the relationship between anti-patterns and software development activities (e.g., [29, 30]). Our work in this paper falls into the third category – we aim to understand how code review practices impact the incidence of anti-patterns in software systems. Sjoberg et al. [30], who investigated the relationship between anti-patterns and maintenance effort reported that anti-patterns have a limited impact on maintenance effort. However, Abbes et al. [29] found that combinations of Blob and Spaghetti Code anti-patterns have a negative impact on code understandability. They recommend applying refactoring to remove such interactions of anti-patterns.

Code review—is a widely-adopted technique used to improve the quality of changes to a software system. The typical modern code review process follows the steps described below. First, an author solicits feedback on a proposed set of changes by submitting them to reviewers. Next, reviewers provide comments and suggestions, based on their expertise. After authors address the issues raised by reviewers, they may resubmit an updated set of changes. The process iterates until reviewers are satisfied with the proposed changes.

Prior work has examined code review practices. Rigby et al. [31] described the code review process in the Apache project as early, frequent reviews performed on small amounts of code that are independent of each other and led by a reduced number of experts. Other studies have established that code review is an effective mean to reduce post-release defects in software systems [19, 32, 33].

While identifying and fixing bugs early in the development process is a strong motivation for code review practices, it is not the sole motivation for performing code reviews. In a study performed at Microsoft, Bacchelli and Bird [16] found that the code review process improves team awareness, and transfers knowledge among members of development teams. Mäntylä and Lassenius [34] examined defects corrected during code reviews activities at 9 software companies and found that 75% of these defects had an impact on the understandability of the code.

Gerrit-based code review—To provide an insight of the code review process in the examined projects take the following example: *Bob* uploads a patch (i.e., a collection of proposed changes to a software system) to a Gerrit server. A set of Reviewers are invited to participate in the review process either by: (a) explicit invitation by *Bob*, (b) automatic invitation due to expertise with the modified component(s), or (c) self-invitation based on the reviewers interest in the patch. The reviewers then provide inline or general feedback about the patch to *Bob*, who can then reply to their comments, or address them by producing a new revision of the patch. Reviewers provide a revision score to indicate their agreement/disagreement and their level of confidence (1 or 2).

In addition to reviewers, Gerrit supports the role of patch *verifiers* who validate that *Bob's* patch accomplishes the task it was designed to without introducing regression of system behaviour. *Verifiers* can also provide comments and assign a score to reflect the verification status (+1 success/-1 fail). Furthermore, verifiers can be other team members or bots that automatically build and test patches.

Finally, Gerrit enables teams to set code review and verification criteria that must be met before changes are integrated into upstream VCS repositories. Once these criteria are satisfied, patches are automatically integrated into upstream code repositories.

III. Study Definition and Design

In this paper, we set out to empirically evaluate the link between design quality and code reviews. In this section, we introduce our research questions, describe the studied systems, and present our data extraction approach. Furthermore, we describe our model construction and model analysis approaches.

(RQ1) Is there a relationship between code review coverage and the incidence of anti-patterns?
Prior work shows that code review is an effective means of improving the quality of software systems [19, 32, 33]. However, the prior work focused on

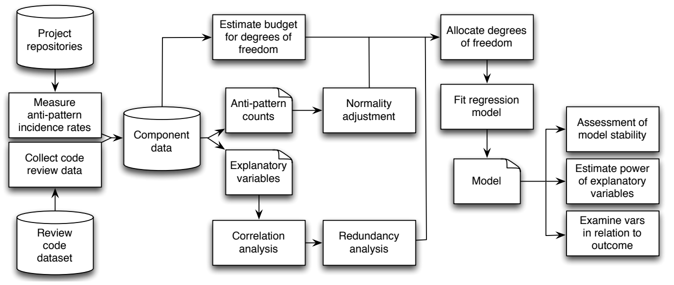
Project repositories
Measure anti-pattern incidence rates
Collect code review data
Review code dataset
Component data
Estimate budget for degrees of freedom
Anti-pattern counts
Normality adjustment
Explanatory variables
Correlation analysis
Redundancy analysis
Allocate degrees of freedom
Fit regression model
Model
Assessment of model stability
Estimate power of explanatory variables
Examine vars in relation to outcome

Fig. 1: An overview of our approach.

software release quality as measured by the incidence of *post-release defects*, i.e., defects that permeate through the quality assurance process to official releases. Yet the impact that code review has on other aspects of software quality, such as design quality, remain largely unexplored. Hence, we first set out to better understand the impact that code review coverage has on software design quality.

(RQ2) Is there a relationship between code review participation and the incidence of anti-patterns?
A code review cannot improve design quality if reviewers are not participating in the review process. Hence, we are interested in studying the relationship between the degree of participation in the review process and software design quality.

A. Studied systems

In order to address our research questions, we perform a case study of 3 open source projects that are primarily implemented in C++. Qt[1] is a cross-platform application and UI framework. VTK[2] is a framework used to develop 3D graphics. ITK[3] is a cross-platform framework and a set of tools for image analysis.

B. Data Extraction

In order to perform our case study, we need to extract data describing the code review process and the incidence of anti-patterns. Figure 1 provides an overview of our approach. We describe each step in our data extraction approach below.

Collect code review data. In order to perform our study, we leverage our previously collected code review datasets [19]. The datasets describe code review activity aggregated at the component (*i.e.*, directory) level. In addition to code review metrics, these datasets includes product, process, and human factors metrics. Table II describes the metrics that we use in our study, and our rationale for including them.

Measure anti-pattern incidence rates. In this study, we use the incidence rates of anti-patterns as a proxy for software design quality. We use the Incode[4] tool to detect anti-patterns in the studied systems. Incode is a mature, commercial design quality tool capable of detecting the following 7 anti-patterns:

[1]http://qt-project.org/
[2]http://www.vtk.org/
[3]http://www.itk.org/
[4]https://www.intooitus.com/products/incode

Table I: Descriptive statistics of the studied systems.

	Qt		VTK	ITK
Version	5.0	5.1	5.10	4.3
Size (LOC)	5,560,317	5,187,788	1,921,850	1,123,614
Components w/anti-patterns	285	143	86	76
Components total	1,164	1,268	129	194
Commits w/reviews	10,003	6,795	554	344
Commits total	10,163	7,106	1,431	352
# Authors	435	422	55	41
# Reviewers	358	348	45	37

1) **Code Duplication**: identical or slightly-modified code fragments.
2) **Blob**: A large class that: (a) is lacking cohesion, (b) monopolizes much of the processing time, (c) makes most of the processing decisions, or (d) is associated with data classes.
3) **Data Class**: A class with an interface that exposes data members, without encapsulating its own functionality.
4) **Data Clumps**: A large group of parameters that appear together in the signature of many methods.
5) **Feature Envy**: A class that uses many methods or data attributes from other classes instead of implementing its own.
6) **Schizophrenic Class**: A class with a large and non-cohesive interface. Typically, schizophrenic classes contain several disjoint sets of public methods that are used by disjoint sets of client classes.
7) **Tradition Breaker**: A class that breaks the interface inherited from a base class or an interface.

We choose to analyze these anti-patterns because: (1) they are well-defined [20], (2) they appear frequently in the studied systems, i.e., between 11% (Qt 5.1) and 66% (VTK 5.10) of components in the studied systems contain at least one of the anti-patterns, and (3) they have been studied in previous works [7, 15, 30].

For each file, we obtain the list of anti-patterns and group them at component (i.e., directory) level. Next, we join the count of anti-patterns with code review data, filtering away files that were not written in a language (*e.g.*, Python, Perl, Javascript) supported by the Incode anti-pattern detection tool. Table I provides an overview of the components that survived our filtering process.

C. Model Construction

To assess the impact that code review has on software design quality, we use Ordinary Least Squares (OLS) multiple regression analysis. Our regression models are trained to explain the number of anti-patterns that occur in a component (the response variable) using the metrics of Table II (explanatory variables).

We use the non-linear regression modeling techniques proposed by Harrell Jr. [37]. These techniques relax linearity assumptions of traditional modeling techniques, allowing the relationship between explanatory variables and the response to change in direction, while being mindful of the threat of *overfitting*, i.e., constructing a model that is too specific to the

Table II: A taxonomy of the considered control (top) and reviewing metrics (bottom).

	Metric	Description	Rationale
Product	Size	Lines of code (LOC).	Large components are hard to maintain and commonly associated with anti-patterns(*e.g.*, Large Class).
	Complexity	The McCabe cyclomatic complexity.	Components with a very high complexity are potential candidates to be split in simpler routines.
Process	Churn	Sum of added and removed lines of code.	Components with anti-patterns are more change-proneness than others [21].
	Change Entropy	A measure of the volatility of the change.	Components that undergo refactoring reduces its Change entropy [35].
Human Factors	Total authors	Number of unique authors.	Components developed by several authors with weak ownership are likely to be defect-prone [36].
	Minor authors	Number of unique authors who have contributed less than 5% of the changes.	Components with several minor contributors may exhibit poor design because of these minor contributors' lack of adequate knowledge about the design intent of the component [36].
	Major authors	Number of unique authors who have contributed at least 5% of the changes.	The design quality of components with Major contributors may benefit from a strong component-specific knowledge from the major contributors [36].
	Author ownership	The proportion of changes contributed by the author who made the most changes.	Components that present notable ownership from very active contributors present less failures [36].
Coverage	Proportion of reviewed changes	The proportion of changes that have been reviewed in the past.	We expect that components with high-review rates will contain less anti-patterns.
	Proportion of reviewed churn	The proportion of churn that has been reviewed in the past.	We assume that the larger is the amount of code churn that undergo code review the higher will be the design quality.
Participation	Proportion of self-approved changes	The proportion of changes to a component that are only approved for integration by the original author.	Changes approved only by their author may not follow all guidelines for a good design.
	Proportion of hastily reviewed changes	The proportion of changes that are approved for integration at a rate that is faster than 200 lines per hour.	There is evidence in previous studies that components whose changes where approved at a rate faster than 200 lines per hour will present more defects [33].
	Proportion of changes without discussion	The proportion of changes related to a component that are not discussed.	Components that have several changes approved without critical discussion may display a poor design quality.
	Typical review window	The length of time between the creation of a review request and its final approval for integration, normalized by the size of the change (churn).	It could be argued that when the review window of a component is not long enough, developers are likely to miss critical design issues.
	Typical discussion length	The discussion length normalized by the size of the change.	Changes with many brief discussions do not leverage code review strengths and minimize its effectiveness.

data that the model is trained on and applying it to similar datasets. Our model construction approach is comprised of four steps that we describe below:

Estimate budget for degrees of freedom. As suggested by Harrell Jr. [37], before fitting our regression models, we estimate a budget for the models, i.e., the maximum number of degrees of freedom that we can spend. As suggested by Harrel Jr. [37], we spend no more than $\frac{n}{15}$ degrees of freedom on our OLS models, where n is the number of components in the modeled dataset.

Normality adjustment. Software engineering data is often skewed. Since OLS assumes that the response variable is normally distributed, we apply a log transformation $[ln(x+1)]$ to the response variables of our studied systems.

Correlation and redundancy analysis. Explanatory variables with high correlation can interfere with each other when interpreting the constructed regression model. Hence, before we construct our models, we remove highly correlated explanatory variables ($|\rho| \geq 0.7$). Moreover, redundant variables that are not highly correlated, but contain similar signals will also interfere with our interpretation. Hence, we perform redundant variable analysis by fitting models that explain each explanatory variable using the others with the `redun` function of the `rms` R package [38]. We remove the explanatory variables that have a model fit with an R^2 exceeding 0.9 – the default threshold of the `redun` function.

Allocate and spend model degrees of freedom. In order to spend our limited budget of degrees of freedom most effectively, we measure the Spearman multiple ρ^2 between response and explanatory variables. To mitigate the threat of overfitting our models, we limit the maximum number of degrees of freedom that we allocate to a single variable to 5.

We divide the explanatory variables into 3 groups. Explanatory variables that share the highest Spearman multiple ρ^2 scores with the explanatory variable are assigned 5 degrees of freedom. Explanatory variables with moderate Spearman multiple ρ^2 scores are assigned 3 degrees of freedom. Explanatory variables with low Spearman multiple ρ^2 scores are limited to a linear relationship with the explanatory variable (i.e., 1 degree of freedom). Note that as the ρ^2 values vary from project to project, it is difficult to define ranges that apply across all of our datasets, therefore, the decision to distribute is based on our knowledge of the domain and the estimated budget in order to avoid overfitting.

Finally, we fit our regression model, spending the degrees of freedom that we allocate to each explanatory variable using *Cubic Splines*. Since cubic splines tend to curl heavily upwards or downwards in the tails, they tend to perform poorly in these areas of the data. Hence, we use restricted cubic splines [37], which force the tails of the first and last degrees of freedom to be linear. We use the function `rcs` in the `rms` R package [38] to fit the allocated degrees of freedom for each explanatory variable.

D. Model Analysis

Once our regression models have been constructed, we analyze them to better understand: (1) the stability of the model, (2) the explanatory power provided by each surviving metric, and (3) the relationship shared between the incidence rates of anti-patterns and the explanatory variables that provide a statistically significant amount of explanatory power. We describe each model analysis step below.

Assessment of model stability. We evaluate the fit of our models using the *Adjusted* R^2, which provides a measure of fit that penalizes the use of additional degrees of freedom. However, since the *adjusted* R^2 is measured using the same data that was used to train the model, it is inherently upwardly biased, i.e., "optimistic". We estimate the optimism of our models using the following bootstrap-derived approach [39]:

1) From original dataset with n components, select a bootstrap sample also of n components with replacement.
2) Fit a model in the bootstrap sample using the allocation of degrees of freedom as the original model, and apply it to both the original and bootstrap samples.
3) The optimism is estimated as the difference in the adjusted R^2 of the bootstrap model in the bootstrap and original samples.

This calculation is repeated 1,000 times, and the model optimism is calculated as the mean of the bootstrap optimism estimates. This mean optimism is subtracted from the adjusted R^2 of the model fit on the original data to obtain the optimism-reduced adjusted R^2. The smaller the mean optimism, the higher the stability of the original model fit.

Estimate power of explanatory variables. The variables that were assigned additional degrees of freedoms are represented in the model by multiple terms. Hence, we jointly test the set of model terms that relate to one explanatory variable using Wald χ^2 maximum likelihood (a.k.a., "chunk") tests. The larger the Wald χ^2 value, the larger the impact that a particular explanatory variable has on the response. We report both the raw Wald χ^2 values, and its significance level according to its p-value.

Examine explanatory variables in relation to the outcome. While the chunk tests of the prior step provide a measure of the impact that an explanatory variable has on our models, it does not provide a notion of the direction(s) of the relationship it shares with the response. To uncover the direction of the relationship, we use plots that hold all other explanatory variables at their typical (median) values, while varying the explanatory variable under test. We achieve this, using the `Predict` function of the `rms` package [38], which also computes bootstrap-derived 95% confidence intervals for each plotted explanatory variable.

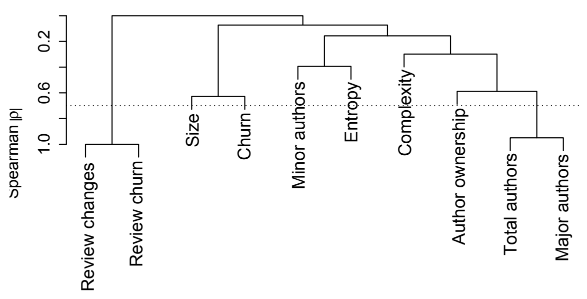

Fig. 2: Hierarchical clustering of variables according to Spearman's $|\rho|$ in QT.50. RQ1

Table III: Statistics of the regression model involving review coverage and anti-patterns counts

		Qt				VTK		ITK	
		5.0		5.1		5.10		4.3	
Adjusted R^2		0.72		0.36		0.63		0.30	
Optimism-reduced adjusted R^2		0.68		0.31		0.57		0.23	
Wald χ^2		2896.49***		690.04***		210.07***		79.65***	
Budgeted Degrees of Freedom		78		85		9		13	
Degrees of Freedom Spent		18		18		8		10	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	4	3	1	–	1	–
	χ^2	$982***$	$599***$	$341***$	$169***$	$30***$	–	$55***$	–
Complexity	D.F.	2	1	4	3	1	–	1	–
	χ^2	$< 1^\circ$	$< 1^\circ$	6°	6°	$7**$	–	$< 1^\circ$	–
Churn	D.F.	2	1	2	1	1	–	1	–
	χ^2	4°	4°	5°	$5*$	2°	–	$< 1^\circ$	–
Change entropy	D.F.	2	1	4	3	1	–	1	–
	χ^2	$< 1^\circ$	$< 1^\circ$	$14**$	$11**$	$8**$	–	1°	–
Total authors	D.F.	3	2	2	1	1	–	†	
	χ^2	$19***$	$9*$	$45***$	$5*$	$63***$	–		
Minor authors	D.F.	‡		‡		‡		1	–
	χ^2							1°	–
Major authors	D.F.	†		†		†		†	
	χ^2								
Author ownership	D.F.	4	3	3	2	1	–	1	–
	χ^2	6°	1°	1°	1°	$< 1^\circ$	–	$< 1^\circ$	–
Reviewed changes	D.F.	1	–	1	–	1	–	1	–
	χ^2	$< 1^\circ$	–	$7**$	–	1°	–	$17***$	–
Reviewed churn	D.F.	†		†		†		†	
	χ^2								

Discarded during:
† Variable clustering analysis ($|\rho| \geq 0.7$)
‡ Redundant variable analysis ($R^2 \geq 0.9$)
Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
$^\circ$ $p \geq 0.05$; $*$ $p < 0.05$; $**$ $p < 0.01$; $***$ $p < 0.001$
– Nonlinear degrees of freedom not allocated

IV. Study Results

In this section, we present the results of our case study with respect to our two research questions. For each research question, we first describe the metrics that we use, as well as the outcome of our model construction and model analysis steps described in Section III.

(RQ1) Is there a relationship between code review coverage and the incidence of anti-patterns?

Metrics. The response variable of our regression models of RQ1 is the *number of anti-patterns*, i.e., the total count of the anti-patterns detected within a component. Furthermore, as was done in prior work [19], we use the *proportion of reviewed changes* (i.e., the proportion of changes to a component that underwent code review) and the *proportion of reviewed churn* (i.e., the proportion of changed lines in a component that underwent code review) to measure review coverage in a component. For example, if 10 changes occurred during the development of a release, and only 7 of them could be connected with a code review, then the proportion of reviewed changes would be 0.7.

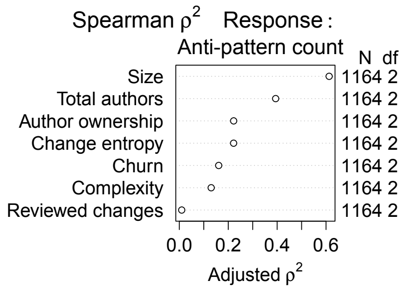

Fig. 3: Dotplot of the Spearman multiple ρ^2 of each explanatory and anti-patterns count in Qt 5.0. RQ1

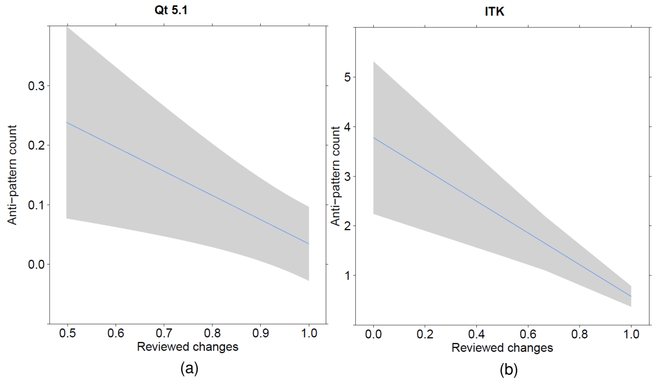

Fig. 4: Estimated count of anti-patterns in a typical component for various proportions of reviewed changes.

Model construction. We use hierarchical clustering to analyze the correlation between explanatory variables. We only retain one variable from clusters of explanatory variables with $|\rho| \geq 0.7$. For example, Figure 2 shows the cluster analysis of Qt 5.0. Similar figures were obtained for the other studied systems, but they are omitted from the paper to conserve space. We have made the other figures available online.[5]

Figure 2 shows that the correlation between the proportion of reviewed changes and the proportion of reviewed churn exceeds our $|\rho|$ threshold of 0.7. Similarly, the correlation between total authors and major authors also exceeds our $|\rho|$ threshold. We retain the proportion of reviewed changes and total authors because we believe that they are the most essential metrics in each pair. We dropped these metrics consistently for the rest of the projects, except for ITK where we find correlation between churn, total authors and major authors. In this case we keep churn as it is the simpler of the three metrics to choose.

Figure 3 shows the non-linear potential of the relationships between the explanatory variables and the count of anti-patterns in Qt 5.0 (the plots of the other studied systems are available online). Based on this figure, we allocate: (1) 5 degrees of freedom to component size, number of total authors, and author ownership ($\rho^2 \geq 0.20$); (2) 3 degrees of freedom to change entropy, churn, and complexity ($0.10 \leq \rho^2 \leq 0.19$); and (3) 1 degree of freedom to the proportion of reviewed changes ($\rho^2 \leq 0.01$). We followed a similar process to allocate degrees of freedom in Qt 5.1, VTK, and ITK datasets. However, we are more stringent with the allocation of degrees of freedom in the more budget-restricted VTK and ITK datasets.

[5]http://swat.polymtl.ca/data/SANER15/

Model analysis. Our regression models achieve optimism-reduced adjusted R^2 values between 0.23 (ITK) and 0.68 (Qt 5.0). The most stable model was obtained from Qt 5.0 with an adjusted R^2 of 0.72 and optimism-reduced adjusted R^2 of 0.68, while the worst stability values were achieved on ITK, i.e., an adjusted R^2 of 0.30 and an optimism-reduced adjusted R^2 of 0.23. The instability of ITK is likely due to the smaller number of components (194). Nonetheless, we scrutinize the results of the ITK model more carefully.

Although the Data Class, and God Class anti-patterns are size-dependent, component size is not the only consistently significant contributor of explanatory power. Unsurprisingly, Table III shows that size offers plenty of explanatory power to our software design models for all of the studied releases. Yet size is not the only significant contributor to the explanatory power of our models. The total number of authors is a significant contributor of explanatory power in three of the four studied releases. In fact, the total number of authors is a more powerful explanatory variable than component size in the studied VTK release.

The proportion of reviewed changes is a significant contributor of explanatory power in two of the four studied releases. Indeed, Table III shows that the proportion of reviewed changes is a significant contributor of explanatory power in Qt 5.1 and ITK models. In Figure 4 we observe the estimated anti-pattern count for Qt 5.1 and ITK of a component when varying the proportion of reviewed changes, while holding the other explanatory variables at their median values. In Figure 4 (b), we can observe a considerable fall of anti-pattern-proneness as the proportion of code review changes rise (4 to 1), while Figure 4 (a) shows a moderate fall. Even in the studied releases where the proportion of reviewed changes does not provide significant explanatory power, it does provide a unique signal that is not highly correlated with other explanatory variables, nor is the information redundant.

To gain a richer perspective about the relationship between review coverage and anti-patterns, we manually inspect the Qt 5.1 components with a high coverage rate and low number of anti-patterns. A good example of this is *OpenGL*, part of the `Qt GUI` module, which provides support for rendering graphics interfaces using OpenGL. This component with 81,648 LOC (the second largest component in the project) and review coverage rate of 96% does not contain any anti-patterns. Another example is the *LabelMap* library, from the *Filtering* subsystem in the ITK project. Despite the fact that it is the largest component outside of the third-party components, it contains no anti-patterns, while having a review coverage of 100%. This results suggest that the appearance of anti-patterns is not entirely reliant on the size of the components. Furthermore, review coverage appears to play a role in the incidence of anti-patterns.

Code review coverage have an impact on the incidence of anti-patterns in the studied software systems. This result suggests that code review practices can help to reduce the incidence of anti-patterns in software systems.

We repeat the analysis described above by building models for each type of anti-pattern (i.e., Code Duplication, Blob, Data Class, Data Clumps, Feature Envy, Schizophrenic Class, Tradition Breaker, MessageChain) separately. We use the same independent variables that were selected above. Specifically, for each project and each type of anti-pattern, we build a regression model using the total count of anti-patterns from that type found in the component as the response variable. For anti-patterns that are identified using the size (*e.g.*, Blob and Data Class), we excluded size from their regression models.

The models for the different types of anti-patterns achieved and adjusted R^2 between 0.10 (Qt 5.1) and 0.77 (VTK). Interestingly, the Code Duplication anti-pattern provides both the most and least stable models of the studied anti-patterns. The most stable model was achieved using VTK components (adjusted $R^2 = 0.81$, Optimism-reduced adjusted $R^2 = 0.76$), while the least stable model was also achieved using Qt 5.1 components (adjusted $R^2 = 0.12$, Optimism-reduced adjusted $R^2 = 0.04$). We attribute the instability of the Code Duplication anti-pattern model on Qt 5.1 to the smaller number of Code Duplication anti-pattern instances found in Qt 5.1 (21 for Qt 5.1 against 368 for VTK).

With respect to review coverage and the specific kinds of anti-patterns, the proportion of review changes is a statistically significant contributor of explanatory power for 3 types of anti-patterns in Qt 5.1 (i.e., The Blob, Data class and Data clumps), and 2 types of anti-patterns in VTK (i.e., Feature envy and Code Duplication). The *Graphics component* in VTK which has only 25% of review coverage contains the largest number of Feature envy (12), Sibling duplication (70) and Blob (42) instances. However, there are also components with a review coverage of 100% that have anti-patterns. The Data Class anti-pattern appears more frequently in components that achieve 100% review coverage than any other anti-pattern; we found 197 components in Qt 5.0 with a median of 2 Data classes per component; 98 in Qt 5.1 with a median of 2; 53 components in ITK with a median of 2, and finally 1 component in VTK with 18 Data Classes. We randomly investigated 133 of these Data Classes (63 Qt 5.0, 38 Qt 5.1, and 32 ITK), and suggest that 120 of them are related to project design (libraries defining common structures to be used in specific components). Take as an example, the filtering subsystem class from ITK that declares several *public structures* for the manipulation of images. Though the tool to detect anti-patterns consider a *struct* as a data class, as their members are public and it mainly holds data, the concept of class and *struct* differs, and their use might obey other reasons such as performance, simplicity, etc.

A high code review coverage can reduce the occurrence of Blob, Data class, Data clumps, Feature envy and Code Duplication in a software system. However, review coverage alone is not enough to ensure the absence of these anti-patterns.

(RQ2) Is there a relationship between code review participation and the incidence of anti-patterns?

Metrics. Similar to RQ1, the response variable of our RQ2 models is the total count of the anti-patterns detected within a component. Since RQ1 has shown that review coverage has an impact on the occurrence of anti-patterns, it is necessary to account for the proportion of reviewed changes when examining the impact of review participation on the occurrence of anti-patterns. Hence, when building our regression models to address RQ2, we select only the components that had a review coverage rate of 100%. We exclude the VTK project from our RQ2 analysis because it did not contain enough components with a review coverage rate of 100%. We measure code review participation using the following five metrics, described in our previous work [19]:

- *Proportion of self-approved changes*, i.e., the proportion of changes approved for integration by only the author of the change.
- *Proportion of hastily reviewed changes*, i.e., the proportion of changes that are reviewed at a rate faster than 200 lines per hour. According to Kemerer et al. [33], a code change should not be reviewed at a rate faster than 200 lines per hour.
- *Proportion of changes without discussion*, i.e., the proportion of changes that were approved for integration without any discussion inspired by human participants (i.e., ignoring comments generated by testing bots).
- *Typical review window*, i.e., the reviewing window of each change made on a component normalized by the amount of churn of the component. To assign a single value to each component, we compute the median value of this metric across all patches involving a component. We chose the median because it is capable of coping with outliers.
- *Typical discussion length*, i.e., the length of discussion for each change made to a component, normalized by the amount of churn in the component. To assign a single value to each component, we compute the median value of this metric across all patches involving a component.

Model construction. Table IV shows that our correlation analysis revealed problematic correlation between *author ownership*, *Total authors*, and *major authors*. We choose *Total authors* as in RQ1 because we believe it to be more essential than the two other metrics. We also observed a high correlation between the *typical review window* and *typical discussion length.* We selected the *typical discussion length* metric because it is more precise. In fact, the *typical review window* metrics may not measure the exact time that reviewers spent on the code review, since the review tool does not record the actual

Table IV: Statistics of the regression model involving review participation and anti-patterns counts

		Qt				ITK	
		5.0		5.1		4.3	
Adjusted R^2		0.74		0.46		0.20	
Optimism-reduced adjusted R^2		0.71		0.41		0.10	
Wald χ^2		3048.31***		919.21***		28.40**	
Budgeted Degrees of Freedom		73		75		8	
Degrees of Freedom Spent		17		20		11	
		Overall	Nonlinear	Overall	Nonlinear	Overall	Nonlinear
Size	D.F.	4	3	4	3	1	–
	χ^2	$937 * **$	$566*$	$233 * **$	$76 * **$	$12 * *$	–
Complexity	D.F.	1	–	2	1	1	–
	χ^2	$< 1^\circ$	–	2°	1°	$< 1^\circ$	–
Churn	D.F.	1	–	4	3	1	–
	χ^2	$< 1^\circ$	–	$42 * **$	$42 * **$	$< 1^\circ$	–
Change entropy	D.F.	2	1	2	1	1	–
	χ^2	$< 1^\circ$	$< 1^\circ$	$9 * *$	$8 * *$	1°	–
Total authors	D.F.	3	2	1	2	1	–
	χ^2	5°	2°	2°	$8*$	2°	–
Minor authors	D.F.	‡		1	–	1	–
	χ^2			$< 1^\circ$	–	$< 1^\circ$	–
Major authors	D.F.	†		†		†	
	χ^2						
Author ownership	D.F.	†		†		†	
	χ^2						
Self-approval	D.F.	2	1	1	–	1	–
	χ^2	2°	2°	$< 1^\circ$	–	$< 1^\circ$	–
Hastily-reviewed	D.F.	†		1	–	1	–
	χ^2			$59 * **$	–	$6*$	–
No discussion	D.F.	2	1	2	1	1	–
	χ^2	$7*$	2°	$139 * **$	$31 * **$	$< 1^\circ$	–
Typical review windows	D.F.	†		†		1	–
	χ^2					2°	–
Typical discussion length	D.F.	2	1	2	1	1	–
	χ^2	3°	$< 1^\circ$	$8*$	2°	3°	–

Discarded during:
† Variable clustering analysis ($|\rho| \geq 0.7$)
‡ Redundant variable analysis ($R^2 \geq 0.9$)
Statistical significance of explanatory power according to Wald χ^2 likelihood ratio test:
$^\circ$ $p \geq 0.05$; $*$ p <0.05;$**$ p <0.01;$* * *$ p <0.001
– Nonlinear degrees of freedom not allocated

time that reviewers spend reviewing. On the other hand, the *discussion length* captures how actively a change was discussed. We also observed a high correlation between *hastily-reviewed changes* and *changes without discussion* in Qt 5.0. Thus, we decided to keep the latter metric to remain consistent with our previous choice of *discussion length*.

Similar to RQ1, for each project, we investigated the non-linear potential of the relationships between the explanatory variables and the count of anti-patterns in order to decide on the allocation of degrees of freedom. Due to space constraints, we do not show detailed results here but we make them available online[6]. For Qt 5.0, based on the obtained results, we allocated (1) 5 degrees of freedom to component size, number of total authors, and proportion of self-approved changes; (2) 3 degrees of freedom to typical discussion length, change entropy and no discussion; and (3) 1 degree of freedom (linear fit) to complexity and code churn. The distribution of degrees of freedom for other studied projects is shown in Table IV.

Model analysis. Our regression models achieve optimism-reduced adjusted R^2 values of 0.20 (ITK), 0.46 (Qt 5.1), and 0.74 (Qt 5.0). The difference between initial adjusted R^2 and the Optimism-reduced adjusted R^2 for Qt projects is considerably low, ranging between 0.03 and 0.1, which indicates that overfitting is not a major concern for our models.

[6]http://swat.polymtl.ca/data/SANER15/

Concerning participation metrics, no discussion rate provides a significant amount of explanatory power for both studied Qt versions. Moreover, in Qt 5.1, this metric has high level of significance in the number of degrees of freedom assigned for both linear and non-linear columns, whereas ITK only reports hastily-reviewed metric as significant. Given the limited budget of degrees of freedom, we opt for not adding any additional knots to any variable; i.e., fitting a linear relationship.

Figure 5 shows the estimated anti-patterns count for Qt (5.0 and 5.1) and ITK projects varying the participation metrics while holding the other variables at their median values. In Figure 5 (a) the proportion of changes without discussion has a positive impact in the appearance of anti-patterns with a little drop at the beginning and then increments progressively until it reaches the value of 0.2. Figure 5 (b) shows a similar trend but with broader confidence intervals in comparison with Qt 5.0. This is because the model for Qt 5.1 is supported by more data points. We also observe that the count of anti-patterns in Qt 5.1 rises considerably higher in comparison to the other projects. Figure 5 (c) shows the estimated count of anti-patterns in ITK when varying the proportion of hastily-reviewed changes. The blue line that represents the original data is straight; indicating that the count of anti-patterns rises when components are reviewed very fast. However, this value should be interpreted with caution, since Table IV shows that the ITK model is not as stable as Qt models.

The lack of participation during code reviews has a negative impact on the occurrence of anti-patterns in components. Software organisations should encourage a large participation of their reviewers to code reviews, in order to improve design quality.

Similar to RQ1, we repeat the analysis described above using each type of anti-pattern. We use the same metrics as explanatory variables and maintain the same distribution of degrees of freedom as in the models with the total anti-pattern count. Results (summarized in Table V) show that for 5 out of 7 anti-patterns, no discussion rate provides a significant amount of explanatory power for both studied Qt versions. For the pair Qt 5.1-ITK, hastily-reviewed rate provides a significant amount of explanatory power to the models of Code Duplication and God Class respectively. The total number of authors also provides a significant amount of explanatory power for Data Class in Qt 5.0 and Qt 5.1.

The lack of participation during code reviews has a negative impact on the occurrence of Tradition Breaker, Code duplication, Data class, Feature Envy, God Class and Schizophrenic class.

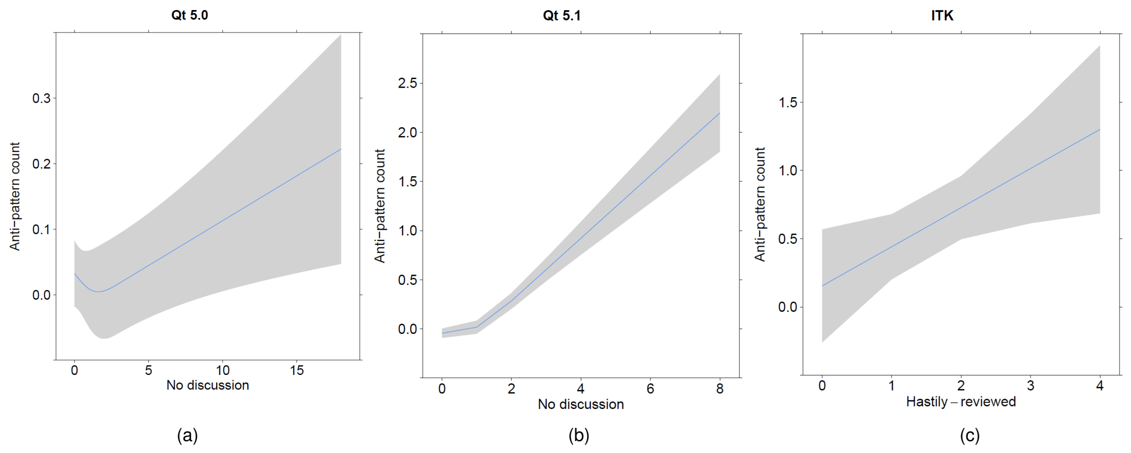

Fig. 5: Estimated count of anti-patterns in a typical component for various proportions of participation metrics.

Table V: Relevance of review participation metrics for individual anti-patterns

	No Discussion	Hastily Reviewed
Qt 5.0-5.1	Tradition Breaker	
	Data Class	
	Feature Envy	
	God Class	
	Schizophrenic Class	
ITK		Code Duplication
		God Class

V. Threats to Validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [40].

Construct validity threats concern the relation between theory and observation. Our modeling approach assumes that each anti-pattern is of equal importance, when in reality, this may not be the case.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. The accuracy of InCode impacts our results. InCode is a commercial tool which has been reported to achieve high precision and recall [41]. However, other anti-pattern detection techniques and tools should be used to confirm our findings.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models.

Reliability validity threats concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from data sets [42]. To mitigate these threats we tested our hypotheses over two versions of Qt and another two open source projects (*i.e.*, VTK and ITK). In addition to this, we attempt to provide all the necessary details required to replicate our study. The source code repositories and issue-tracking systems of Qt, ITK and VTK are publicly available to obtain the same data.

Threats to external validity concern the possibility to generalize our results. Our study is focus on three open source software systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable, considering systems from different domains, as well as several systems from the same domain. In this study, we used a particular yet representative subset of anti-patterns as proxy for software design quality . Future work using different anti-patterns are desirable.

VI. Conclusion and Future Work

In this study, we examine the impact that Modern Code Review practices, a lightweight, tool-supported variant of the code review process, can have on software design quality. We present a quantitative study of three large open source projects (i.e., Qt, VTK and ITK) that relates *code review coverage* and *code review participation* to design quality, by leveraging anti-patterns as indicators of poor design quality. Results show that:

- Code review coverage has an impact on the incidence of anti-patterns in two of the four studied systems. A high code review coverage can reduce the occurrence of Blob, Data class, Data clumps, Feature envy and Code Duplication in a software system. However, review coverage alone is not enough to ensure the absence of these anti-patterns.
- The lack of participation during code reviews has a negative impact on the occurrence of Tradition Breaker, Code duplication, Data class, Feature Envy, God Class and Schizophrenic class. Components containing a high proportion of changes that are hastily-reviewed may exhibit God Classes and Code duplication.

The results of this study provide empirical evidences that good code review practices could help improve the design quality of software systems. Software organisations should encourage a large participation of their developers to code reviews, in order to improve design quality.

References

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] W. S. Humphrey, T. R. Snyder, and R. R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, vol. 8, no. 4, pp. 11–23.

[3] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proc. of the 8th IEEE Symposium on Software Metrics*, 2002, pp. 249–258.

[4] S. McConnell, *Code Complete*. Microsoft, 2004.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[8] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. of the 29th Int'l Conference on Software Maintenance*, 2013, pp. 270–279.

[9] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th Int'l Conf. on*. IEEE, 2010, pp. 23–31.

[10] D. L. Parnas, "Software aging," in *ICSE '94: Proc. of the 16th Int'l conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 279–287.

[11] M. M. Lehman, "Laws of software evolution revisited," in *In European Workshop on Software Process Technology*, 1996, pp. 108–124.

[12] C. Izurieta, "Decay and grime buildup in evolving object oriented design patterns," Ph.D. dissertation, 2009.

[13] S. Vaucher, F. Khomh, N. Moha, and Y. Gueheneuc, "Tracking design smells: Lessons from a study of god classes," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, Oct 2009, pp. 145–154.

[14] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conf. on*. IEEE, 2009, pp. 145–154.

[15] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. on the*. IEEE, 2010, pp. 106–115.

[16] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 Int'l Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 712–721.

[17] M. V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?" *Transactions on Software Engineering (TSE)*, vol. 35, no. 3, pp. 430–448, 2009.

[18] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 202–211.

[19] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 192–201.

[20] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[21] F. Khomh, M. Di Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, Oct 2009, pp. 75–84.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

[23] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development*, 1^{st} ed. Prentice-Hall, Upper Saddle River, NJ (2005), 2005.

[24] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.

[25] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, Apr. 2011.

[26] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *3rd Int'l Symposium on Empirical Software Engineering and Measurement, ESEM 2009,*, 2009, pp. 390–400.

[27] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conf. on*. IEEE, 2012, pp. 411–416.

[28] R. Shatnawi and W. Li, "An investigation of bad smells in object-oriented design," in *Information Technology: New Generations, 2006. ITNG 2006. 3rd Int'l Conf. on*. IEEE, 2006, pp. 161–165.

[29] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on*, March 2011, pp. 181–190.

[30] D. I. K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[31] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of the 30th Int'l Conf. on Software engineering*, 2008, pp. 541–550.

[32] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: An effective verification process," *IEEE Softw.*, vol. 6, no. 3, pp. 31–36, May 1989.

[33] C. F. Kemerer and M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 534–550, 2009.

[34] M. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" *Software Engineering, IEEE Transactions on*, vol. 35, no. 3, pp. 430–448, May 2009.

[35] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How changes affect software entropy: an empirical study," *Empirical Software Engineering*, vol. 19, no. 1, pp. 1–38, 2014.

[36] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 4–14.

[37] F. E. Harrell, *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer, 2001.

[38] F. E. Harrell Jr, "rms: Regression modeling strategies. r package version 3.4-0," 2012.

[39] B. Efron, "How biased is the apparent error rate of a prediction rule?" *Journal of the American Statistical Association*, vol. 81, no. 394, pp. 461–470, 1986.

[40] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[41] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[42] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.