Impacts of Design Pattern Decay on System Quality

Melissa R Dale
Computer Science Department
Montana State University
Bozeman, MT
mdale@cs.montana.edu

Clemente Izurieta
Computer Science Department
Montana State University
Bozeman, MT
clemente.izurieta@cs.montana.edu

ABSTRACT

Context Software systems need to be of high enough quality to enable growth and stability.

Goal The purpose of this research is to study the effects of code changes that violate a design pattern's intended role on the quality of a project.

Method To investigate this problem, we have developed a grime injector to model grime growth, a form of design pattern decay, on Java projects. We use SonarQube's technical debt software to compare the technical debt scores of six different types of modular grime. These six types can be classified along three major dimensions: strength, scope, and direction.

Results We find that the strength dimension is the most important contributor to the quality of a design and that temporary grime results in higher technical debt scores than persistent grime.

Conclusion This knowledge helps with design decisions that help manage a project's technical debt.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design Concepts—*OO Design Methods*; D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*

General Terms

Measurement

Keywords

Technical Debt, Design Pattern Grime, Object Orientated Design Patterns, System Quality

ESEM'14, September 18-19, 2014, Torino, Italy.

http://dx.doi.org/10.1145/2652524.2652560

1. INTRODUCTION

Design patterns are used in software engineering to reinforce consistent solutions to common problems. However, as a system ages, changes are introduced as a result of bug fixing or new features being added. As systems evolve, the coupling between pattern and non-pattern classes tends to increase and the intended design patterns can become obscured by code that violates the pattern's intended purpose. The unintended growth in coupling disharmonies was defined by Izurieta and Bieman [4] as modular grime, and is the most common form of design pattern decay.

We investigate the effects that modular grime has on the overall quality of a system when quantified as technical debt. Technical debt is a metaphor borrowed from the financial domain and introduced by Ward Cunningham [2]. It describes the amount of work needed to repay the debt incurred by taking shortcuts, such as choosing decisively negative coding practices in order to meet a deadline. We hypothesize that not all types of modular grime have the same impact on the technical debt of a project. To investigate, we use SonarQube [1] to measure technical debt and construct a grime injector to model instances of modular grime.

In the following sections, we discuss technical debt (Background Section 2.1) and design pattern grime (Background Section 2.2), a statement of the problem we are trying to solve (Problem Statement Section 3), a description of the experiment (Experiment Section 4) and the results(Results Section 5). We cover threats to the validity of this study (Threats to Validity Section 6) We present our conclusions (Conclusion Section 7), and finally we discuss future research (Future Research Section 8).

2. BACKGROUND

2.1 Technical Debt

The term "technical debt", coined by Ward Cunningham in 1992 [2], describes the cost (which can be measured in terms of dollars or man-hours) that a design decision will cost in the future at the expense of a short term gain. Like financial debt, technical debt is necessary for a product to advance.

One proposed method to calculate technical debt is the Software Quality Assessment Based on Lifecycle Expectations (SQALE) method. Letouzey and Coq [8] presented the SQALE methodology in 2010, and in 2012 Letouzey [9] uesed this method to evaluate technical debt. SQALE utilizes four key concepts to build a technical debt framework:

quality model, analysis model, indices, and indicators. The quality model consists of a a list of rules code should follow to have high quality. The analysis model defines the cost to fix violations, also known as the remediation cost, of the quality model. The quality model is mapped to the analysis model using indices. The SQALE Indicators highlight potential areas of concern in a system. They are used for analysis and visual representations, such as dashboards.

An example of a quality model and analysis model are given in Figure 1.

Characteristic	Sub-Characteristic	Requirement
Maintainability	Readability	There is no commented out block of code

Requirement	Remediation Details	Remediation Function
There is no Commented out block of code	Remove (Because there is no impact on compiled code)	2 minutes per occurrence

Figure 1: SQALE Quality and Analysis Model Examples

2.2 Design Pattern Grime

Izurieta and Bieman [6] breakdown design pattern decay into two components: pattern rot and pattern grime. Rot is the breakdown of the structural integrity of a design pattern realization. The term "grime" refers to the accumulation of code that violates the intended role of the design pattern, but does not break the structural integrity of that design pattern. Rot and Grime are mutually exclusive.

Three types of grime were defined by Izurieta and Bieman [4]: organizational, modular, and class. This study focuses on modular grime. Schanz and Izurieta [10] defined taxonomy for modular grime along three dimensions as shown in Figure 2: the scope of the coupling, the direction of the coupling, and the strength of the coupling.

2.2.1 *Strength: Temporary or Persistent*

Strength refers to the difficulty of removing the coupling [3]. Strength may be either temporary or persistent. In temporary couplings, a class A uses a method with a parameter, a return value, or a local variable of another class type, i.e. class B. Persistent couplings occur when a class A contains an attribute of type class B.

2.2.2 *Scope: Internal or External*

The scope of the coupling refers to where the coupling occurs. If both classes that are coupled reside in the design pattern, the scope is internal. If the coupling connects a non-pattern class to a pattern class, the scope is external.

2.2.3 *Direction: Efferent or Afferent*

If the grime connects a pattern class to a non-pattern class, the direction of that coupling is classified according to its origination source. An instance of grime that originates inside a pattern and forms a relationship with a non-pattern class, is referred to as efferent. If the grime originates outside of a pattern and forms a relationship with a pattern class, then the grime is referred to as afferent.

Using these dimensions, Schanz and Izurieta defined six types of modular grime: Persistent External Afferent Grime (PEAG), Persistent External Efferent Grime (PEEG), Persistent Internal Grime (PIG), Temporary External Afferent Grime (TEAG), Temporary External Efferent Grime (TEEG), and Temporary Internal Grime (TIG). The diagram in Figure 2 depicts the structure of the taxonomy.

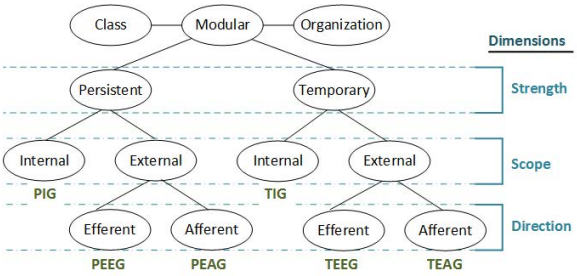

Figure 2: Grime Taxonomy defined by Schanz and Izurieta [10]

3. PROBLEM STATEMENT

The relationships between technical debt and grime are important to understand when considering the role grime plays in the technical debt of a system. Izurieta et al. [7] identify design pattern grime as a component of the technical debt landscape.Further work by Zazworka et al. [11] studied the impact of grime as one component of design disharmony.

Some initial work has been done to understand the negative impacts of grime. Izurieta and Bieman [5] found that as grime grows, so do testing requirements, which can negatively impact system testability. Research to quantify grime in terms of technical debt does not exist. This research takes the first steps in quantifying the effects of modular design pattern grime on technical debt.

4. EXPERIMENT

To study differences in the perceived effects of different types of modular grime on technical debt, we model the growth of modular grime on Java projects that implement design patterns. We have created a Grime Injector which allows us to modify Java bytecode with couplings that characterize all the modular grime types we are modeling. Given a list of the pattern classes and a list of the non-pattern classes, the injector randomly selects a pattern class and a non-pattern class for external grime types, or two pattern classes for internal grime types. A coupling is then created between these two classes. If the grime type is external and afferent, the coupling originates from the non-pattern class to the pattern class. If the grime type is external and efferent, the coupling originates from the pattern class to the non-pattern class. If the grime type is internal, the direction of the coupling is irrelevant. [1]

[1]A complete description of the Grime Injector and the methodologies used to model Modular Grime may be found at: www.cs.montana.edu/∼mdale/grime-injector

4.1 Methodology

Using three programs from an introduction to software engineering class (which implements design patterns) and the injector described above, three sets of modified projects are produced. Each set consists of six manipulated sub-projects, one for each of the six types of modular grime. Each set can then be analyzed with SonarQube [1] to obtain a technical debt score to examine differences in types of modular grime.

4.2 Hypotheses

We investigate the following research question: is there a difference in the technical debt scores reported by SonarQube for the different types of modular grime? Our hypotheses are:

H_o: $\tau_{peag} = \tau_{peeg} = \tau_{pig} = \tau_{teag} = \tau_{teeg} = \tau_{tig}$ That is, there is no difference in the treatment effects of the six different types of modular grime on technical debt as reported by SonarQube.

4.3 Experimental Units

The experimental units are Java programs used to teach design patterns in an introduction to software engineering course. We use three kinds of design patterns, one for each of the categories of design patterns: behavioral, structural, and creational [3]

4.4 Experimental Design

We use a Randomized Complete Block Design (RCBD) because we would like to control the variability that comes from the different design patterns. The six modular grime types are the treatments for this design, the design pattern categories are the blocks, and the technical debt scores reported by SonarQube are the response variables. For each block and treatment, five scores are generated.

5. RESULTS

We use Analysis of Variance tests to analyze treatment effects between treatments. We repeat the experiment three times, one with 10 instances of each type of modular grime, one with 50 instances of each type of modular grime, and one with 100 instances of each type of modular grime. We find for all cases, 10, 50, and 100 instances of modular grime there is sufficient evidence to reject the null hypothesis that all types of grime have the same effect on technical debt. All cases have a p-value of <0.001, less than an alpha value of 0.05. In other words, there is less than a .01 percent chance we observed these results purely by chance. Figure 3 gives the ANOVA results.

Because the null hypothesis that all treatment effects are equal was rejected, we performed a Tukey's test to test all pairwise mean comparisons to see which treatment effects are statistically different from each other. We found that all three types of persistent grime (PEAG, PEEG, PIG) showed significantly lower technical debt scores than all three types of temporary grime (TEAG, TEEG, TIG). The SAS results for the Tukey test (grouped to show statistical differences) are given in Figure 4. Figure 5 displays the technical debt score calculated by SonarQube for 10, 50, and 100 instances of modeled grime for each of the modular grime types.

10 Instances

Source	DF	Sum of Squares	Mean Square	F Value	Pr>F
Model	7	10.5478	1.5068	583.44	<0.0001
Error	82	0.2118	0.0026		
Corrected Total	89	10.7596			

50 Instances

Source	DF	Sum of Squares	Mean Square	F Value	Pr>F
Model	7	46.143	6.5919	329.46	<0.0001
Error	82	1.6407	0.0200		
Corrected Total	89	47.784			

100 Instances

Source	DF	Sum of Squares	Mean Square	F Value	Pr>F
Model	7	165.7464	23.6781	503.03	<0.0001
Error	82	3.8598	0.04711		
Corrected Total	89	169.6062			

Figure 3: ANOVA Results

Means with the same letter are not significantly different.

10 Instances			50 Instances			100 Instances		
TG	MEAN	TYPE	TG	MEAN	TYPE	TG	MEAN	TYPE
A	3.7667	TIG	A	4.8733	TEEG	A	6.5533	TIG
A	3.7600	TEEG	A	4.8200	TIG	A	6.5267	TEAG
A	3.7267	TEAG	A	4.8067	TEAG	A	6.5000	TEEG
B	3.5533	PEAG	B	3.7600	PEAG	B	4.2533	PIG
B	3.5333	PEEG	B	3.7133	PIG	B	4.2133	PEAG
B	3.5133	PIG	B	3.7067	PEEG	B	4.1400	PEEG

TG: Tukey Grouping

Figure 4: Tukey Grouping Results for Tukey's Pairwise Comparison Test

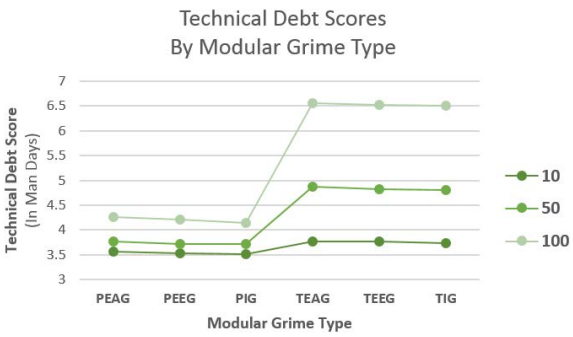

Figure 5: Graph of Technical Debt Score Results for 10, 50, and 100 Modeled Modular Grime Instances

6. THREATS TO VALIDITY

6.1 Construct Validity

Currently there is no benchmark for measuring technical debt. Because of this, the response variable (technical debt) being reported by SonarQube is a threat to the construct validity of this research. SonarQube's ability to accurately measure technical debt may not accurately reflect the technical debt of a system.

There exists a potential to inject false-positive grime. Because the injector works by selecting two random classes, there is no assurance that the coupling will violate the design pattern's intended purpose and they may not in actuality be considered grime. Future work will include efforts to better model and validate the grime injector's accuracy.

6.2 Internal Validity

In order to manipulate the Java projects to model grime growth, the bytecode is modified by the Grime Injector and then decompiled back to Java source code. Elements of the original code such as comments, are lost and added in this process. This risk is minimized by the fact that outputs will be equally skewed between each modeled grime type. If attempting to perform analysis between original code and modeled code, this factor needs to be taken into account.

6.3 External Validity

The research conducted used solely Java projects, therefore any findings can only be generalized to Java projects. Further research will be needed to be able to generalize findings to a larger code population.

We have only measured technical debt using SonarQube. This is also a threat to the external validity as we cannot speak to how other means of calculating technical debt based on practical quality models might compare.

Another threat to the external validity is that we have only used one representative pattern for the categories of creational, behavioral, and structural design patterns.

7. CONCLUSIONS

Understanding the role modular grime plays in the technical debt field will help lead to better understanding of the cost associated with grime and technical debt management. Knowing the effects of different types of grime will allow software engineers to make design decisions that result in lower technical debt and a more comprehensive technical debt framework.

In this research, we used SonarQube to calculate a technical debt score for Java projects modified by our grime injector to represent modular grime growth. We then performed an ANOVA analysis on the results to find that not all types of modular grime result in equivalent technical debt scores. Tukey's test shows that every type of temporary grime (TEAG, TEEG, TIG) is statistically significant and has higher technical debt score (as reported by SonarQube) than every type of persistent grime (PEAG, PEEG, PIG).

Our research provides reasons to care not only about grime growth, but also the type of grime. Knowing temporary grime types can be more costly than persistent grime types, engineers can make better informed design decisions or repayment plans that will result in lower technical debt.

Quantifying grime in terms of technical debt is the first step to including grime in a technical debt management plan. The findings of this research form a foundation to continue exploring the relationship between design pattern grime and technical debt. Further research will explore ways to provide a more holistic view of grime and technical debt.

8. FUTURE RESEARCH

While the research presented here are the first steps towards understanding the role design pattern grime plays on technical debt, there is still more investigation to be conducted. The following paragraphs describe possible areas of future research.

Here we investigated differences between the different types of modular grime on technical debt scores reported by SonarQube. Some of our findings suggest that temporary grime types are not only more costly in technical debt scores reported by SonarQube, but also accrues debt at a quicker rate. Further investigation can be conducted to understand the rates at which technical debt grows as grime instances increase.

Lastly, SonarQube is only one tool that evaluates technical debt. Investigating modified programs with other tools that evalute software quality and technical debt will expand our understanding of the true role grime growth plays in technical debt.

9. REFERENCES

[1] Sonarqube 3.7.4. http://www.sonarqube.org/downloads/.

[2] Cunningham, W. The wycash portfolio management system. In *ACM SIGPLAN OOPS Messenger* (1992), vol. 4, ACM, pp. 29–30.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[4] Izurieta, C., and Bieman, J. M. How software designs decay: A pilot study of pattern evolution. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* (2007), IEEE, pp. 449–451.

[5] Izurieta, C., and Bieman, J. M. Testing consequences of grime buildup in object oriented design patterns. In *Software Testing, Verification, and Validation, 2008 1st International Conference on* (2008), IEEE, pp. 171–179.

[6] Izurieta, C., and Bieman, J. M. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal 21*, 2 (2013), 289–323.

[7] Izurieta, C., Vetro, A., Zazworka, N., Cai, Y., Seaman, C., and Shull, F. Organizing the technical debt landscape. In *Proceedings of the Third International Workshop on Managing Technical Debt* (2012), IEEE, MTD 2012, pp. 23–26.

[8] Letouzey, J.-L. The sqale method for evaluating technical debt. *MTD, ICSE* (2012).

[9] Letouzey, J.-L., and Coq, T. The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In *Proceedings of the 2010 Second International Conference on Advances in System Testing and Validation Lifecycle* (2010), IEEE Computer Society, pp. 43–48.

[10] Schanz, T., and Izurieta, C. Object oriented design pattern decay: a taxonomy. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (2010), ACM.

[11] Zazworka, N., Cai, Y., Izurieta, C., Seaman, C., and Shull, F. Comparing four approaches for technical debt identification. *Software Quality Journal, Springer US* (April 2013), 1–24.