Building Empirical Support for Automated Code Smell Detection

Jan Schumacher[1], Nico Zazworka[2], Forrest Shull[2],
Carolyn Seaman[2,3], Michele Shaw[2]

[1]University of
Applied Sciences
Mannheim
Germany
+49 621 2926224
jan@schuma.eu

[2]Fraunhofer Center
College Park
MD, USA
+1 240 487 2904
{nzazworka, fshull, mshaw}@fc-md.umd.edu

[3]UMBC
Baltimore
MD, USA
+1 410 455 3937
cseaman@umbc.edu

ABSTRACT

Identifying refactoring opportunities in software systems is an important activity in today's agile development environments. The concept of code smells has been proposed to characterize different types of design shortcomings in code. Additionally, metric-based detection algorithms claim to identify the "smelly" components automatically. This paper presents results for an empirical study performed in a commercial environment. The study investigates the way professional software developers detect god class code smells, then compares these results to automatic classification. The results show that, even though the subjects perceive detecting god classes as an easy task, the agreement for the classification is low. Misplaced methods are a strong driver for letting subjects identify god classes as such. Earlier proposed metric-based detection approaches performed well compared to the human classification. These results lead to the conclusion that an automated metric-based pre-selection decreases the effort spent on manual code inspections. Automatic detection accompanied by a manual review increases the overall confidence in the results of metric-based classifiers.

Categories and Subject Descriptors

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

General Terms

Measurement, Design, Empirical Study, Verification.

Keywords

Code Smells, God Class, Code Inspection, Maintainability

1. INTRODUCTION

Code smells (first introduced by Fowler and Beck [3]) are an established concept to classify shortcomings in software that follows object-oriented design. Each "smell" is an indicator that points to the violation of object-oriented design principles such as data abstraction, encapsulation, modularity, and hierarchy.

One of the code smells introduced by Fowler and Beck is the large class or god class smell. God classes do too much, often have more than one responsibility, and only delegate minor tasks to other classes [3] [6]. To ensure that object-oriented software remains easy to understand and maintainable over time, Fowler and Beck argued that these classes should therefore be split up into multiple classes, or else sub-classes should be extracted from the god class.

Our study aims to improve the understanding how humans utilize the concept ofcode smells in improving software quality. We investigate how expert developers detectgod classes and whether it is a repeatable process (e.g., whether there is agreement among multiple judges). Based on observations of real developers and real systems, we can begin to formulate the symptoms that let humans identify agod class as such.

We use these results to investigate the feasibility of tools for supporting this process. Fowler's and Beck's intention was that the detection ofcode smells would be based on human judgment and intuition. However, researchers (e.g., Marinescu[11]) have also proposed a metric-based approach for the detection of code smells.

Furthermore, we investigate whether there are measurable effects (e.g., greater change frequency) ofgod classes in the software system that would corroborate Fowler and Beck's contention that correcting these symptoms will lead to improvements in software maintainability.

2. CONTEXT AND RELATED WORK

The term Code Smell was first introduced by Kent Beck and gained popularity through Fowler's and Beck's book on refactoring [3]. They describe refactoring as "Improving the design (of software) after it has been written". This can become necessary since deadline pressure, strong focus on functionality, and inexperienced developers can have a negative influence on the design of a software system.

Ward Cunningham notes that these factors, often present in hasty software development, may lead to the build-up of what he refers to as technical-debt [1]. On the one hand it can be tolerable to build up such debt in order to progress more rapidly in the development of a product. On the other hand, the debt may reach a level where the interest paid in the form of code that becomes

ESEM'10, September 16-17, 2010, Bolzano-Bozen, Italy.
Copyright 2010 ACM 978-1-4503-0039-01/10/09...$10.00.

hard to understand, and deviations from the design, will outweigh these short-term benefits. Refactoring can be used to transition the design of a system that suffers fromtechnical debt into a more favorable one.

However, there are situations, when it is less obvious where and what kind of refactoring would improve a certain part of a software system. Fowler and Beck use the concept of code smells to help developers identify design flaws in their software [3]. Each code smell has a set of refactoring techniques associated with it. Strategies for using and evaluating Code Smells can be grouped into automated, and manual approaches.

2.1 Automatic Detection of Code Smells

The detection of code smells has been described by Fowler and Beck as a human discipline based on intuition and experience [3]. Lanza and Marinescu argue [6] that the manual detection of code smells is time consuming, non-repeatable and does not scale. In conclusion, they propose detection strategies; a metric-based approach to detectcode smells automatically. In [11]Marinescu conducts a case study to assess the performance of detection strategies. He tries to find 10 differentcode smells in a medium size business application. The suspects that were found using detection strategies are examined again by humans. Based on their opinion, the precision of the automatic detection is reported to be 70%. Recall is not calculated. Recent work by Olbrich et al. [14] also useMarinescu's detection strategies to investigate the relationship between certaincode smells and an increased maintainability effort. They report increased maintainability effort forgod classes.

Munro et al. [13] refine Marinescu's method by suggesting a more systematic approach. This includes providing empirical evidence for choosing the software metrics used for the automatic detection. They also propose new metrics based on the characteristics and design heuristics associated with thecode smell. By automatically detecting the lazy class and temporary field smell in two academic applications (1500LOC and 16000 LOC) Munro et al. determine the performance of their model. For the smaller of the two software systemscode smells were also detected manually. Due to the low number of false-positives (high precision) they conclude their model performed effectively.

Other work by van Emden [2] uses the extracted meta-model of a software system to detect code smells. She automatically detectscode smells in software that does automatic facial recognition as part of a case study. She reports the developers' feedback to be "generally positive". The developers also found the additional visualizations the tool provided "were useful for conformance checking and refactoring support."

Mohaet. al[12] designed a comprehensive framework for the automatic detection ofcode smells. It includes a component for formally describing acode smell using adomain specific language. This description is not limited to metrics and is later on used to automatically generate detection algorithms for thecode smell. The authors validate their model using the Java framework Xerces. In addition, to automatic detection, the system was evaluated by three students on graduate level and two independent software engineers for the existence of certaincode smells. Based on these two assessments, Moha et al. report 100% recall and an average precision above 40%.

In addition, to metric-based detection, Kreimer [4] proposes an adaptive detection technique that uses manual classification by developers to train automatic detection algorithms through machine learning. By doing this he tries to "overcome problems caused by different perceptions of CS (code smells) by different developers." Using a relatively small training set (20 instances), the accuracy of the model is reported to be between 95% and 100% for the long method and large class smell. Running the same detection algorithm on 20 sample program locations that were also evaluated by a human for the existence of the two smells, Kreimer reports precision to be between 80% and 90%.

2.2 Human Detection of Code Smells

Compared to the approaches for the automatic detection of code smells, the way they are detected by humans has not been thoroughly explored. To date, Mäntylä et al. have done the most comprehensive investigation on human performance of the classification of code smells. In two publications [9][10], they investigate the subjective evaluation of code smells using an empirical study. The study is run in a small software development company. Mäntylä et al. analyze the effect of demographic factors, such as experience and capabilities, on the manual detection ofcode smells. They conclude that demographics, to some extent, may be used to explain fluctuations in the smell evaluation. In addition, they investigate if there is agreement on the existence of code smells in the reviewed software modules among the judges. Here they report that the conflicting perception ofcode smells among the developers caused a lack of uniformity in the smell evaluations. Lastly they try to find correlations between code smells identified by the subjects and software metrics. However, no significant correlations were found.

Mäntylä ran two more studies [7][8]using students to review a purposely poorly programmed small application (1000 NLOC[1])

In the first publication [8], Mäntylä et al. examined the inter-rater agreement for the existence of three method level code smells and the question if subjects would refactor the method. Agreement was high for simple code smells like Long Method and Long Parameter List. For the more complicated Feature Envy code smell and the refactoring decision, there was significantly lower agreement. In addition, they tried to find explaining factors for the above results using a metric-based regression model and demographic information on the subjects of the study. The regression model explained about 70% of the code smell detections. Only 40% of the refactoring decisions could be explained based on the model.

Mäntylä et al. concluded that metrics and demographics are not ideal predictors for refactoring decisions. In the second study [7], they investigated the human rationale behind the refactoring decisions. They accomplished this by coding the answers they received and assigning them to different categories. Again, a regression model was used to determine how well these drivers explain the refactoring decision. The inter-rater agreement for the different categories was reported as poor. So the subjects did not have a uniform opinion as to what kind of issues existed in the examined methods. The regression model showed that qualitative data was a valuable explaining factor for the refactoring decision. Method Length in particular turned out to be the best predictor for the subjects refactoring decision.

[1] Non-commented lines of code

2.3 Summary of Related Work

The work presented above shows that automatic code smell detection yields reasonable results. Yet, except for Moha, none of the publications present numbers for the recall rate of their approach when compared to manual detection. This means that true-positives, so classes that a human might have considered god classes, might be missed by the automatic detection. In the publications covered so far, manual detection ofcode smells plays a secondary role and was used to assess the performance of an automatic detection technique. The information on how the detection was performed by humans is relatively sparse. Only one author (Mäntylä) reports on this topic.

3. RESEARCH QUESTIONS

We believe the work done by Mäntylä et al., especially their evaluation of drivers behind refactoring decisions, yields interesting results. Consequently, we decided to conduct a similar study where the refactoring decisions made by humans were not directly evaluated but one of the most prominent reasons for refactoring identified by Fowler and Beck:large classes orgod classes. The authors describegod classes as "prime breeding ground for duplicate code, chaos, and death" [3]. According to Lanza and Marinescu, a god class is "an aggregation of different abstractions and (mis)use other classes (often mere data holders) to perform its functionality" [6].

This leads to the common belief that god classes have a negative effect on the understandability and evolvability of a software system. Due to their expansive impact on a software system, god classes represent a code smell worth examining. Tool supported detection of god classes has been covered in some of the publications presented above ([11][13]), there has been no research on the human perception and performance of detecting god classes.

Our study builds on and extends Mäntylä's work. It is run in a professional environment, and the subjects are the actual developers of the software under review. This is an important component to the work, since the subjects are familiar with the software they are reviewing. They know the design decisions that were made for the software and are aware of the constraints that existed during the development phase.

3.1 Evaluation of Human Performance

Our research questions focus on evaluating the human performance and perception of the process of detecting classes infected by thegod classcode smell in a professional environment. Answering these questions will help to understand how much effort is required when using a code review to identify such smells. Also, it will be shown how high agreement between the human evaluators can be expected using reviews. In addition, the characteristics and code issues (expressed by the reviewers) in a class that are associated with a specific code smell are collected.

R1: How difficult is it for humans to identify god classes and how much effort does it take?

R2: How well do humans agree on identifying god classes?

R3: What issues in code make humans identify a god class as such?

3.2 Evaluation of Automated Classifiers

In addition, we evaluate if any of the issues that were used by the subjects to identify god classes can be linked to metrics used by automated classifiers. We investigate if we can optimize the metrics and thresholds used by automatic classifiers based on the findings from the first set of research questions.

R4: How well do the previously proposed metric based classifiers perform in terms of precision and recall when compared to human classification?

R5: How are the identified code issues related to the metric based classification approach?

R6: Is it possible to improve existing metric based approaches?

3.3 Effects of God Classes on Maintainability

The last research question aims at finding evidence that classes identified with thegod class smell demand an increased level of maintainability effort. Such findings would corroborate Fowler and Beck's contention that correcting these symptoms can lead to improvements in the software's maintainability.

R7: Do god classes require a higher maintainability effort than non-god classes?

4. STUDY DESIGN

4.1 Setting

The study was conducted in a professional environment at a midsized software development company located in Washington D.C. Most of the company's projects are web applications written in the C# programming language and are based on Microsoft's ASP.NET framework. Two of the company'songoing projects (project A and B) were selected as candidates for the study. The two projects are database-driven web applications that enable the user to manipulate data through a browser. Both projects are based on a common architecture that is developed across all the projects in the company. Moreover, the projects have undergone a full product lifecycle (elicitation, design, implementation) and have booth been maintained by the company for more than a year. Additional characteristics of projects A and B can be found in Table 1.

Table 1: Project characteristics of the two systems

	Project A	**Project B**
# Classes	154	105
Lines of Code*	11264	5676
Avg. LOC / Class	73	55
# Developers	2	2
Project History in SVN	10 months: 1268 revisions	4 months: 645 revisions

* based on the total size of methods in a class, including blank lines and comments [6].

4.2 Evaluation of Human Performance

The design of the study was done in an iterative fashion. Three pilot studies were conducted with computer science students at graduate (1 student) and undergraduate level (2 students). These studies helped to identify flaws in the procedure and allowed for a step-wise refinement of the inspection process used.

A total time of 90 minutes was allotted for the study for each participant, where the inspection of classes should not take longer than 75 minutes. The second class and third class were inspected for project A and project B respectively. This selection strategywas chosen to assure a widespread of classes over all

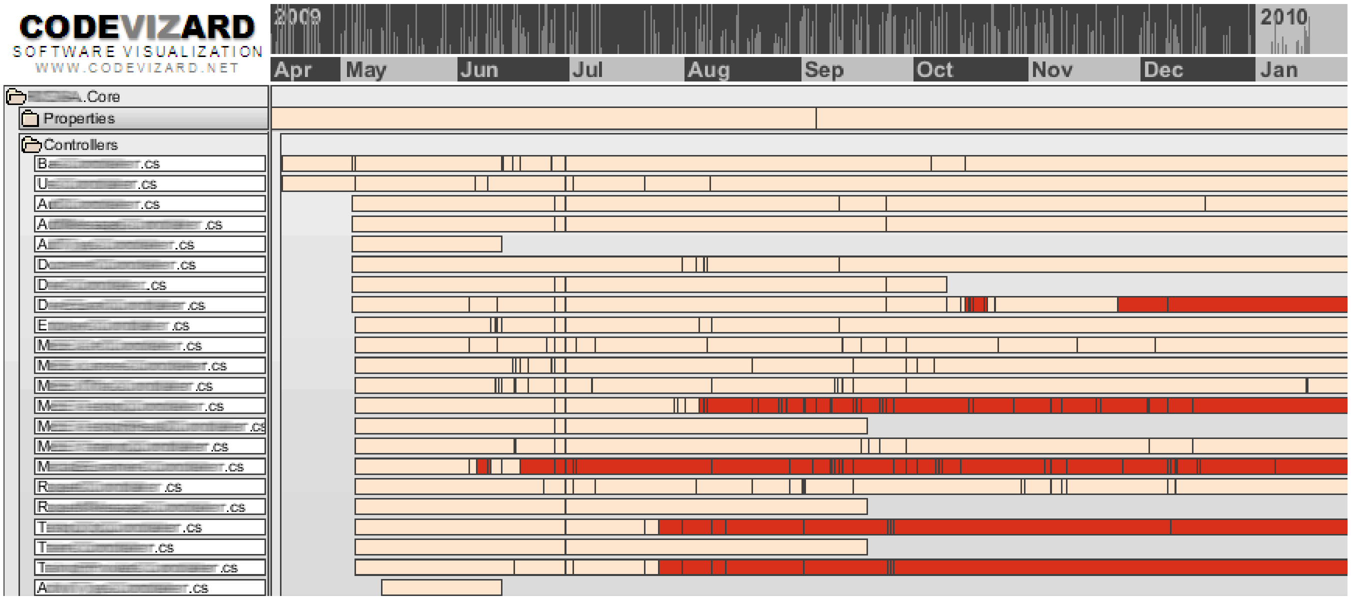

Figure 1: Classes (made anonymous) in project B in the Core.Controllers name space visualized by CodeVizard. The life lines of each class (starting at creation and ending at deletion time) are plotted over time (x-axis). Light coloring indicates that a class is *not* a GodClass, dark red that it is a God Class. Grey bars in the lifeline show when the class was modified.The bars in the time ruler on top show the commit activity by visualizing how many classes were changed per commit (weekly work intervals are visible).

namespaces of the software system. Additionally, this strategy ensured an un-biased selection of classes. This means, large classes were not given preference in order to not automatically exclude small god classes that might have been detected by the human subjects.

4.2.1 *Description of Subjects*

Subject A1 from study A has the role of a programmer and has been programming using object-oriented languages for more than three years. Subject A2 is the technical lead of the project and has been programming for more than 7 years using object-oriented languages. Both subjects had never heard the termscode smell orgod class before. Their current refactoring practice included an ad-hoc process (whenever it becomes necessary) discussed as a team.

Subject B1 from study B is the technical lead of the project and has been programming for more than three years in an object-oriented language. The subject had heard aboutgod classes before. Subject B2 has been programming in object-oriented languages for less than a year. The terms code smells and god classes were new to her. The subjects describe their refactoring approach to be ad-hoc as well as planned. They decide on refactoring using software reviews either done by one person or in a team.

4.2.2 *Study Procedure*

During the study, subjects were first asked to fill out a pre-study questionnaire. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with code smells and god classes and their current refactoring approach. Next, the subjects were introduced to the god classcode smell using a short presentation[2].This was done in such a way that any mention of software metrics was strictly avoided (e.g., concepts such as complexity, cohesion, coupling, etc.). Instead, the introduction focused on the refactoring techniques used to eliminate such a problem. A list of questions was compiled to help the subjects identify god classes:

- Does the class have more than one responsibility?
- Does the class have functionality that would fit better into other classes?
  - By looking at the methods, one could ask: "Is this the class' job?"
- Do you have problems summarizing the class' responsibility in one sentence?
- Would splitting up the class improve the overall design?

For the inspection of classes, a mini-process was designed. This ensures that all subjects perform the inspection of classes in a similar fashion.The process consists of the following steps for each class:

1) Subjects confirm they are familiar with the class. The pilot studies showed, that subjects have problems identifying God Classes whenever they had not seen the code before (when it was written by a different developer for example). One pilot study subject indicated that, "I might have tended to classify it as a God Class when somebody else wrote it, because I could not fully understand all the design decisions."
2) Classes are opened in an editor for examination. In one of the pilot studies a subject did not review each class by looking at the code (because we did not explicitly instruct them to do so). In that case the classification was solely based on memory. This step was added to avoid such a scenario.
3) Classes are reviewed with the questions presented above in mind.
4) Subjects rate the class on the following scale:
   (1) It is not a God Class

[2] The presentation slides can be found in the online lab package.

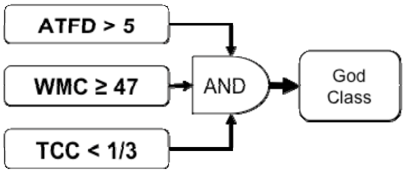

Figure 2: Detection Strategy for God Classes

(2) It might be a God Class
(3) It is a God Class

The scale is based on the findings from the pilot studies where Likert scales with four and five different values were used. It was hard for subjects to decide on values that were not on the extreme ends of the scale.

During the entire process, subjects were encouraged to "think-aloud", thus sharing their thoughts and rationales with the facilitators of the study. This was also recorded as audio. In addition, the subjects were asked if the concept of god classes made sense to them and if they would consider such classes harmful.

4.2.3 Evaluation of Qualitative Data

Coding was used to categorize the comments collected during the think-aloud phase from the subjects. To do this, first the audio recordings were transcribed. For each class that was examined, the subject's comments were collected in a spreadsheet. Next, the researchers reviewed the transcripts and created a number of codes that reflect the issues that were expressed by the subjects during the study using the methodology described in [16]:

- Class is not used anymore
- Class is highly complex
- Class is misplaced
- Class is special (e.g. a necessary framework class)
- Method is wrongly named
- Method is highly complex
- Method is misplaced
- Attribute is not used
- Method/Class lacks comments

In a second step two researchers assigned the codes independently to the transcribed data on a class-by-class basis. For each class, each researcher decided which of the above issues, if any, were mentioned by a subject during thethink-aloud review of that particular class. Inter-rater agreement between the two researchers was calculated using Cohen's Kappa. Kappa is a chance corrected measure of agreement that is used when two raters judge items on a binary scale. For the first round of coding Kappa was calculated to be 85%. According to Landis and Koch [5] this indicates almost perfect agreement between the judges. Those items the researchers could not agree on (a total of 12 out of 1664 ratings) were discussed. This resulted in an agreement of 100%.

4.3 Instrumentation for Evaluation of Automated Classifiers

We used the results of our human subjects as a way to evaluate a metrics based approach to god class detection. We encoded this approach into a software tool (CodeVizard – see Figure 1) developed by our research group.The tool can automatically mine data from source code repositories (i.e. Subversion) and visualize it in various ways. The tool also includes a component that calculates software metrics for Java programs that was used in earlier studies [14].The capabilities of this component were extended to parse C# programs and calculate code metrics for them. The metrics are based on the definitions presented by Lanza andMarinescuin [6]. In addition the tool implements Marinescu's detection strategies [11]for the automatic detection of code smells (including god classes).

Table 2: Example change Likelihood for God Classes and non-God Classes

Revision	1407	1415	1416	1421	1424	Likeli-hood
GC	0/6	1/6	2/6	0/6	0/6	0.100
N-GC	2/218	2/220	9/219	2/219	2/218	0.015

4.3.1 The God Class Detection Strategy

A detection strategy is a logical composition of appropriate code metrics and corresponding thresholds that automatically detects design flaws in an application. Lanza and Marinescudefine them as "the quantifiable expression of a rule by which design fragments that conform to that rule can be detected in the source code"[6].

Marinescudefined god classes exhibiting the following characteristics: (1) high complexity, (2) low cohesion and (3) extensive access to the data of foreign classes. In conclusion he uses the following code metrics to capture theses characteristics: (1) weighted method count (WMC), (2) tight class cohesion (TCC),and (3) access to foreign data (ATFD). Figure 2illustrates the structure of the detection strategy for god classes. More details on the metrics and thresholds can be found in Lanza's andMarinescu'sbook on object oriented metrics [6].

4.4 Effects of God Classes on Maintainability

To evaluate if god classes require an increased maintenance effort and to answer RQ 7, an analysis of the change likelihood for god classes and non-god classes from the two studies was conducted. For this, CodeVizard identifies god classes not only for the latest revision of a class in a repository but also over the complete history (e.g., all revisions) of that class. It is important to take a class' history into account in this context. It would be wrong to conclude that if a class is a god class in the latest revision it has been one for all its existence. Figure 1 (a screenshot from CodeVizard) illustrates the evolution of god classes in project B.

A second issue that needs to be considered for this analysis is the fact that god classes are usually among the larger classes in a software system. This is due to their nature of accumulating large parts of the software's functionality. From a statistical point of view, a change to the software will more likely manifest itself in a large class instead of a small class, assuming that changes are uniformly distributed over all lines of code in the software. To account for this circumstance one should normalize the change likelihood by the lines of code (LOC) of a class. In earlier work by Olbrich et al. [14], it was shown that, without normalizing by LOC, god classes are changed four to five times as often as non-god classes.

The change likelihood is computed as follows. For each revision the number of changed god and non-god classes is determined. Then, both numbers are divided by the total number of god and non-god classes existing in the system. Table 2 shows an extract of the revisions from project A. For example, in revision 1415 one

Table 3: Classification Performance of Subjects

subject	# classes	time taken	classes / hour
Subject A	34	73 min	ca. 28
Subject B	52	85 min	ca. 37
Subject C	53	78 min	ca. 40
Subject D	49	65 min	ca. 45

out of six (16.6%) god classes was changed and two out of 220 (0.9\%) non god classes were changed.

The change likelihood is then calculated as the average of the change ratios over all revisions forgod and non-god classes. In other words, this is the likelihood of one particular god or non-god class to be changed during one revision. The hypothesis is thatgod classes have significant higher change likelihood than non-god classes. Two things need to be taken into consideration: (1) if a revision does not have any changes to source files (e.g. only documentation stored in the repository was changed) then the revision is ignored, and (2) if a system at a revision does not contain any god classes (e.g. in the beginning of the project) then this revision is ignored in the average for the god classes.

Normalization by lines of code is done by dividing the values in Table 2 by the average lines of code of all changed God/non-God Classes for that revision.

5. RESULTS

In the following section the collected and observed results for the research questions stated above are presented.

5.1 Evaluation of Human Performance

R1:*How difficult is it for humans to identify God Classes and how much effortdoes it take?*

All subjects were able to identify classes they felt weregod classes. In the follow-up interview to the study, subjects were directly asked if it was hard for them to detect god classes. Subject A1 from project A said that, due to her familiarity with the project under investigation, it was easy. The same subject said that the classification process became easier after a while. She explained that the mini-process and its accompanying set of questions "help you to organize your mind." Also the fact that she could follow a plan and the repetitive nature of the classification process made the review easier after a while.

Table 4: God Class results for investigated classes from Project A. God Classes are highlighted

Class	Namespace	Subj. A	Subj. B	Mar.
A1, …, A5	Core.Controller	no	no	no
A6	**Core.Controller**	no	**maybe**	**yes**
A7, …, A9	Core.Controller	no	no	no
A10	**Core.Controller**	**maybe**	no	**yes**
A11, ..., A19	Core.Models	no	no	no
A20, ..., A30	Web	no	no	no
A31	**Web**	no	no	**yes**
A32, …, A33	Web	no	no	no
A35	**Web**	no	no	**yes**
A36, …, A52	Web	*skipped*	no	no

Subject B2 from project B said that, based on the introduction, she had no problem understanding what makes a class agod class. The subject also agreed that classification became easier after a while and said that "most of the times when I was unsure, was when it was a class I was unfamiliar with."

Subject B1 of project B also said that detecting god classes did not present a difficult task for him. He said in the post-study interview that, "Going through the classes method by method … there is not really a way somebody could not do it."

Concerning the effort required to review a number of classes for the existence of God Classes we made the findings presented in Table 3.

R2:*How well do humans agree on identifying God Classes?*

Table 4and Table 5show the identified god classes for the two projects. In addition to the subject's classification, the table contains a column that indicates which namespace the class belongs to and a classification based on Marinescu's detection strategy. Classes that were identified as god classes by either the subjects or the detection strategy are highlighted in the tables.

For the analysis and presentation of results, the two cases, "might be a god class" and "is a god class", from the classification scale were combined into "evidence for god class found". This is necessary due to the sparseness of the "is a god class" case.

The first observation that can be drawn from the data is that god classes were rather rare in the set of inspected classes. In project A each of the subjects identified only one class as "maybe a god class" out of 52 inspected classes. Both subjects identified a different class. Using Cohen's Kappa, an agreement of -2% was calculated between the two subjects. Based on Landis's and Koch's[5] interpretation of Kappa, this indicates no agreement between the two judges. In project B, one subject (subject C) identified one class as "is a god class" and the second subject (subject D) identified the same class as "maybe a god class".

Table 5: God Class results for investigated classes from Project B. God Classes are highlighted

Class	Namespace	Subj. C	Subj. D	Mar.
B1, ..., B4	Core.Controllers	no	no	no
B5	**Core.Controllers**	**yes**	**maybe**	**yes**
B6, …, B18	Core.Controllers	no	no	no
B19, …, B35	Core.Models	no	no	no
B36	Core.Properties	no	*skipped*	no
B37, B38	Core.Utils	no	no	no
B39, B40	Web	no	no	no
B41	**Web**	**maybe**	**no**	**yes**
B42	**Web**	**no**	**maybe**	**yes**
B43, …, B45	Web	no	no	no
B46, B47	Web	no	*skipped*	no
B48, …, B51	Web	no	no	no

Table 6: Coding for all classes."X/Y" means that out of Y decisions (review of one class by one subject) X times subjects indicated that the issue in the same column was present.

	Class				Method			Attribute	Method/Class
	not used	high complexity	misplaced	special/ framework	wrong name	high complexity	misplaced	not used	lack of comments
God Classes	0/6	2/6	0/6	0/6	2/6	1/6	6/6	0/6	0/6
Non-God Classes	6/182	4/182	1/182	9/182	1/182	2/182	2/182	1/182	1/182
Total	6/188	6/188	1/188	9/188	3/188	3/188	8/188	1/188	1/188

Further, each of the subjects marked an additional class as “maybe a god class”. As in the first project these two classes did not match. The calculated Kappa is 48% in this case. This number suggests moderate agreement between the two judges. Due to the apparent low agreement between the subjects, it is important to identify the reasoning behind the subjects' classification of god classes.

R3:*What issues in code makes humans identify a God Class as such?*

To answer this question the data obtained through the think-aloud transcripts from the two studies was used. Table 6 shows the combined coding results of both studies. The column headings represent the issues identified through the coding process. In the rows a distinction is made between god classes and non-god classes identified by the subjects and by Marinescu's detection strategy. The numbers in the cells are to be interpreted as follows. The “special/framework” category includes classes that are required by the underlying architecture or frameworks that were used.

For god classes, “X/Y” means out of Y times that someone identified a god class, X subjects indicated the issue in the same column to be present. The same applies for non-god classes. For example, out of six reviews where subjects classified a class as god class, twice they indicated the class had high complexity.

The row with totals shows that misplaced methods and classes that are not used anymore are among the most prominent issues identified by the subjects.

Those classes that were identified as god classes by the subjects are only linked to a limited number of issues. These are methods that are misplaced, have the wrong name, or exhibit a high level of complexity. Among these issues, misplaced method is the most prominent one. Every time a subject identified agod class, the subject also indicated that the class contained a misplaced method. For the remaining 182 times, the subjects classified a class as a non-god class, they only determined in two cases that the class contained a misplaced method.

In addition, the agreement on the identified issues between the subjects for the two studies was calculated using Cohen's Kappa. Kappa for project A is 37% and 39% for project B. This indicates ”fair agreement” between the subjects [5]. The agreement for individual issues was calculated as well. For misplaced methods, the agreement is 46% for project A and 47% for project B. This results in ``moderate agreement'' between the subjects for the misplaced method issue.

5.2 Evaluation of Automated Classifiers

R4:*How well do the previously proposed metric based classifiers do compared to human classification?*

Table 4 and Table 5 show that all classes that were identified as god classes by the human subjects, were also detected by Marinescu's detection strategy. Assuming the classification done by the human subjects to be the ground-truth, this results in a recall of 100% for the automatic detection (all god classes were found). The precision of the detection strategy for god classes is also successful - Only two additional classes were detected that were not identified by the humans. This results in a precision of71%.

R5:*How are the identified code issues related to the metric based classification approach?*

In section 4.3.1 it was shown how Marinescu's detection strategy uses metrics to detect high complexity, low cohesion and extensive access to the data of foreign classes to automatically determine whether or not a class is a god class. Only ”methods that do not belong into a class” could be identified as a strong driver that let humans classify a class as a god class. This finding will be further investigated in the discussion section for this research question.

R6:*Is it possible to improve existing metric based approaches based on the previous findings?*

Marinescu's detection strategy detects two additional classes as god classes that were not identified by the subjects as such. An improvement in the context of the two studies would be if the detection strategy would not detect those extra two classes.

When comparing the values for ATFD, TCC and WMC of these two classes to those of the other god classes the values for ATFD are relatively low (8, 9) and close to the Detection Strategy’s minimum threshold for ATFD of 5. For allother god classes the values for ATFD are higher (14, 55, 29, 38, 28). Increasing the god class detection strategy's threshold for ATFD to 10 would result in a precision of 100% for automatic detection.

5.3 Effects of God Classes on Maintainability

R7:*Do God Classes require a higher maintainability effort than non-God Classes?*

Table 7 and Table 8 show the results of the change likelihood analysis, once without and once with normalizing by lines of code. In Table 7 one can see that in project A there were 421 changes to the software where god classes were present at that time. The likelihood of a god class being part of a change was about 9%. In other words, if a class is a god class then it is changed in almost every 10th revision.

On the contrary, the chance of a non-god class being part of a change is only 1.7%. For project B the likelihood is about 19% for god classes and 2% for non-god classes. To test the results for statistical significance a Shapiro-Francia test for normality [17] was first performed ($p<0.001$) on all datasets. Afterwards a two tailed, two sample t-test was applied similar to Olbrich's approach in [14]. Both results are significant when tested with the t-test ($p<0.05$). The p-values are given in the last row of the table.

In Table 8 the same numbers are presented after normalizing by lines of code. One can see that the difference between the likelihood for project A is small (0.029% for god classes vs. 0.022% for non-god classes}. For project B the likelihood of a line of code being changed in a god class is almost twice as high as in a non-god class (0.042% vs. 0.027%). However, both results are not significant when tested with a two sample t-test ($p<0.05$). Therefore, the hypothesis that the likelihood of a change of a line of code in agod class is significantly higher than in a non-god class has to be rejected.

6. DISCUSSION

In this section we present answers to our research questions and compare our findings to the related work mentioned earlier.

6.1 Evaluation of Human Performance

R1:*How difficult is it for humans to identify God Classes and how much effortdoes it take?*

The results show that the subjects in the two studies did not find the task of identifying god classes to be challenging. We believe that this is in large part due to the way subjects were introduced to god classes. The introduction, itself, and the identification process were purposely designed in a way related to the decisions developers would typically make on their code. For example, the subjects were not told how god classes violate the paradigm of high cohesion and loose coupling. Instead, the set of questions that accompanies the process was designed to relate to their refactoring activities.

One subject told us after the study that he really liked the problem-oriented nature of the experiment that focused on a certain flaw. He said, "I have never done an exercise like this, where I looked at the [B5] classwith god classes in mind I could see, how this could definitely be broken up. And I see the benefit of splitting that up into multiple classes. I have never really thought about that before, but it is easy to see, once you look at it in that way."

Another subject told us that he could definitely see how god classes are harmful. He considered our introduction and the focused review approach to be very helpful. He said, “It was good. I think it was a good learning experience.”

R2:*How well do humans agree on identifying God Classes?*

The results show that the agreement on the identification of god classes was low among the subjects in the studies. The low agreement may be due to differing perception of code issues by different developers. This will be investigated in the discussion of research question RQ3.

Mäntylä et al. [8] state similar results. They report high agreement among the subjects for simple code smells like Long Method and Long Parameter List. For more complicated ones like Feature Envy they observed low agreement among the judges. When comparing god classes to these three code smells they clearly fall into the category with the more complicated ones. Unlike long method and long parameter list, god classes cannot be identified by simply counting certain size characteristics.

R3:*What issues in code makes humans identify a God Class as such?*

Thesubjects linked the following issues to god classes they identified: misplaced methods, methods with the wrong name, methods with high complexity, and classes with high complexity. Misplaced methods can clearly be identified as the strongest driver for letting the subjects classify a class as a god class.

Based on this observation it is worthwhile to investigate what effects a misplaced method can have on a class. If a method is misplaced, there has to be another class that would serve as a better container for that method. The most obvious reason for why a method would better fit into another class is when the method actually processes data from foreign classes rather than the class that contains it. The transcripts from the study show examples where subjects identified such behavior:

"It would make more sense for me to define [this method] in the ABC class so I am not sure why it is in here. ... I think it would make more sense for that to go in with the rest of the retrieving ABC information." Another subject identified the exact same issue: "In here we have retrievedABC, now this actually should not be here, this should be in the ABC"

The combination of these observations indicates that the presence of methods in a class that do not belong there are an indicator for a lack of cohesion.Mäntylä et al.report in [7]poor inter-rater agreement for the refactoring drivers that were identified by their subjects. Based on[5] we determined "fair agreement" between the subjects. These results might not be directly comparable, since Mäntylä does not give details on how the agreement was calculated.

Table 7: Change likelihood for God and non-God Classes in both projects. N represents the number of changes

	Project A		Project B	
	God Classes	Non-God Classes	God Classes	Non-God Classes
N	421	534	121	167
mean	0.090	0.017	0.197	0.020
s	0.201	0.041	0.333	0.023
	p-value: 1.1E-15		p-value: 5.0E-11	

Table 8: LOC Normalized change likelihood for God and non-God Classes for both projects N is the # of changes

	Project A		Project B	
	God Classes	Non-God Classes	God Classes	Non-God Classes
N	421	534	121	167
mean	0.00029	0.00022	0.00042	0.00027
s	0.00090	0.00068	0.00074	0.00064
	p-value: 0.192		p-value: 0.060	

6.2 Evaluation of Automated Classifier

R4:*How well do the previously proposed metric based classifiers do compared to human classification?*

The results show that Marinescu's detection strategy is effective at detecting god classes in this case. Using the classification of the subjects as a reference, recall was 100% and precision 71%. Related work from Marinescu [11] (70% precision), Moha[12], (40% precision, 100% recall) and Kreimer[4], (90% precision) show similar results. A reason for Moha's relatively low precision might be the fact that her's is actually the only publication, except for this study, that also reports recall. The other two studies may have easily missed some of the code smells that would have been identified by humans.

Table 4 shows the two classes (A31, A35) that were identified by the detection strategy but not by the subjects both reside in the Web namespace of the application. These classes contain the logic behind the web pages in the application. Each class is directly associated with one page of the web interface. Usually the developers placed all the logic that is required by a web page in its accompanying class.

As a consequence, several of the classes in the Web namespace have multiple responsibilities and contain a collection of functionality required by the web page. However, a subject explained that all functionality for a given web page must go in the same class. He identified this as a disadvantage of the underlying architecture used by both applications examined in the studies. In conclusion, he suggests that the applications "would definitely benefit from having another layer to go in between these models and the controllers that handles your business objects and does preprocessing before you display data on the pages." This example shows how the underlying architecture influences the design in a negative way.

R5:*How are the identified code issues related to the metric based classification approach?*

The results show how a lack of cohesion and complexity are identified as issues that let the human subjects identify a class as a god class. This corresponds with the use of the weighted method count metric (WMC) and tight class cohesion metric (TCC) used by Marinescu to express complexity and cohesion in his detection strategy for god classes.

In addition to these two metrics, the detection strategy also uses the access to foreign data metric (ATFD) to express the classes' degree of coupling to other classes in terms of data usage. Nevertheless, no issues could be identified during the coding process that can be associated with this metric or coupling directly. This might be due to the fact that coupling is difficult to evaluate in a software review that inspects classes in a file-by-file fashion. Without a diagram (e.g., a UML class diagram) that reveals the connections between the components of the software, the degree of coupling might be hard to assess by human reviewers.

R6:*Is it possible to improve existing metric based approaches?*

The results show how increasing the detection strategy's threshold for the metric access to foreign data (ATFD) to 10 results in a precision of 100% for the automatic detection of god classes. Nevertheless, one has to keep in mind that such an adjustment may result in over-fitting the model to these particular software systems.

Another finding in this context is the fact that the detection strategy could be reduced to an evaluation of the class' complexity (WMC) only. As a reaction to the previous findings ATFD was removed from the detection strategy. For the remaining metrics the thresholds were kept the same. For both software projects, the same god classes were detected. The detection strategy was further simplified by leaving out the TCC metric. It was still possible to detect the exact same god classes as before by only marking the classes with a WMC higher than 47.

6.3 Effects of God Classes on Maintainability

R7:*Are God Classes correlated with higher maintainability effort?*

God Classes are indeed, in both systems, changed five to ten times more often than their counterparts. However, after taking the increased size of a God Class into account no significant difference can be found.

7. THREATS TO VALIDITY

The most obvious threats to external validity are the sample size and scope. The experiment was only run in one company and with two projects. For each project, only two developers participated (both projects were 2-person projects). Therefore, it cannot be concluded that the results will hold in general. Additionally, only six god classes were found using automatic detection and human judgment combined. Consequently, this may affect the validity of the conclusions drawn.

As a threat to internal validity, the instructional material and the god class classification process might contain potential for biasing the subjects. However, this thread was minimized by avoiding the mention of metrics, focusing on concrete refactoring techniques and using previously presented definitions for god classes.

8. CONCLUSION

The purpose of this thesis was to find empirical support for the detection of code smells. To do this the approach was divided into three phases.

First, the human performance of detecting god classes was investigated. The findings show that the task of finding god classes in a software system was perceived as not challenging by the subjects. We conclude that, if provided with a suitable process, humans can detect code smells in an effective fashion. Nevertheless, the calculation of Cohen's Kappa for the classification by the subjects shows that agreement was low. Other studies report similar results. The fact that a class contains a method that would fit better into another class was identified as a strong driver for whether or not the subjects identified the class as a god class. Misplaced methods are interpreted as a lack of cohesion.

Secondly, the performance of automated classifiers was compared to the human results. Marinescu's detection strategy for god classes [11] was run on the same classes that were reviewed by the subjects. With a recall of 100% and a precision of 71%, the detection strategy proved effective in this context. Complexity and cohesion related issues identified by the subjects could be related to metrics used in Marinescu's detection strategy. Adjusting the thresholds of metrics used in the detection strategy allowed to increase its precision to 100%.

Thirdly, an investigation of the effects of god classes on maintainability effort did not show in all cases that they require proportional more maintenance effort.

By conducting our studies in a professional environment, conclusions can be drawn for real-life software development projects. Based on the results and observations, a god class centric review process can be recommended. The results and discussion on research question RQ1 show how such an issue-focused perspective helps the developers conduct a software review in an effective and goal-oriented fashion.

Furthermore, the results for the automatic detection of god classes show how Marinescu's detection strategies recall successfully for the two projects. A computer assisted strategy in which automatically detected god classes undergo a second human review can reduce the required effort. The results of the study increase the overall confidence in the results of automatic code smell detection.

9. EXPERIMENTAL REPLICATION AND FUTURE WORK

To facilitate the replication of the study we have created a lab package available online at www.codevizard.net. The package includes the questionnaires, introductory material, questions from the post-study interview,and protocols for the presented studies. From our point of view, the following variations of the study would be most interesting to conduct: (a) a variation of the application type inspected (not web) and (b) a variation of the kind(s) of Code Smell(s) under investigation.

10. ACKNOWLEDGEMENTS

We would like to thank Kathleen Mullen, Rick Flagg, and our subjectsfor their valuable feedback.

This research was supported by NSF grant CCF 0916699, "Measuring and Monitoring Technical Debt " to the University of Maryland, Baltimore County.

11. REFERENCES

[1] Cunningham, W. 1993. The WyCash portfolio management system. *SIGPLAN OOPS Mess.* 4, 2 (Apr. 1993), 29-30.

[2] E. Van Emden, L. Moonen 2002, Java Quality Assurance by Detecting Code Smells. Ninth Working Conference on Reverse Engineering (2002).

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

[4] Kreimer, J. 2005. Adaptive Detection of Design Flaws, Electronic Notes in Theoretical Computer Science. Volume 141, Issue 4, Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (12 December 2005), 117-136.

[5] Landis, J. R., Koch, G. G. 1977. The measurement of observer agreement for categorical data. Biometrics , 33, 159-74.

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

[7] Mäntylä, M. V. and Lassenius, C. 2006. Drivers for software refactoring decisions. In *Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering* (Rio de Janeiro, Brazil, September 21 - 22, 2006). ISESE '06. ACM, New York, NY, 297-306.

[8] Mäntylä, M. (2005). An experiment on subjective evolvability evaluation of object-oriented software: Explaining factors and interrater agreement. In: Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE 2005). Noosa Heads, Queensland, Australia. 17-18 November 2005, 10 pages.

[9] Mäntylä, M., Vanhanen, J., Lassenius, C. (2004, Sep). Bad smells - humans as code critics. Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on , 399-408.

[10] Mäntylä, M., Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Engineering, Volume 11, Issue 3, 395-431.

[11] 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.

[12] Moha, N., Gueheneuc, Y.-G., Duchien, L., Meur, A.-F. L. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. Software Engineering, IEEE Transactions on , 36, 20 -- 36.

[13] Munro, M. J. 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of the 11th IEEE international Software Metrics Symposium* (September 19 - 22, 2005). METRICS. IEEE Computer Society, Washington, DC, 15.

[14] 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.

[15] Parnin, C., Görg, C., and Nnadi, O. 2008. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM Symposium on Software Visualization* (Ammersee, Germany, September 16 - 17, 2008). SoftVis '08. ACM, New York, NY, 77-86.

[16] Seaman, C., "Qualitative Methods," in Shull, F., Singer, J., and Sjøberg, D. I. K., (eds.) Guide to Advanced Empirical Software Engineering, London: Springer, pp. 35-62, 2007.

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