Investigating the Impact of Design Debt on Software Quality

Nico Zazworka[1], Michele A. Shaw[1], Forrest Shull[1], Carolyn Seaman[1,2]

[1]Fraunhofer Center for Experimental Software Engineering
5825 University Research Court, Suite 1300
College Park, Maryland 20740-3823
240-287-2925
{nzazworka,mshaw,fshull}@fc-md.umd.edu

[2]UMBC
1000 Hilltop Circle
Baltimore, MD, USA
cseaman@umbc.edu

ABSTRACT

Technical debt is a metaphor describing situations where developers accept sacrifices in one dimension of development (e.g. software quality) in order to optimize another dimension (e.g. implementing necessary features before a deadline). Approaches, such as code smell detection, have been developed to identify particular kinds of debt, e.g. design debt. What has not yet been understood is the impact design debt has on the quality of a software product. Answering this question is important for understanding how growing debt affects a software product and how it slows down development, e.g. though introducing rework such as fixing bugs. In this case study we investigate how design debt, in the form of god classes, affects the maintainability and correctness of software products by studying two sample applications of a small-size software development company. The results show that god classes are changed more often and contain more defects than non-god classes. This result complements findings of earlier research and suggests that technical debt has a negative impact on software quality, and should therefore be identified and managed closely in the development process.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics

General Terms

Measurement, Design, Experimentation, Verification.

Keywords

Technical Debt, Design Debt, Code Smells, God Class, Maintainability, Refactoring

1. INTRODUCTION

Technical Debt is a metaphor describing the tradeoffs of technical development activities that are delayed in order to get short-term payoffs, such as a timely software release [10]. The term debt is used because the consequences associated with these skipped activities accumulate in the software product as debt. Even though it might pay off in the short run with quicker delivery, similar to going into financial debt to buy a new car, additional costs must be paid back later. If too much technical debt accumulates in a software project, development will slow down, e.g. caused by increasingly poor maintainability of the code. Technical debt can take many forms, some of them manifesting themselves in code bases, others in documentation, requirements, or human interaction (e.g. communication debt caused by a lack of communication between developers).

1.1 Identifying and Quantifying Technical Debt

The goals and challenges of research in technical debt are to identify and quantify debt in existing projects and to provide mechanisms to manage debt in those projects. Quantifying technical debt can provide insight as to when debt should be paid, or when it is acceptable to be in debt and defer debt for resolution later. One specific form of technical debt that has been studied for some time is design debt, also referred to as architecture debt. Design debt occurs whenever the software design no longer fits its intended purpose. A software project can run into design debt for various reasons. For example, adding a series of features that the initial architecture was not intended to support can cause debt and decrease the maintainability of software. Or, drifting away from a proposed architecture can bring short-term payoffs (e.g. quick and dirty implementation of a new feature) but might have consequences for the portability and interoperability of software in future. Reducing or eliminating design debt means in most cases that the design should be tailored and adapted towards changing requirements immediately and continuously.

1.2 Paying off Design Debt

One possible activity of adapting the architecture or design to facilitate future requirements is called refactoring. Refactoring means adjusting the design without changing the external behavior of a program. From our experience with the studied software projects in this paper, refactoring is usually performed when developers feel that the current architecture cannot support the software's purpose anymore. In that case, the cost to pay off the design debt is the time spent refactoring (e.g. planning the new design, rewriting code, adjusting documentation). Refactoring activities can be on different scales. Small-scale refactoring includes activities like renaming or moving small code fragments between classes (e.g. one function). Medium-scale refactoring may include splitting up classes (e.g. classes that have grown too large). Large-scale refactoring will affect multiple classes, e.g. changes in inheritance hierarchy or the translation of software parts into design patterns.

In many software projects the cost and benefit of a refactoring is usually not quantified and therefore is difficult for developers to justify and communicate to management that refactoring should be done and more importantly how it will pay off in terms of improved product quality. More specifically, it is challenging to predict the effort required to refactor a system, and the impact in terms of maintainability and making future product changes easier. The latter is in most cases almost impossible to forecast since the usefulness of an architecture will depend on the type of future changes, which most of the time is unknown. However, one possible way to understand the impact of refactoring is to understand what impact the currently unpaid design debt has on the development, e.g. how much does poor design slow down development, decrease maintainability, or affect the correctness of the software? The decision regarding whether a refactoring is necessary can then be evaluated based upon these observations.

1.3 Which Design Debt Symptoms are Worth Investigating?

Growing design debt can take many forms that express themselves through different symptoms. Growing debt affects software development negatively by making the code less maintainable, less correct, more difficult to extend, less portable, and more complicated to understand. Possible symptoms of design debt are, for example, simple changes that take a long time or require writing an unusually large amount of code. Or, it may express itself in code structures that drift away from good object-oriented design principles, e.g. become too entangled, too complex and too hard to modularize.

One class of possible symptoms of design debt is code smells. Code smells are indicators of violations against the principles of good object-oriented design. For example, a common object oriented design principle states that a class should have a single purpose and not implement multiple functions of the system. If a class tries to accomplish too many purposes it is said to exhibit the "god class" code smell and should be refactored. In the case of this code smell the recommended refactoring strategy would be to split up the class into multiple classes. Section 1.5 will give more details on god classes.

Since there are many symptoms, such as code smells [6], that can be partially identified automatically [4], it is important to test whether a code smell is a strong indicator for pending debt in the software. To test this, one must build evidence that the presence of the code smell correlates or, even better, causes the effects of the growing debt. More specifically, one wants to show, for example, that components infected with a specific code smell are indeed less maintainable or less correct. Or, one would be interested in showing that in development phases that showed fewer code smells productivity was higher than in the ones with more code smells. If a relationship cannot be established it is questionable to what extent the code smell represents debt, and if it should be addressed at all.

God classes have been claimed [4] to be maintenance bottlenecks and to have a high potential as refactoring opportunities. Therefore, we will investigate in this paper whether a correlation between the design debt symptom code smell (god class) and software quality characteristics, such as maintainability and correctness, can be established.

1.4 Research Questions

The first part of this paper will be concerned with investigating the relationship between a particular kind of design debt, i.e. god classes, and two quality characteristics of software: maintainability and correctness. Our hypotheses are that classes infected with the god class code smell are harder to maintain, and therefore have to be changed more often; and that god classes contain more defects than classes without the smell.

More precisely we will test the following two hypotheses:

H1: The change likelihood of god classes is higher than the change likelihood of non-god classes.

H2: The defect likelihood of god classes is higher than the defect likelihood of non-god classes.

In past work we and other authors have argued that the above analysis might be confounded by the circumstance that god classes tend to be the larger classes of the system. The assumption was that larger classes should be more change and defect prone since they implement more functionality.

The second part of this paper will investigate the validity of these assumptions. Specifically we will question whether normalization by lines of code (LOC-normalization) is correct by analyzing the correlation between changes and class size, and defects and class size. The goal will be to provide better guidance for future research to whether or not data should be normalized, and how.

Following research questions are investigated:

R1: Is there a linear relationship between class size and change likelihood that justifies LOC-normalization?

R2: Is there a linear relationship between class size and defect likelihood that justifies LOC-normalization?

Last, we will exemplify how LOC-normalizing the results in H1 and H2, as done in earlier work, will influence the results.

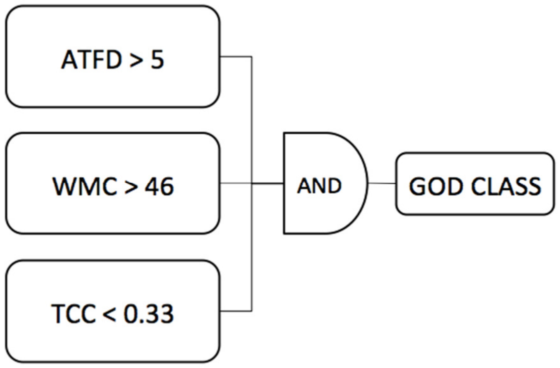

Figure 1: Detection strategy for god classes as presented in Lanza and Marinescu [4]

1.5 Code Smells and God Classes

The evaluation and proposed method will focus on god classes as technical design debt indicators. The god class code smell describes classes that over-centralize functionality in a system and are more concerned with the data of other classes than their own data. God classes are, to our knowledge, the most well studied code smell in scientific literature. This may be due to the easy intuition behind their definition: classes that fulfill multiple purposes in a software system and grow too complex will be harder to understand and maintain over time. The life cycle of a god class corresponds with the characteristics of the technical debt metaphor: developers decide on overloading a class with functionality when they face time pressure and new features have to be implemented quickly. Over time a god class will grow more complex, become less coherent, and increase its coupling to other classes. God classes can be identified using Marinescu's [5] detection strategy that combines the use of three software metrics as shown in Figure 1. The weighted method count (WMC) metric

Table 1: Related work and statistically significant findings

Related Work	Investigated Software	Analysis Resolution	Investigated code smells	God classes more change prone if not normalized? (p<0.05)	God classes more change prone if LOC normalized? (p<0.05)	God classes more defect prone if not normalized? (p<0.05)	God classes more defect prone if LOC normalized? (p<0.05)
Li 2007 [7]	Eclipse	3 releases	6 (including god class)			Yes	
Olbrich 2009 [3]	Lucene, Xerces	Every 50th revision	god class, shotgun surgery	Yes			
Schumacher 2010 [1]	Two commercial applications	Every revision	god class	Yes	No		
Olbrich 2010 [2]	Lucene, Xerces, Log4j	Every 50th revision	god class, brain class	Yes	No, even less change prone	Partly, in 2 out of 3 cases	No, even less defect prone in 2 out of 3 cases
Khomh 2009 [8]	Azereus, Eclipse	Every main release	29 code smells, (including large class code smell)	Partly, in 5 out of 10 releases for large class smell			
Study results presented here	Two commercial applications	Every revision	god class	Yes	Partly, in 1 out of 2 cases	Partly, in 1 out of 2 cases	Partly, in 1 out of 2 cases

computes the sum of methods for a class weighted by their cyclomatic complexity (in other words WMC is the sum of McCabe's cyclomatic complexity of all methods). The tight class cohesion (TCC) metric measures the internal cohesion of the class. The access to foreign data (AFTD) metric measures the number of accesses to data in other classes (either directly or through getter methods) that are performed. All three metrics are compared to thresholds and if each metric threshold is out of bounds, the class is marked as a god class. Often, god classes are among the more complex and larger classes of software, however, the reverse does not always hold true: all large classes are not always god classes.

2. RELATED WORK

The concept of code smells was first introduced by Fowler and Beck (as Bad Smells) [6] and describes patterns in object oriented code that are less than ideal, e.g. that violate the rules of good object-oriented design, and should be refactored. Code smells are a type of technical debt, i.e. design debt, because they are believed to slow down development when not removed early. The original proposal of code smells by Fowler and Beck suggested developers identify code smells by performing code reviews or during development, i.e. continuous refactoring. Lists of code smells exist also online in the agile community [7].

After the introduction of code smells Marinescu [5] was the first to see the opportunity to detect code smells by using metric based rules that can be executed automatically by a tool. Our previous work [1] indicated that recall and precision can be sufficiently high to use the metric-based approach as a filter for finding code smells, better focusing human-intensive code reviews. In his original thesis Marinescu proposes rules (detection strategies) for identifying a set of 11 code smells. The precision and recall for Marinescu's classifiers have been studied for the most popular code smell, god classes, and found to be high [1] (precision: 71%, recall: 100%). For the remaining code smell detection strategies there is less or no empirical evidence that the classifiers are identifying the right classes and methods.

Several studies have looked into the relationship between code smells and change proneness. A summary of results is presented in Table 1. Olbrich et al. [9] report that classes infected with the code smells god class and shotgun surgery are more change prone (4-5 times higher for god classes) in two open source projects (Lucene, Xerces). Further, changes to god classes are bigger (e.g. more lines of code are changed) than changes to non-god classes. Following this work, Schumacher et al. [1] investigated the same questions on two commercial applications with the following findings: god classes are more change prone if not normalized. When normalizing results by LOC, god classes are not significantly more change prone. Work by Khomh et al. [8] investigates the change behavior of classes with smells in 10 releases of Eclipse. Their results show that classes infected with code smells (i.e. any code smell) are changed more often. In their analysis they also included the large class code smell that is most similar to the god class code smell. It showed that in five out of ten releases classes with the large class smell were significantly more often changed. The author of this work did not perform a normalized analysis.

Some work has also analyzed the relation of god class code smells and defect proneness. Li et al. [9] analyzed three revisions of the Eclipse project and found god classes to be more defect prone. In a study by Olbrich et al. [2] the authors analyzed three open source projects. It showed that god classes were more often part of defect fixes in two out of the three projects. When normalized by lines of code this result did not hold; god classes were found to be even less defect prone in in two out of the three applications.

3. CASE STUDY

The study was conducted at a small software development company with about 40 employees. The company develops web based applications in C# (using .NET and Visual Studio) and has been appraised at CMMI® Maturity Level 3. Two larger projects that the company developed and currently maintains were selected for analysis of code smells. Key characteristics of the two projects, referred to as project J and project F, are highlighted in Table 2.

Table 2: Project characteristics of the two studied applications

	Project J	Project F
Lines of Code	35,000	45,000
Age (active development)	11 months: Nov 2009 - Sep 2010	17 months: May 2009 - Sep 2010
Number of Developers	4	4
Number of SVN Commits	1259	1611
JIRA issues in category: defect	17	32

3.1 Analysis Procedure for H1-H2

In order to determine whether god classes are changed more often and contain more defects than their non-god counterparts, the information stored in the project's version control system, subversion, and the project's defect tracking system, JIRA, was analyzed. Two measures, the change likelihood [1] and defect likelihood for god and non-god classes can be computed for each of the projects. The following two examples explain this procedure in detail.

3.1.1 Change Likelihood

To understand the change behavior of god classes vs. non-god classes one can ask how likely is it that a god class or non-god class is part of a change to the software.

Table 3: Example for change likelihood for god classes and non-god classes in project F

Revision	1452	1457	1471	1472	1424	Likeli-hood
Changed God Classes	0/4	1/4	1/4	2/4	2/4	0.300
Changed Non-God Classes	1 /223	4 /223	6 /225	4 /225	2 /225	0.015

The above example in Table 3 shows a sample of five revisions of project F. The revision numbers are not consecutive because only the revisions are considered that contain code changes (e.g. some revisions contain changes to documentation or requirements documents). For each of the revisions the number of god and non-god classes is computed using Marinescu's [5] definition. For example, at revision 1452, the inspected project contained four god classes and 223 non-god classes. In general, god classes are rather rare and other papers [1][2][3] have reported similar numbers, with god classes usually comprising less than 10% of the total classes in software systems. At each revision one can then compute the likelihood of a god class being part of this change. For example, at revision 1457, one (out of four) god classes were part of the change; therefore the likelihood for one god class being part of the change is one fourth. The same is done for the non-god classes. The overall project likelihood (last column in Table 3) is then calculated by averaging the likelihoods for each revision. In the example data, one can see that over the five revisions god classes had a 30% chance of being part of a change whereas non-god classes had a 1.5% chance. Following this example, this analysis can be expanded for all revisions of the software project.

3.1.2 Defect Likelihood

A very similar analysis as presented in the previous section can be done for defects. For this analysis we consider all defects that were reported in the project's defect tracking system (JIRA). Further, only the defects that lead to code changes and are completed, i.e. defect fixes, are of interest. In other words, open issues are not considered since their fixes are not completed yet. For each defect fix, one can identify the corresponding software change (i.e. revision) through links provided in the subversion commit comment[1]. Once one knows which files and classes have been included in a particular defect fix one can construct a table as presented below:

Table 4: Example for defect fix likelihood for god classes and non-god classes in project J

Defect (JIRA issue)	J-166	J-161	J-377	J-396	J-228	Likeli-hood
Fix Revisions	9097, 9098	8939	11990	12842, 12844	10269	
God Classes	1/3	0/1	0/8	3/8	0/3	0.1417
Non-God Classes	0/94	1/94	1/156	0/157	1/101	0.0067

In Table 4 a sample of five defects for project J is shown. The first defect fix (J-166) was implemented by two changes (revisions 9097 and 9098), and overall one out of three god classes were modified. No non-god class was changed for this fix. Overall, the likelihood for a god class being part of a defect fix J-166 is one third, and the likelihood for a non-god class is zero. The overall defect fix likelihood in the last column is calculated by averaging the values across each defect fix implemented for the project. For the sample of five defect fixes analyzed and their corresponding revisions the likelihood of a god-class being part of a defect fix is about 14%, whereas non-god classes have a likelihood of 0.7% to be changed.

3.2 Study Results

The results for the two studied applications are presented in Table 5 and Table 6. For each project, the following statistics are presented:

N: is the number of revisions (e.g. a column in Table 3) analyzed. The number of analyzed revisions can differ between god and non-god classes because in some revisions no god classes were present (e.g. early in development). The sample size for change likelihood and defect fix likelihood differs. For the change likelihood the sample size (e.g. number of revisions) is large. For defect fixes the sample size is smaller since the number of reported defects (in JIRA) was rather small.

mean: the arithmetic mean of the values of the sample. This corresponds with the average likelihood presented in Table 3 and Table 4. This number also shows how much more often god classes are changed or are part of defect fixes than non-god classes.

[1] The company's policy requires developers to provide the issue number(s) in their commits to the subversion repository. This information helps in identifying changes, such as defect fixes, for a reported defect.

s: the standard deviation indicates the dispersion of the data from the mean.

p-value: shows the p-value of the two sample t-test. A p-value below 0.05 indicates that the result is statistically significant at our chosen α-level of 5%.

Table 5: Change likelihood for god and non-god classes

	Project F		Project J	
	God Classes	Non-God Classes	God Classes	Non-God Classes
N	545	658	282	328
mean	0.07848	0.01619	0.12565	0.01725
s	0.18448	0.03837	0.24754	0.02391
	p-value: **4.282e-14**		p-value: **2.461e-12**	

Table 6: Defect fix likelihood for god and non-god classes

	Project F		Project J	
	God Classes	Non-God Classes	God Classes	Non-God Classes
N	32	32	17	17
mean	0.03939	0.00956	0.16911	0.00624
s	0.13669	0.01094	0.22266	0.00796
	p-value: *0.2276 (not sig.)*		p-value: **0.008217**	

To test the results for statistical significance a Shapiro-Francia test for normality [11] was first performed ($p<0.001$) on all datasets. The first hypothesis states that the change likelihood of god classes is higher than for non-god classes. The data from both projects (Table 5) supports the hypothesis on a significance level of 5%. In project F, 545 revisions contained god classes, and 658 revisions contained non-god classes. The change-likelihood for god classes is 0.078, meaning that a change to the software required a change to a god class 7.8% of the time. In other words, every 13th change required changing the code of a god class. Comparing this result to the change-likelihood of non-god classes (0.016), one can see that god classes are roughly five times as often part of a change than non-god classes. In project J god classes are changed about seven times as often. Both results are statistically significant.

The second hypothesis investigates whether god classes are more often part of defect fixes. The data from project J supports the hypothesis on a statistical significant level: god classes are about 17 times more often part of defect fixes. In project F god classes are about four times as often modified during a defect fix, however, because of the relatively small sample size, this result is not statistically significant (p-value: 0.22 > 0.05).

3.3 Investigating the Normalization Assumption

The analysis for research questions three and four and previous work [1][2] has used LOC based normalization to account for size differences between god classes and non-god classes. This approach to normalization is a linear transformation that assumes that class size and change likelihood are linearly related with a gradient of 1.0. For example, it assumes that a class A that is five times as large as class B will naturally be changed five times as often. And, concerning defects, class A will be part of five times as many defect fixes. The goal of the following analysis is to investigate whether this assumption is supported by the data in the two analyzed projects.

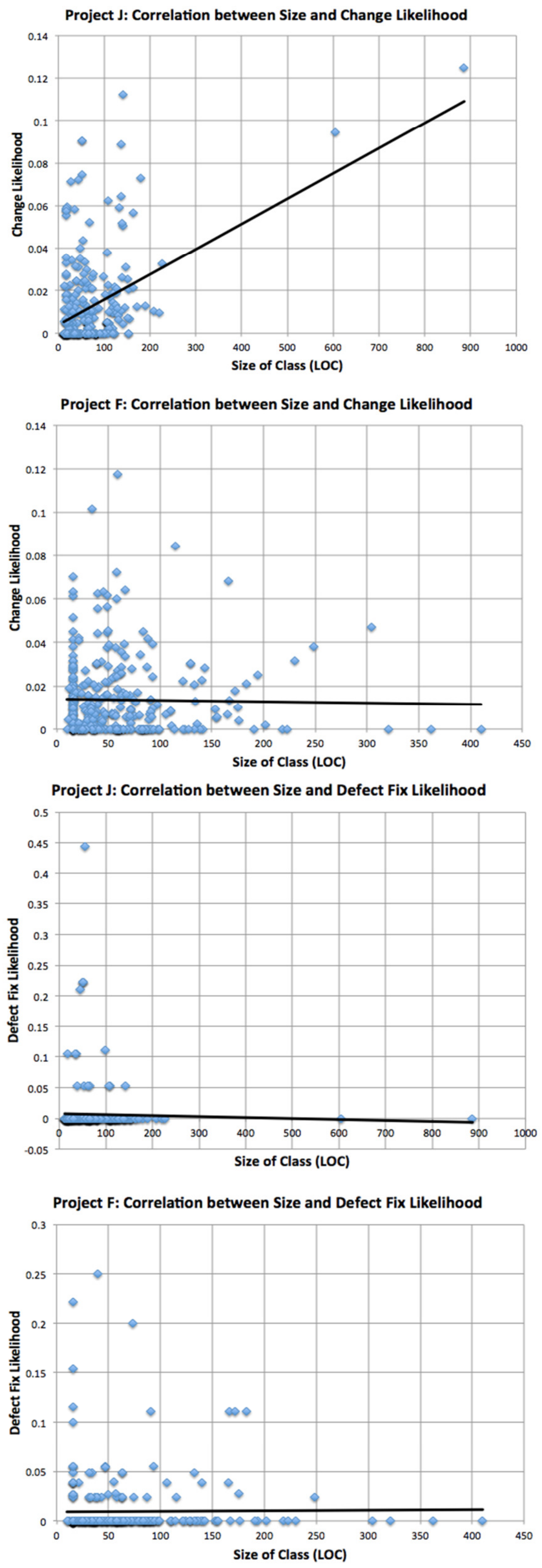

Figure 2: Correlation plots

Figure 2 shows four scatterplots and linear trend lines for the correlation between these variables. In each graph a point represents one code class (i.e. a C# class). The position in the scatterplot indicates how many lines of code the class has at the latest point in time in the repository (x-axis). The top two charts in Figure 2 show the change likelihood and the lower two figures show the defect fix likelihood, both on the y-axis. The higher the value on the y-axis, the more likely it is that the class is part of a change or part of a defect fix.

Table 7: Pearson Correlation Coefficient for graphs in Figure 2

	Project J	Project F
Change and Size	0.42	-0.029
Defect and Size	-0.018	0.011

If the LOC normalizing assumption were true one would expect the trend line in all four figures to have an incline of 1.0 (i.e. a line that dissects each grid in the plot). At first sight one can see that this is not the case.

For project J and the change likelihood (first plot), larger classes are tending to be more often changed (the trend line increases). The gradient of the line is less than 1.0; it is only about 0.8. Therefore in this project, one should normalize by 0.8*LOC (instead of 1.0*LOC). Normalizing by LOC will "over-normalize" the result. Further the correlation coefficient is 0.42, which indicates some degree of correlation between the variables.

For project F and change likelihood (second plot), the trend line is not increasing, rather it is slightly decreasing. For example, the three largest classes in the plot are never changed (they are only checked into the repository) and have therefore a change likelihood of 0.0. On the contrary, the two classes with the highest likelihood are rather small with around 50 lines of code. This observation leads to conclude that normalizing this data is the wrong approach, and will falsely lead to a less significant result. Lastly, the correlation between the two variables cannot be supported by Pearson's correlation coefficient of -0.029.

Similar observations can be made for the correlation between class size and defect fix likelihood. In both projects the linear trend line does not increase. Therefore, the assumption that larger classes are more often part of defect fixes cannot be supported by the data. The correlation coefficient is in both cases very close to 0, indicating no correlation between defect fix likelihood and class size. We conclude that also in this case normalizing the data will falsely lead to less significant results.

To illustrate the impact of LOC normalization we followed the example of our and other earlier work and normalized the results by lines of code. The LOC-normalized likelihood is computed by dividing each of the cells in Table 3 by the average lines of code (LOC) of the changed classes. For the god classes, this measure now represents the likelihood of a line of code being changed in a god class during a revision. The same is done for the non-god classes, and for defect likelihood.

In this situation (see Table 8), god classes in both projects are still changed more often, but only 1.2 times more often in project F and 1.5 times more often in project J. Further, the result is only of statistical significance in project J. A change towards less significant results can also be seen for the normalized data for defects (Table 9). In project J, a line of code in a god classes is 10 times more likely changed (instead of 17 times) than in a non-god class. This result is still statistical significant.

4. DISCUSSION

The first part of the results showed, as presented in previous work, that god classes are changed significantly more often than their non-god counterparts. This implies that it is appropriate to view god classes as instances of technical debt, as they appear to increase future maintenance costs in several ways. They require more change (both defect-related and non-defect-related change), so it is reasonable to assume that refactoring the god class would eliminate the need for some of those changes. This is a bit of a conceptual leap, as our analysis only shows correlation, not causality. Also, splitting up a problematic class into multiple classes could lead to generating multiple smaller problematic classes that, on a system level, still need to be changed more often. However, even if we don't assume that the number of changes would decrease through refactoring, making frequently-changed classes more understandable and less complex would reduce the effort to perform many of those changes. Refactoring, in theory, results in easier to maintain classes. So applying refactoring to classes that are changed frequently (i.e. god classes) is a strategy to maximize the positive effect on maintenance costs.

The second part of the analysis concentrates on an important threat to validity identified in prior work. It seems reasonable that a potential threat to validity of our initial analyses, and similar analyses in prior work [1][2] would be the possibility that god classes are more change- and defect-prone simply because they are larger and encompass more code and functionality that might contain problems. The next logical step would be to normalize the data by code size in order to remove this bias. This approach was taken in [1][2], as well as other studies in empirical software engineering. However, before applying this approach ourselves, we first investigated the underlying assumption, i.e. that there is an increasing linear relationship between size and defect-

Table 8: LOC normalized change likelihood for god and non-god classes

	Project F		**Project J**	
	God Classes	**Non-God Classes**	**God Classes**	**Non-God Classes**
N	545	658	282	328
mean	0.00024	0.00020	0.00029	0.00019
s	0.00080	0.00062	0.00058	0.00048
	p-value: *0.3097 (not sig.)*		p-value: **0.02991**	

Table 9: LOC normalized defect fix likelihood for god and non-god classes

	Project F		**Project J**	
	God Classes	**Non-God Classes**	**God Classes**	**Non-God Classes**
N	32	32	17	17
mean	0.00012	0.00008	0.00040	0.00004
s	0.00046	0.00009	0.00052	0.00006
	p-value: *0.6513 (not sig.)*		p-value: **0.01347**	

likelihood (and change-likelihood). But in fact we found the support for this assumption to be rarely present in the datasets. Further, when we did normalize our data and tested our hypotheses again using normalized defect- and change-likelihood, we found less significant results. Thus normalization by code size does not provide useful information in our case, and by extension in other cases in which a linear relationship between size and quality characteristics is not evident. This leaves open the question of whether other types of normalization might be appropriate for this type of analyses.

5. THREATS TO VALIDITY

This small-scale case study bears typical threats to validity. Analyzing two commercial systems is not sufficient to draw general conclusions. However, with the help of the related work of Schumacher et al. [1], Olbrich et al. [2][3], and Khomh et al. [8] this study provides additional evidence that god classes have a negative impact on maintainability in several commercial and open source systems.

A further limitation is the focus on one specific code smell that is arguably easy to grasp, but is not the only indicator of pending design debt in a software system. More design debt symptoms need to be studied in order to validate that these types of indicators are good predictors of technical debt in general.

As for the internal validity of the analysis one can remark that these separate studies used different analysis methods depending on the source data and resolution of the analysis technique. On the one hand this will prevent comparing the exact numerical results directly, on the other hand more compelling evidence is built when different analysis techniques lead to similar resulting trends.

6. CONCLUSION

In this paper we have investigated questions regarding the impact of god classes on software quality, with the broader aim of exploring them as indicators of design debt. Our study analyzed two real world commercial software systems and investigated in detail whether god classes are more change- and defect-prone. The results complement earlier work and indicate that god classes are in general more change prone and in some cases more defect prone. This is a strong indicator that the god class smell is important to monitor and manage in software development projects, and that they are in fact related to technical debt. The evidence from this research shows that these classes will require more maintenance effort by requiring more changes, more changes related to fixing bugs, and more effort per change (due to their inherent complexity and conceptual size). These results are supported by prior research. To conclude, god classes have been confirmed to have the technical debt property of incurring interest since they "cost" the organization more in terms of changes and defects as the software development lifecycle progresses. We recommend that god classes be identified early and often in the lifecycle and reviewed for potential refactoring (i.e. to pay off the technical debt) on a regular basis.

As a second contribution we have argued against the usefulness and correctness of normalizing analysis results by a code size measure. Our data indicates no correlation between defect likelihood and class size, and only in one case a slight correlation between change likelihood and class size. Future work should consider this finding and report on correlation coefficients before normalizing data.

7. ACKNOWLEDGMENTS

We would like to thank Kathleen Mullen and Rick Flagg for their valuable feedback. This research was supported by NSF grant CCF 0916699, "Measuring and Monitoring Technical Debt " to the University of Maryland, Baltimore County.

8. REFERENCES

[1] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. 2010. Building empirical support for automated code smell detection. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA

[2] Olbrich, S.M., Cruzes, D.S., Sjoberg, D.I.K. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. Software Maintenance, ICSM 2010, pp1-10, Timisoara

[3] Olbrich, S., Cruzes, D., Basili, V., Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on , 390-400.

[4] Lanza, M., Marinescu, R. 2006. Object-oriented metrics in practice. Springer

[5] Marinescu, R. 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In Proceedings of the 20th IEEE international Conference on Software Maintenance (September 11 - 14, 2004). ICSM. IEEE Computer Society, Washington, DC, 350-359.

[6] Fowler, M., Beck, K. 1999. Refactoring: improving the design of existing code. Addison Wesley.

[7] http://c2.com/xp/CodeSmell.html, last retrieved Jan 27, 2011

[8] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE '09). IEEE Computer Society, Washington, DC, USA, 75-84.

[9] Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw. 80, 7 (July 2007), 1120-1128.

[10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. 2010. Managing technical debt in software-reliant systems. In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). ACM, New York, NY, USA, 47-52

[11] Thode Jr., H.C.: Testing for Normality. Marcel Dekker, New York, 2002