# Prioritizing Design Debt Investment Opportunities

Nico Zazworka
Fraunhofer Center for Experimental Software Engineering
5825 Univ. Research Ct., Suite 1300
College Park, MD, USA
+1 240-287-2928
nzazworka@fc-md.umd.edu

Carolyn Seaman
UMBC
1000 Hilltop Circle
Baltimore, MD, USA
+1 410 455 3937
cseaman@umbc.edu

Forrest Shull
Fraunhofer Center for Experimental Software Engineering
5825 Univ. Research Ct., Suite 1300
College Park, MD, USA
+1 240-287-2925
fshull@fc-md.umd.edu

## ABSTRACT
Technical debt is the technical work developers owe a system, typically caused by speeding up development, e.g. before a software release. Approaches, such as code smell detection, have been developed to identify particular kinds of debt, e.g. design debt. Up until now, code smell detection has been used to help point to components that need to be freed from debt by refactoring. To date, a number of methods have been described for finding code smells in a system. However, typical debt properties, such as the value of the debt and interest rate to be paid, have not been well established. This position paper proposes an approach to using cost/benefit analysis to prioritize technical debt reduction work by ranking the value and interest of design debt caused by god classes. The method is based on metric analysis and software repository mining and is demonstrated on a commercial software application at a mid-size development company. The results are promising: the method helps to identify which refactoring activities should be performed first because they are likely to be cheap to make yet have significant effect, and which refactorings should be postponed due to high cost and low payoff.

## 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 (TD) is a metaphor describing the tradeoffs between technical development activities that are delayed in order to get short-term payoffs, such as a timely software release. The term "debt" is used to describe how these skipped activities might pay off in the short run, such as going into financial debt to buy a new car, but will have to be paid back later. If too much technical debt accumulates in a software project, development will slow down, e.g. due to increasingly poor maintainability of the code.


*MTD*'11, May 23, 2011, Waikiki, Honolulu, HI, USA


### 1.1 Goals and Recent TD Research
Recent research in technical debt has arrived at a point where it is possible to identify certain classes of technical debt even in large systems with computer-assisted methods. One example of this ongoing research is the identification of code smells [5][6], which point to software components that are less than ideally designed and embody design debt. It is believed that if too much debt accumulates in a software system the ease of maintenance will significantly degrade.

To date, empirical studies of technical debt have demonstrated that detection approaches can succeed in finding issues that are of value to developers [1]. An important next step is to provide more guidance for decisions about whether or not to pay off particular instances of design debt at a given point in time. As explained in [7], this decision depends on various factors, such as the value of the debt, the interest rate one currently pays on the debt, and potential impact that the debt has on future development. Typically, debt will not be paid for the purpose of paying debt but for the purpose of making a lucrative investment in the software. As an example, one does not want to pay design debt in a legacy system that is guaranteed to be left unchanged in the future.

Paying off design debt and adapting the architecture towards new requirements is called refactoring. Refactoring means adjusting the design without changing the external behavior of a program. The cost to pay off this type of debt is in most cases the time spent in the refactoring activity itself (e.g. planning the design and architecture, rewriting code, adjusting documentation). In many software projects the cost and benefit of refactoring is not easily quantified or estimated. This makes it difficult for developers to justify and communicate refactoring needs to the management level because it's not clear how to prioritize refactorings and or if the refactoring will pay off.

### 1.2 Criteria for Paying Design Debt
As outlined in the previous section, reasonable decision criteria for paying off design debt are, on the one hand, the cost to successfully refactor the design and, on the other, the rate at which the current debt decreases the software quality and productivity (in the following called quality characteristics). A logical strategy for paying debt is to refactor the debt items that are inexpensive to fix and promise to have the biggest positive impact on the quality characteristics. On the other hand, one would like to defer paying debt for such items that are expensive to fix and only promise small, or no, gain in software quality. Therefore, an approach for quantifying and prioritizing design debt should incorporate both dimensions: cost of refactoring and the quality gain from the refactoring.

In this paper, we will propose and exemplify a method for prioritizing refactorings of classes with a particular type of design

debt (i.e. god classes) in order to recommend which debt items should be paid first, and which are better to leave in the system.

## 1.3 Research Question

The research question this work is attempting to answer can be formulated as:

Q1: Given a set of technical debt issues (in this case, design debt expressed through god classes), how can we prioritize and decide which god classes to refactor based on estimating cost and impact of the refactoring?

This paper will be limited to describing the approach and giving a realistic example from an industrial software application, but will not provide a full validation of the proposed approach (other than common sense argumentation). It will be up to future research to evaluate this approach appropriately.

## 1.4 Code Smells and God Classes

The proposed method focuses on god classes as technical debt indicators. The god class code smell describes classes that over-centralize functionality[1] and have multiple responsibilities[2] in a system and are more concerned with the data of other classes than their own data[3]. God classes can be identified using Marinescu's [5] detection strategy that uses three software metrics, as shown in Figure 1. Details on metric implementation are presented in [1][2][3][4].

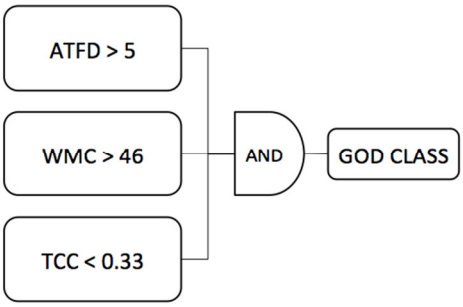


**Figure 1: Detection strategy for god classes as presented in Lanza and Marinescu [4]**

# 2. COST OF PAYING DESIGN DEBT

To pay off technical debt caused by a god class one will have to refactor this class. Typical refactoring activities for god classes are:

- Splitting up of the god class into multiple classes
- Simplifying the god class (freeing it of unused code)
- Moving parts of the functionality (e.g. methods) into classes where they better fit

The cost of the refactoring can be estimated by investigating more closely the measures that the detection strategy (in Figure 1) uses for god class identification. Previous studies [1] have built evidence that this metric based model is useful, i.e. that the model does in fact identify problematic modules. Therefore, we propose to further leverage the components of the model for estimation of the size of the change with the following argumentation.

The metric based model uses linear metrics and thresholds. One can make an argument that classes that are only slightly outside the thresholds will be easier to refactor than classes that are multiple magnitudes outside the thresholds. For example, a class having a WMC value of 50 (and therefore only four points above the threshold) will be potentially easier to refactor than a class with a WMC of 150. In the first case, deleting some unused functionality, or simplifying some decision structures will be sufficient to lower the class' WMC value. In the second case these simple transformations are likely not to be sufficient. A class with a WMC of 150 will have to be split up into multiple classes. This refactoring activity will possibly impact further classes (e.g. classes accessing the god class) and will therefore be more expensive to implement.

The same argumentation can be made for the metric ATFD. A class that accesses only a few more attributes from other classes than the suggested threshold of five will be potentially easier to refactor than a class that access 50 attributes from other classes. A similar argument can be made for the last metric, TCC.

The proposed method to approximate the size of the refactoring will rank the potential cost to refactor classes towards improved WMC, TCC, and ATFD metrics. The metrics and threshold originally proposed by Marinescu [5] will give in most cases a good starting point as shown in [1]. In some cases, it will become necessary to tailor the thresholds towards the application, e.g. code for a web frontend might require different thresholds than code for a database backend. The proposed method will also work with tailored metric thresholds.

The calculation of the overall effort rank will be illustrated by example. Table 1 shows the metrics involved in the god class detection for all nine identified god classes in the feasibility study project. First, for each class and each of the three metrics the rank within the metric (columns) is determined. The closer a class is to the threshold of each metric, the lower will be the assigned rank. For example, the god class `GodClass7`[4] has the lowest WMC value of 47 and will be ranked as first for that metric. After ranking all three metrics the total rank will be determined by adding the three ranks for each god class and ranking this result. This ensures that each metric is given the same importance. In the example, `GodClass7` receives a rank of 1 because it has values of WMC (47), TCC (0.219), and ATFD (7) that are closer to the detection strategy's thresholds than any other god class. A refactoring towards a non-god class should be inexpensive. The potentially most expensive class is `GodClass3`, having WMC, TCC, and ATFD values of 107, 0.0 , and 28.

# 3. IMPACT OF DEBT ON QUALITY

The second dimension of the approximation method determines the impact a god class has on a set of quality characteristics. The two characteristics considered in our example are:

- Correctness, represented by the *defect likelihood* and
- Maintainability, represented by *change likelihood* of a god class.

[1] The large amount of implemented functionality is represented by the complexity measure: weighted method count (WMC)

[2] Multiple responsibilities are characterized by low internal cohesion of the class and are reflected in the tight class cohesion metric (TCC)

[3] The use of other class data (either through member access, or through getter access) is reflected in the access to foreign data metric (ATFD)

[4] Real class names of the application were anonymized for security and privacy reasons.

**Table 1: Refactoring effort ranking based on detection strategy metrics**

| God Class Name | WMC (>46) | | TCC (<0.33) | | ATFD (>5) | | Overall Score and Rank | |
|---|---|---|---|---|---|---|---|---|
| | Value | Rank | Value | Rank | Value | Rank | Rank Sum | **Rank** |
| GodClass1 | 49 | 3 | 0.0 | 8 | 20 | 6 | 17 | 6 |
| GodClass2 | 87 | 8 | 0.005 | 7 | 28 | 7 | 22 | 7 |
| GodClass3 | 107 | 9 | 0.0 | 8 | 28 | 7 | 24 | 9 |
| GodClass4 | 69 | 7 | 0.026 | 6 | 34 | 9 | 22 | 7 |
| GodClass5 | 49 | 3 | 0.065 | 5 | 9 | 3 | 11 | 3 |
| GodClass6 | 60 | 5 | 0.177 | 4 | 19 | 4 | 13 | 4 |
| GodClass7 | 47 | 1 | 0.219 | 1 | 7 | 1 | 3 | 1 |
| GodClass8 | 48 | 2 | 0.199 | 2 | 7 | 1 | 5 | 2 |
| GodClass9 | 61 | 6 | 0.192 | 3 | 19 | 4 | 13 | 4 |

**Table 2: Software quality characteristic ranking based on change and defect likelihood**

| God Class Name | Change Likelihood | | Defect Likelihood | | Overall Score and Rank | |
|---|---|---|---|---|---|---|
| | Value | Rank | Value | Rank | Rank Sum | **Rank** |
| GodClass1 | 0.016 | 1 | 0.0 | 1 | 2 | 1 |
| GodClass2 | 0.097 | 8 | 0.0 | 1 | 9 | 4 |
| GodClass3 | 0.102 | 9 | 0.029 | 5 | 14 | 9 |
| GodClass4 | 0.068 | 7 | 0.177 | 6 | 13 | 7 |
| GodClass5 | 0.040 | 3 | 0.0 | 1 | 4 | 3 |
| GodClass6 | 0.0455 | 4 | 0.133 | 7 | 11 | 5 |
| GodClass7 | 0.0458 | 5 | 0.133 | 7 | 12 | 6 |
| GodClass8 | 0.052 | 6 | 0.133 | 7 | 13 | 7 |
| GodClass9 | 0.027 | 2 | 0.0 | 1 | 3 | 2 |

Correctness of a class can be estimated by the defect likelihood measure similar to the one defined in [1]. For a god class one can compute how many defect fixes affected this god class by mining the code repository and issue tracker. More specifically, one will account for the time the class was a god class (e.g. from May to September), then count the number of defects that lead to fixes in the god class in this time period, and divide by the number of all defects that were fixed in this time period. The higher the resulting value the more likely a defect manifests in the god class, e.g. a likelihood of 0.5 would indicate that every second defect fix leads to changes in this god class.

Maintainability can be estimated by investigating how often a class is changed. For this the change likelihood, as defined in [1], indicates how likely a class is to be modified when a change to the software is made. For example, a change likelihood value of 0.1 shows that the class was, on average, modified with every $10^{th}$ change (i.e. revision) to the software. Three arguments can be given why high change likelihood for god classes compromises maintainability:

- Classes with high change likelihood indicate that changes to the software lead to repetitive changes in a specific class. This points to a maintainability issue since the class has to be changed too frequently.
- Classes that are often changed should have superior understandability and extendibility to enable developers to make a change within the least amount of time. Therefore, a high change likelihood should (if at all) manifest in classes that are easy to modify and understand. God classes, on the contrary, are classes that are complex and hard to understand and modify.
- Classes that were changed more often in the past are also more likely to be changed in future. This corresponds with the technical debt principle that only such debt should be paid that will have a sufficiently positive future impact. A god class that is likely to be changed often in future is a prime subject for refactoring.

Both metrics, change and defect likelihood, can be calculated for each of the god classes and the results can be ranked using an approach similar to that described in Section 2.1. The results for our example are shown in Table 2.

# 4. FEASIBILITY STUDY

To investigate the feasibility of the method, one sample application from a small-size software development company (CMMI® Level 3) with about 40 employees, was selected. The company develops web-based applications in C# (using .NET and Visual Studio). Key characteristics of the project are highlighted in Table 3.

**Table 3: Project characteristics of the studied application**

| | |
|---|---|
| Lines of Code | 35,000 |
| Age (active development) | 11 months:<br>Nov 2009 - Sep 2010 |
| Number of Developers | 4 |
| Number of SVN Commits | 1259 |
| Number of reported defects (JIRA) | 17 |

## 4.1 Design Debt Cost Benefit Matrix

Ranking of refactoring effort and quality impact was done, as explained in Section 2.1 and 2.2. To understand the new insights one can plot the ranking results on a matrix (see Figure 2), similar to a scatter plot. The two axes correspond to the two ranking dimensions: refactoring effort and quality impact.

Looking at Figure 2 one can make following observations: classes that fall on the diagonal tend to have balanced effort/impact ranking characteristics. For example, `GodClass3` is the most expensive to fix but is also likely to impact quality characteristics most. Classes above the diagonal are the most promising to refactor since the impact is ranked higher than the effort. For example `GodClass7` and `GodClass8` are potentially inexpensive to refactor and have a relatively high negative impact on software quality. On the contrary, classes below the diagonal

tend to have little impact and high refactoring cost. This debt is likely to have low interest (i.e. low impact on quality) and high value. For example, `GodClass1` is likely to be one of the more expensive ones to fix but does show only little negative impact. Fixing this debt can be deferred to a later point in time.

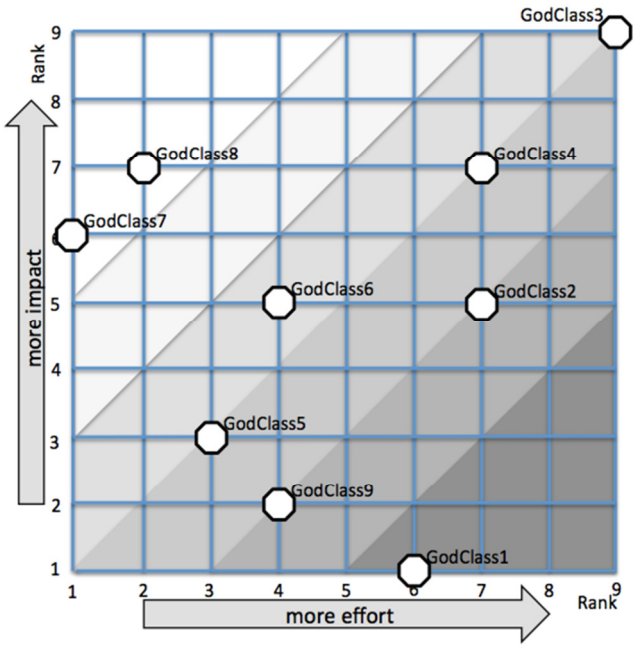


**Figure 2: Design Debt Cost Benefit Matrix for god classes in project J**

If a total ranking is desired (e.g. to create a list ordered by profitability of the refactoring) one can compute a profitability measure by subtracting the effort rank of a class from its quality impact rank. A positive value will be assigned to classes above the diagonal, and a negative value will be assigned to the classes below the diagonal. Classes on the diagonal will receive a value of zero. Such a measure, however, obscures more detailed, and valuable, information about the cost and impact of each debt item.

## 5. FUTURE WORK

The ranking approach is amenable to multiple extensions. First, it is extendable to other code smells defined in a similar way (e.g. the definitions in [4]). Second, one can think of adding, replacing and weighting different quality characteristics for the quality ranking.

Empirical evaluation will be required in the future to build compelling evidence that god classes with worse metrics require higher refactoring effort. We recommend that future empirical studies investigate if the above ranking is consistent with the actual measured effort of refactoring activities. Further, the choice of the quality characteristics and their impact (i.e. weight) in the ranking process needs to be validated. Studies need to be conducted to confirm that the predicted impact ranking correlates with the experience of software experts (i.e. developers of the software system). This research will help to ultimately move from this ranking approach towards prediction models that help in estimating the absolute value and cost of refactoring.

Lastly, more qualitative data should be collected (e.g. through interviews with developers) to understand *why* certain god classes have a higher impact on quality than others, and which short term gains, if any, were made by creating a god class in first place.

**Table 4: Total Ranking**

| Class | Effort Rank | Impact Rank | Overall Score |
|---|---|---|---|
| GodClass7 | 1 | 6 | 5 |
| GodClass8 | 2 | 7 | 5 |
| GodClass6 | 4 | 5 | 1 |
| GodClass3 | 9 | 9 | 0 |
| GodClass4 | 7 | 7 | 0 |
| GodClass5 | 3 | 3 | 0 |
| GodClass2 | 7 | 5 | -2 |
| GodClass9 | 4 | 2 | -2 |
| GodClass1 | 6 | 1 | -5 |

## 6. 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.

## 7. 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] 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.