The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems

Steffen Olbrich
Dept. of Computer Sciences, University of Applied Sciences, Mannheim, Germany
solbrich@gmail.com

Daniela S. Cruzes
IDI, Norwegian University of Science and Technology (NTNU), Trondheim, Norway
dcruzes@idi.ntnu.no

Victor Basili
University of Maryland and Fraunhofer Center Maryland, College Park, USA
basili@cs.umd.edu

Nico Zazworka
Dept. of Computer Science, University of Maryland, College Park, USA
nico@cs.umd.edu

Abstract

Code smells are design flaws in object-oriented designs that may lead to maintainability issues in the further evolution of the software system. This study focuses on the evolution of code smells within a system and their impact on the change behavior (change frequency and size). The study investigates two code smells, God Class and Shotgun Surgery, by analyzing the historical data over several years of development of two large scale open source systems. The detection of code smells in the evolution of those systems was performed by the application of an automated approach using detection strategies. The results show that we can identify different phases in the evolution of code smells during the system development and that code smell infected components exhibit a different change behavior. This information is useful for the identification of risk areas within a software system that need refactoring to assure a future positive evolution.

1. Introduction

Good object-oriented software design features a set of non-functional quality characteristics, e.g., maintainability, understandability, ease of evolution, etc. In order to assure these non-functional requirements, object oriented software systems should follow a common set of design principles such as data abstraction, encapsulation, and modularity [7] [19] [20]. However, even when developers are familiar with those techniques, deadline pressure, too much focus on pure functionality, or just inexperience may lead to violations of these design rules.

One possible way of identifying such design flaws in object oriented designs is the detection of 'code smells' [6] [18]. The term was coined by Fowler and Beck [6] by presenting an informal definition of 22 code smells that provide a set of characteristics used as indicators for design flaws with respect to the maintainability of software systems. Each code smell examines a specific kind of system element (e.g. classes or methods), that can be evaluated by its inner and external characteristics.

A manual detection of code smells by code inspections [6], leads to different issues which are identified by Marinescu [13] as: time-expensive, non-repeatable and non-scalable. Even more issues concerning the manual detection of design flaws were identified by Mäntylä [11] [12]. He showed that as the experience a developer has with a certain software system increases, his ability to perform an objective evaluation of the system as well as his ability to detect design flaws decreases.

Marinescu created a generic approach for identifying code smells in software systems [13], to avoid those issues. Instead of a purely manual detection approach, he used code metrics for the detection of risk areas in the system. This approach was later refined in [14], [15] and [16] with the introduction of metric based detection strategies.

Li and Shatnawi [9] investigated the relationship between the class error probability and bad smells based on three versions of the Eclipse[1] project. Their result showed that classes which are infected with the code smells Shotgun Surgery, God Class or God Methods have a higher class error probability than non-infected classes. Deligiannis et al. investigated in the impact of God Classes, based on Riel's definition [18], on the maintainability of object oriented design [3] [4]. In an experiment, he showed that existing design

[1] Eclipse <http://www.eclipse.org>

violations in software systems lead to an increased probability that later changes, e.g. maintenance tasks, causes further design violations.

These results show that it is important to evaluate how software evolves with respect to components with code smells. The question arises: How do components with code smells effect maintainability. Hidden dependencies of structurally unrelated, logically coupled system elements exhibit a high potential to negatively affect software evolution by exhibiting architectural deterioration over time. One question that arises is: Am I maintaining the quality of my software as I change or add functionality? Whereas related work in [3], [4] and [9] analyzed the impact and occurrence of code smells by single, static versions of software systems, this paper addresses questions related to the evolution of the code: How do code smells evolve over time and how do code smells influence the change behavior of the infected system elements? One hypothesis is that the bad smells increases over time, because as the software changes are performed, the software degrades. Another hypothesis is that the classes infected by the code smells suffer more and larger changes than the ones that are non-infected. Within the scope of this study we analyzed the historical data of two major Open-Source-Projects, Apache Lucene[2] and Apache Xerces 2 J[3]. The study is focused on two well-known code smells: God Class, which "refers to those classes that tend to centralize the intelligence of the system." [14] and Shotgun Surgery, a code smell which indicates excessive, incoming low-level coupling.

The paper is structured as follows: first we introduce code smells and discuss how they can be detected, based on the detection strategies defined by Marinescu. Then we introduce the research questions we want to investigate in our study, followed by the experiment. Finally we will describe the results and discuss them.

2. Detection Strategies

Code metrics are quantification mechanisms that support the examination of system element characteristics. They can provide insight into whether the observed system artifact might contain a design flaw, but don't provide an identification of the specific flaw. In other words, a metric result can be interpreted as a certain symptom of one or more problems but doesn't show the actual problem.

In an attempt to fill this gap, Marinescu [14] defined what he called a "detection strategy" which combines different code metrics, filters the result and combines this information in order to detect problems in the code architecture. His formal definition is: "the quantifiable expression of a rule by which design fragments that conform to that rule can be detected in the source code" [16]. Thus, we can use code metrics on a more abstract level. The engineer is not forced to post-process and interpret a big and possibly confusing set of metric results by himself. Instead detection strategies are used as an automated mechanism for the interpretation of metric results. For this reason detection strategies are much closer to our purpose for using code metrics: the detection of design flaws. Even if it is reasonable that the result of a detection strategy is re-evaluated by an engineer to verify the flaw, it provides an overview of possible risk areas in the system in a quick and repeatable way.

Each Detection Strategy is structured in three consecutive elements:

1) A set of code metrics.
2) A set of filtering rules, one rule for the interpretation of each metric result (i.e., comparison with an adequate threshold value).
3) The composition of the filtered results (Logical composition of the results).

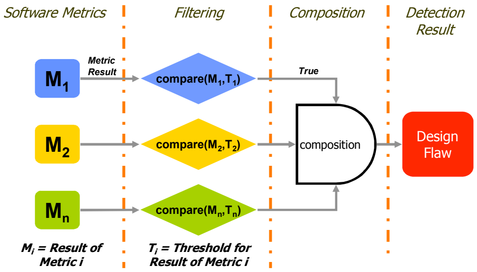

Figure 1. Schematic structure of a Detection Strategy [16]

When applied to a software system, the result of a detection strategy (*DS*) for an entity (*E*), i.e., a system element like a class, is 1 or 0 depending on whether the rule of the detection strategy is satisfied or not (i.e., the entity possesses the design flaw addressed by the detection strategy). Figure 1 shows the generalized structure of a Detection Strategy related to Equation 1, where *M* denotes Metric and *T* Threshold.

$$DS(E) = \begin{cases} 1, & ((M_1\ comp_1\ T_1)\ conc_1\ (M_2\ comp_2\ T_2)\ conc_2 \ldots conc_{n-1}(M_n\ comp_n\ T_n)) \\ 0, & else \end{cases}$$

$$comp_{1..n} \in \{<,>,=,\leq,\geq\}$$

$$conc_{1..n} \in \{\wedge,\vee\}$$

Equation 1

2.1. God Class Detection Strategy

A class is identified as a God Class if it performs too much work on its own, delegating only minor

[2] Apache Lucene <http://lucene.apache.org>
[3] Apache Xerces 2 J <http://xerces.apache.org>

details to a set of trivial classes and using the data from other classes [16].A God Class violates the object-oriented design principle that the intelligence of a system should be uniformly distributed among the top-level classes [18].

A God Class instead represents "an aggregation of different abstractions and (mis)use other classes (often mere data holders) to perform its functionality." [16]. This causes a negative effect concerning the understandability and the evolution (i.e. the reusability and maintainability) of the system [3] [4]. The understandability issue is even intensified since God Classes tend to be large classes. Furthermore a higher error-probability can be observed on God Classes [9], which might be caused by the complexity of maintenance tasks on God Classes. Nevertheless, classes which show the structural characteristics of a God Class but are rather untouched and reside in a stable part of the system do not cause problems [16] [17].

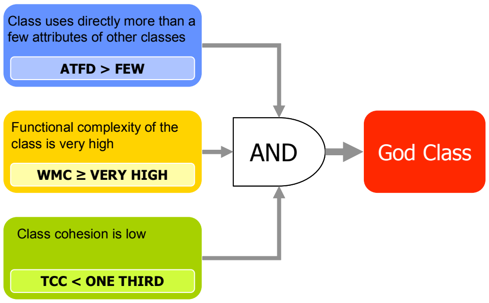
Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
God Class

Figure 2. Schematic structure of the Detection Strategy for the detection of the God Class code smell [16]

A God Class (*GC*) code smell features (1) a high complexity, (2) a low, inner-class cohesion and (3) a extensive access to the data of foreign classes [16]. Below we introduce the set of code metrics used to express those characteristics quantitatively:

1) Weighted Method Count (*WMC*) is the sum of the static complexity of all methods in a class [2]. McCabe's cyclomatic complexity is used as complexity measure for methods [10].
2) Tight Class Cohesion (*TCC*) is the relative number of directly connected public methods in the class. Two visible methods are directly connected, if they are accessing the same instance variables of the class [1].
3) Access to Foreign Data (*ATFD*) represents the number of attributes of foreign classes accessed directly or by using accessor methods [14].

Figure 2 shows the structure of the detection strategy, its formal representation is shown in Equation 2. The threshold values for the filtering rules are based on the work of Marinescu [16].

$$GC(E) = \begin{cases} 1, & ((WMC \geq 47) \wedge (TCC < \frac{1}{3}) \wedge (ATFD > 5)) \\ 0, & else \end{cases}$$

Equation 2

2.2 Shotgun Surgery Detection Strategy

The Shotgun Surgery bad smell describes classes in which even a small semantic change may cause a cascade of different changes in many coupled classes [6]. The smell is an indicator of excessive low-level couplings, in the sense of strong afferent (incoming) coupling. It takes into account not only the coupling strength, based on the distinct number of calling operations, but also the dispersion over the system [16].

Operations in classes infected by Shotgun Surgery have many design entities that depend on it. If a semantic change is performed on such operations, the developer is forced to make correlative modifications on each coupled entity. Because of the system-wide dispersion and the high amount of coupled entities there is a high risk of missing a required change, which causes maintenance problems [6] [16]. Similar to the God Class, Shotgun Surgery is also positively associated with the class error-probability [9].

Since the Shotgun Surgery (*ShS*) code smell is caused by afferent coupling instead of the structural characteristics of the observed system element (i.e., a class), an examination of its environment is necessary. Thus (1) the number of methods which call a method of the class has to be determined as well as (2) the number of classes which contain the calling methods. The set of code metrics used to measure those factors are:

1) Changing Methods (*CM*): number of distinct methods that call a method of the class [14].
2) Changing Classes (*CC*): number of classes in which the methods that call the measured method are defined [14].

Marinescu defined the Shotgun Surgery Detection Strategy on methods [16] instead of classes. Since we want to work with it on the class level, we defined that if at least one method of the class is infected by the smell then the class itself is infected. Figure 3 shows the structure of the detection strategy, its formal representation is shown in Equation 3. The threshold values for the filtering rules are based on Marinescu's work [16].

$$ShS(E) = \begin{cases} 1, & ((CM > 10) \wedge (CC > 5)) \\ 0, & else \end{cases}$$

Equation 3

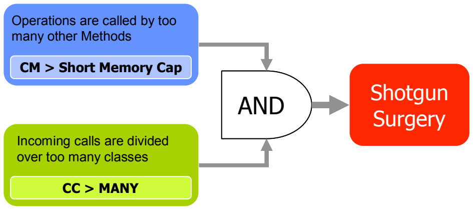
Operations are called by too many other Methods
CM > Short Memory Cap
Incoming calls are divided over too many classes
CC > MANY
AND
Shotgun Surgery

Figure 4. Schematic structure of the Detection Strategy for the detection of the Shotgun Surgery code smell [16]

3. Research Questions

3.1. Research Question I

The first research question addresses the evolution of code smells in a software project. We want to know how the number of code smells changes over a long period of development time. The work of [3] [4] [6] shows that if no countermeasures are taken software quality decays over time. Our hypotheses are therefore:

H1: The total number of code smells increases steadily.

H2: The relative number of components having code smells increases over time.

3.2. Research Question II

The second question of interest aims at investigating if we can find evidence that code smells effect component development in terms of frequency and size of changes. Therefore, we want to investigate if software components associated with smells show a different change frequency than components without smells and if changes to classes with smells force successively larger changes on the whole system. Our theory is that classes, especially the ones with a God class smell, have to be maintained more frequently since code smells lead to a decreased measure of maintainability. Our two hypotheses are:

H3: The change proneness of components with smells is higher than the ones without.

H4: The code churn of changes in infected classes is significantly larger than the code churn on non-infected classes.

4. Experimental Setup

4.1. Instrumentation

The research questions stated above require an analysis of the evolution of software systems. Therefore, a Framework for the computation of the code metrics and the application of the detection strategies was developed as part of a larger tool framework, developed by our research group (CodeVizard[4]) see Figure 4. CodeVizard provides functionality to gather and mine data from source code repositories (i.e. Subversion[5]) and hence the past evolution of the stored systems. The tool also offers various visualizations for examining the infected classes and their change history. CodeVizard was designed for analysis on source file level. Thus, our analysis has the restriction, that the information of multiple classes per file is not stored in the data system, so only the name giving (i.e., *public*) class in each source file is investigated.

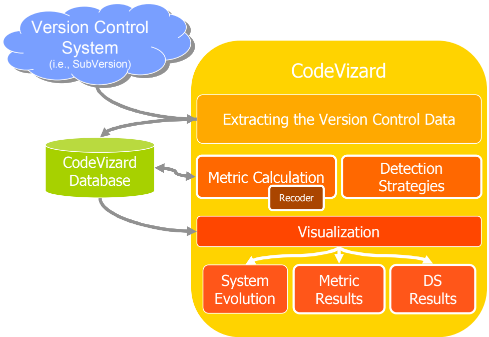

Figure 3. Schematic representation of CodeVizard

The metric calculation module uses the Recoder API, “a Java framework for source code meta-programming”[6]. The Recoder API allows the extraction of a source code meta-model including cross reference information of the Java source files. The extraction is dependent on a closed world assumption, i.e., all used system elements (i.e., classes, libraries, etc.) have to be known, and that the source code is syntactically correct. In other words, the system has to be compilable. Since each revision represents a snapshot of a system in development, this is not always assured.

4.2 Working Products

The applications selected for this case study had to meet different criteria, like the implementation language, system size and quality aspects. The different criteria can are shown in Table 1.

We defined mature open source projects as projects which are still active and provide a present development time of more than three years. We defined well-established as projects with a broad user community (based on the users and authors) and well known in the open source community. For the

[4] CodeVizard <http://hpcs.cs.umd.edu/index.php?id=21>

[5] Subversion <http://subversion.tigris.org >

[6] Recoder <http://recoder.sourceforge.net>

evaluation of those rather difficult to measure aspects we used the open source network Ohloh[7].

Table 1 shows an overview of our selection criteria. Based on the criteria, we selected two open source projects: Apache Lucene and Apache Xerces 2 Java, both developed under the leadership of the Apache Software Foundation.

Table 1. Selection criteria

Implementation Language:	Java
Version Control System:	SubVersion
Historical data (# of revisions):	> 1000 revisions
Project size (# of classes):	> 350 classes
Q_1 – Mature (development time):	> 3 years
Q_2 – Well-established:	(1) broad user community (2) well-known in OS community

Apache Lucene is "a high-performance, full-featured text search engine library written entirely in Java". Apache Lucene, referred to as Lucene, contains 383 Java files in its latest version and the historical data contains 1408 versions.

Table 2. Application fact-sheet

	Lucene	**Xerces**
Project website:	lucene.apache.org	xerces.apache.org
Project size (latest version):	383 source files	618 source files
Historical data:	1400 revisions	3250 revisions
Q_1 – Mature (dev. time):	6 years	6 years

Apache Xerces 2 J is a "library for parsing, validating and manipulating XML documents". The project size of Apache Xerces 2 J, referred to as Xerces, is 687 Java files and 3902 versions of historical data are available. An overview of both applications is shown in Table 2**Error! Reference source not found.**[8].

4.3 Data Collection

With the high number of revisions for each of the two software applications it became necessary to define a strategy to select a reasonable but still significant revision sample of the two applications. We decided to use the aggregation of changes in chunks of 50 revisions as shown in Figure 5. Based on the given constraints (See also section 4.1), 25 revisions of Lucene and 51 revisions of Xerces were analyzed.

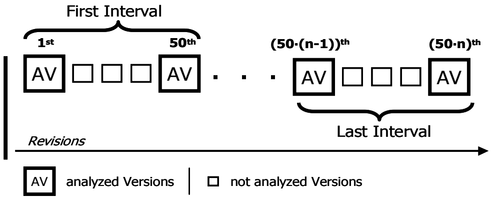

Figure 5. Analyzed Intervals

4.3.1 Data Collection for Research Question I. For the analysis of the evolution of code smells (research question 1), we compare the number of entities (*E*) (i.e., classes) which contain a code smell (*F*) with the total number of entities in the system (|*S*|, whereas *S* is representing the system as set of entities *E*) at time *t* (i.e., the respective revision).

Equation 4 shows if an analyzed entity features a code smell at time *t* (the respective result of the applied detection strategy). Equation 5 represents the calculation of the number of flawed entities (*FE*) at time *t* based on the detected, flawed classes (*Suspect*$_t$) defined in

Equation 4. Equation 6 represents the calculation of the total number of entities (*TE*) at time *t*.

$$Suspect_t(E,F) = \begin{cases} 1, & E_t \textit{ features smell } F \; (GC(E_t)/ShS(E_t)) \\ 0, & else \end{cases}$$

Equation 4

$$FE_t(S,F) = \sum_{\forall E \in S_t} Suspect_t(E,F)$$

Equation 5

$$TE_t(S) = |S_t|$$

Equation 6

4.3 Data Collection for Research Question II. The monitored data for the investigation of the change frequency and change size of each revision contains the following information:

1) A unique identifier for each entity (*E*).
2) The detected status of the entity (Equation 4) at time t_1.
3) Transition type between the entity status at revision t_0 and revision t_1, e.g., code smell class to non-code smell class.
4) The number of changes performed on the entity between revision t_0 and revision t_1 (*ChangeFrequency(E,t_0,t_1)*; Equation 7).
5) The average *code churn* (i.e., number of lines modified, added and deleted) based on each change between time t_0 and time t_1 (*AverageChurn(E,t_0,t_1)*; Equation 8).

[7] Ohloh, the open source network <http://www.ohloh.net>

[8] Based on the used historical data (Only the historical data from 1999-2005 is used for the analysis of Xerces.).

$$ChangeFrequency(E,t_0,t_1) = \sum_{t=t_0}^{t_1} change(E_t)$$

Equation 7

$$AverageChurn(E,t_0,t_1) = \frac{\sum_{t=t_0}^{t_1} churn(E_t)}{ChangeFrequency(E,t_0,t_1)}$$

Equation 8

If there is a change in the status within revision t_0 to revision t_1, it would not be possible to identify whether the changes or churns are associated with the code smell or not. Thus we eliminated those transitions from our study. We identified two stable transitions: a normal class (i.e., non-infected class - *NC*) at time t_0 to normal class at t_1 denoted as *NCNC* (Equation 9) and a smell class (i.e., infected class - *SC*) at time t_0 to smell class at t_1 denoted as *SCSC* (Equation 10). If we include classes appearing and disappearing, i.e., the transitions to and from *NC* or *SC* to non-existing class (*NA*) then we have eight possible transitions as shown in Figure 6.

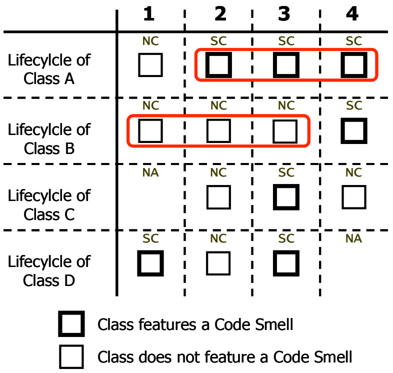

Figure 6. Only the stable transition sets of Class A and Class B are used for the analysis

$$NCNC(E,F,t_0,t_1) = (Suspect_{t_0}(E,F) == Suspect_{t_1}(E,F)) \wedge \; Suspect_{t_1}(E,F)$$

Equation 9

$$SCSC(E,F,t_0,t_1) = (Suspect_{t_0}(E,F) == Suspect_{t_1}(E,F)) \wedge Suspect_{t_1}(E,F)$$

Equation 10

5. Results

5.1 Results of Research Question I

The analysis of the evolution of the code smells in the system is based on a comparison of the set of classes which are infected with a certain smell *F* at time *t* relative to the set of all classes in the system at time *t* (*RFE$_t$(S,F)*, the number of infected classes, *FE$_t$*, divided by the number of all classes, *TE$_t$*, in the system).

We investigated the evolution of each analyzed revision compared to the previous one with respect to patterns in the behavior of the relative number of code smells. Patterns were identified as increasing (*RCSI$_t$(S,F)*, Equation 11), decreasing (*RCSD$_t$(S,F)*, Equation 12) or stable (*RCSS$_t$(S,F)*, Equation 13). In order to avoid a too fine a grained result which might lead to an overfitting of the analysis, we defined thresholds for the indication of significant increases or decreases in the relative number of code smells (T_{CS}). The thresholds are *0.5%* for the God Class and 0.28% for Shotgun Surgery smell were used. These are generic values, based on the mean of *1/10* of the distance between the observed maximum and minimum value of the relative number of a certain smell over both projects.

$$RCSI_t(S,F) = \begin{cases} 1, & RFE_t(S,F) > RFE_{t-1}(S,F) + T_{CS} \\ 0, & else \end{cases}$$

$$(t \geq 1)$$

Equation 11

$$RCSD_t(S,F) = \begin{cases} 1, & RFE_t(S,F) < RFE_{t-1}(S,F) - T_{CS} \\ 0, & else \end{cases}$$

$$(t \geq 1)$$

Equation 12

$$RCSS_t(S,F) = \begin{cases} 1, & RFE_{t-1}(S,F) - T_{CS} < RFE_t(S,F) < RFE_{t-1}(S,F) + T_{CS} \\ 0, & else \end{cases}$$

$$(t \geq 1)$$

Equation 13

Observing the evolution behavior of the code smell density allows us an identification and interpretation of three different evolution phases, listed below:

Positive evolution:

RCSD$_t$(S,F) = 1; the relative number of classes infected with code smells is decreasing significantly.

Decreases in the relative number of code smells may imply that code smells were reduced on purpose (i.e., by refactoring) or that newly added classes are mostly *non*-infected.

Stable evolution:

RCSS$_t$(S,F) = 1; the relative number of classes infected with code smells is stable.

The density of code smells in the system remains stable, if the absolute number of infected classes behaves proportional to the system size. This may imply that performed maintenance tasks did not lead to a further infection of classes. Furthermore it might show that added or deleted classes contain a proportional subset of infected classes.

Negative evolution:

$RCSI_t(S,F) = 1$; the relative number of classes infected with code smells is increasing significantly.

The increase of the relative number of classes infected with code smells may imply that functionality was added to the system by extending existing classes, which led to an infection with code smells or by adding new, infected classes. Another reason would be the reduction of functionality by deleting mostly *non*-infected classes.

Even more specific assumptions and interpretations of each phase are possible if we enrich this information with the observed evolution of the system size.

5.1.1. God Class. The first analyzed revision of Lucene (date *10/06/03*) had five (5) God Classes and the last version analyzed (*01/04/09*) had 21 god classes. The first analyzed revision of Xerces 2 J (date *11/03/99*) contained 12 God Classes. Its last analyzed version (*05/02/05*) had 69 God Classes. Those values are also the minimum and maximum of the detected absolute number of flawed classes. Figure 7 shows the evolution of the God Class code smell in Lucene, concerning the relative number of infected classes. Figure 8 shows the same information for Xerces.

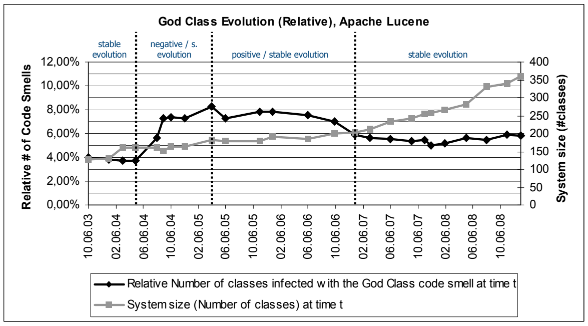

Figure 7. God Class Evolution (Relative), Lucene

In the historical data of Lucene we can identify four successive evolution phases. Initially we can see a stable evolution with increasing system size at the end of the phase. The second phase shows a negative evolution whereas the system size stays stable. A possible interpretation is that the maintenance tasks on existing classes led to an infection of some classes with the God Class code smell. This phase is followed by a positive and partially stable evolution. Since the system size does not show a significant change, this may imply that a refactoring was performed, focused on the reduction of God Class smells. Finally the system shows a stable evolution phase with an increase in system size. In this phase, new smells were introduced proportional to the absolute number of classes.

Xerces historical data can be partitioned into six God Class evolution phases. Initially we observed a positive evolution with a strong increase in the system size and a simultaneous decrease in the relative number of God Classes (but a slight increase of the absolute number of God Classes). The second phase shows a negative evolution since the number of code smells is increasing at a higher speed than the system size. The following phase indicates a positive / stable evolution with one negative outlier in the middle. Since the system size is increasing, at some points significantly (more than *10%*, relative to the size of the first revision), we can assume that new classes were added to the system with no or just a few God Classes. The fourth detected phase shows a negative / stable evolution. The last revision in the phase shows a dramatic reduction in the system size with the relative number of code smells increasing; this may imply that a subsystem was deleted which did not contain any smells or just a few. The next phase shows a stabilization of the system with a small increase in the relative number of God Classes. The last phase again shows a negative evolution with a significant decrease in system size, as before, this may imply the deletion of a subsystem with a low God Class density.

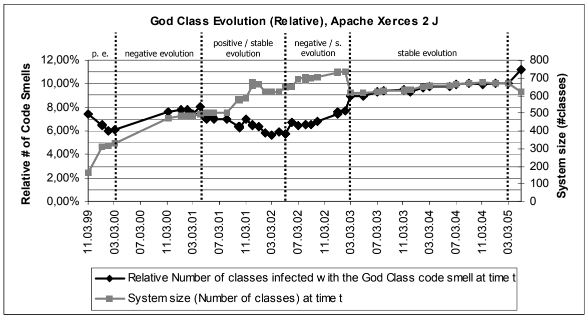

Figure 8. God Class Evolution (Relative), Xerces

Consider our initial hypotheses about God Classes: *H1* (The total number of code smells increases steadily) and *H2* (The relative number of components having code smells increases over time). Since we can identify revisions that show a reduction in the absolute number as well as in the relative number of detected smells, we should reject both hypotheses. However, we can identify a large correlation between system size and the number of God Class smells (0.83 for Lucene and 0.77 for Xerces). This also leads to the assumption that the bigger the size of a system the more God Classes it contains.

5.1.2 Shotgun Surgery. The first analyzed revision of Lucene (date *10/06/03*) had five (5) classes infected with the Shotgun Surgery code smell and the last version analyzed (*01/04/09*) had 17 classes. The first analyzed revision of Xerces 2 J (date *11/03/99*) contained three (3) infected classes. Its last analyzed version (*05/02/05*) had 69 classes featuring the Shotgun Surgery smell. Similar to god classes, these values are the detected minimum and maximum of the detected absolute number of classes with Shotgun Surgery. Figure 9 shows the evolution of the Shotgun Surgery code smell in Lucene, concerning the relative number of infected classes. And Figure 10 shows the same information for Xerces.

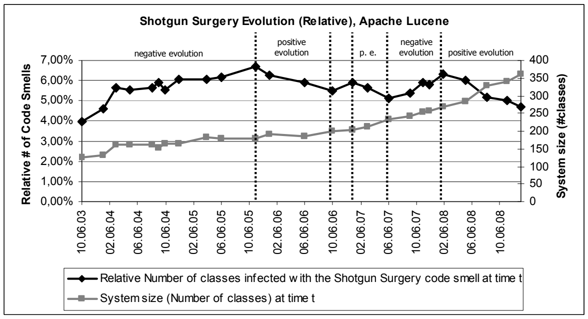

Figure 9. Shotgun Surgery Evolution (Relative), Lucene

The historical data of Lucene features six different phases concerning the evolution of Shotgun Surgery smells. Initially, we can identify a negative / stable evolution with one positive outlier; since the system size is not or only slightly increasing we can assume that maintenance tasks led to an infection of existing classes with the Shotgun Surgery smell. The second phase shows a positive evolution with a rather stable system size. This may imply that maintenance tasks were focused on an improvement of the design. This is followed by a small negative phase which led to an increase in smells. Since the system size is stable, this implies that existing classes got infected with code smells. In phase four we can see again a positive evolution; since the system size is increasing we can assume that smells where also reduced on purpose. The following phase features a negative evolution; since the system size is increasing we can assume that newly added classes which coupled with existing classes led to an infection of them with the Shotgun Surgery smell. The final phase shows a positive evolution with a simultaneous increase in the system size, which may imply that maintenance tasks were performed in order to reduce the existing, excessive low level coupling which also led to a reduction of the smells density.

Xerces' historical data also allows the partition into six phases. Initially we observed a negative / stable evolution connected with a simultaneous increase in the system size. This may imply that, since the system grew, new classes got coupled to existing classes which led to an infection with the Shotgun Surgery smell. Phase one is followed by a stable phase concerning the relative number of infected classes and the system size. After a short negative evolution phase with the same implications as the first phase, the evolution gets stable again with an insignificant subset of outliers. The fifth phase shows a negative evolution, with an initial significant decrease in the system size. This may imply that a subsystem with a low Shotgun Surgery density was deleted. Afterwards the system again shows a stable evolution with only insignificant changes in the system size and the number of infected classes. The last phase again shows a negative evolution with the same implications as in phase five, since the system size is significantly decreasing.

Since the observed evolution of the Shotgun Surgery code smell features intervals with a reduction of the absolute as well as the relative number of infected classes we can reject *H1* and *H2* for both projects. Still, we can see the same large correlation as for God Class smells, concerning the system size and the number of Shotgun Surgery smells (0.94 for Lucene and 0.88 for Xerces).

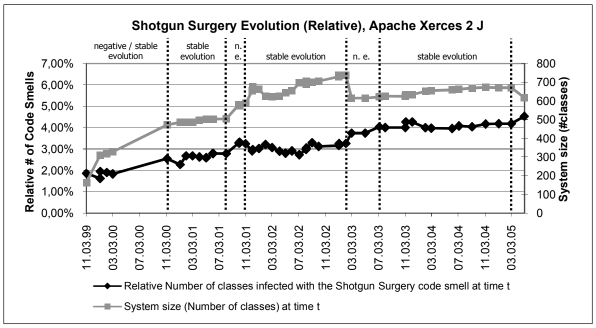

Figure 10. Shotgun Surgery Evolution (Relative), Xerces

5.2. Results of Research Question II

This section presents the results concerning the change frequency and change size of infected and *non*-infected classes. To test that the results were directly related to the investigated code smell and not co-influenced by the other smell. Therefore, we investigated the intersections of infected class in the last revision of both systems. There is an intersection of *0.28%* in Lucene and *0.97%* in Xerces of classes which exhibit both smells. Based on this insignificant intersection we can interpret the results for both smells independently of each other.

5.2.1 Part I: Entity change behavior. The results of this section address *H3* (The change proneness of components with smells is higher than the ones without).

To analyze the statistical difference in the change behavior of classes which are infected with a code smell and non-infected classes, we applied a two-sample *t-test* (Difference between means μ) on the collected data. The two samples are the change information of the classes with stable transitions *without* smell (*NCNC*) and the stable transitions *with* smells (*SCSC*) (See section 4.3.2). As level of significance, we use a p-value of *0.05*.

Table 3 shows the characteristics of the two samples as well as the result of the hypothesis test applied on the change data of Lucene and Xerces considering the God Class smell.

Table 3. Entity change behavior, Lucene and Xerces, God Class Code Smell

Stable transitions without smell			Stable transitions with smell		
	Lucene	Xerces		Lucene	Xerces
n_1	4602	26121	n_2	282	2148
$\overline{X}_1$	0.45	0.22	$\overline{X}_2$	1.79	1.08
s_1	0.86	0.64	s_2	2.61	2.20
Lucene: $H_A: {}_1 - {}_2 < 0; \quad p-value: 3.96 \cdot 10^{-16}$					
Xerces: $H_A: {}_1 - {}_2 < 0; \quad p-value: 2.28 \cdot 10^{-68}$					

Table 4. Entity change behavior, Lucene and Xerces, Shotgun Surgery Code Smell

Stable transitions without smell			Stable transitions with smell		
	Lucene	Xerces		Lucene	Xerces
n_1	4622	27388	n_2	276	924
$\overline{X}_1$	0.53	0.28	$\overline{X}_2$	0.68	0.67
s_1	1.09	0.88	s_2	2.61	1.54
Lucene: $H_A: {}_1 - {}_2 < 0; \quad p-value: 0.034$					
Xerces: $H_A: {}_1 - {}_2 < 0; \quad p-value: 1.91 \cdot 10^{-14}$					

Based on the applied two-sample *t-test* on the experimental data, we can see that classes which are infected with a God Class code smell get changed significantly more often in both projects. Therefore we can accept *H3*. This is an expected behavior since a god class describes a class which tends to centralize the intelligence of the system; consequently, it is necessary to ‘touch’ it more often than a regular class.

The calculated mean values ($\overline{X}_1; \overline{X}_2$) about the frequency of performed changes (within each 50th revision) show that god classes get changed *4* times more often in Lucene and *5* times more often in Xerces.

Table 4 shows the characteristics of the two samples as well as the result of the hypothesis test applied on the change data of Lucene and Xerces considering the Shotgun Surgery smell.

The analysis of the change behavior of classes infected with a Shotgun Surgery smell was mainly done for comparison to the god class behavior. The change behavior of classes infected with Shotgun Surgery smell does not allow an investigation of the actual maintenance issues caused by the smell (changes on the infected component cause a lot of minor changes on coupled components). For the analysis of this issue the change behavior of the coupled classes in relation to changes on class infected by Shotgun Surgery would have to be investigated. However, we can assume that classes infected with a Shotgun Surgery smell may show a higher change frequency, since they might be used as e.g., function libraries or data access classes, thus, their functionality has to be improved more often, which leads to more changes.

Similar to the God Class smell, classes which are infected with Shotgun Surgery show a significantly higher change frequency compared to *non*-infected classes for both projects. Based on the *t-test* result we can accept *H3* for Shotgun Surgery code smells.

5.2.2. Part II: Entity churn comparison. This section contains the results regarding H4 (The size of changes of infected classes is significant larger than the size of changes on non-infected classes.) Similar to the analysis of the change behavior, we also applied a two sample t-test to identify significant differences in the change size between classes which provide stable transitions with smell (*SCSC*) and the ones with stable transitions without smell (*NCNC*). Again we use a p-value of 0.05 as level of significance.

Table 5. Entity churn comparison, Lucene and Xerces, God Class Code Smell

Stable transitions without smell			Stable transitions with smell		
	Lucene	Xerces		Lucene	Xerces
n_1	1409	4249	n_2	169	993
$\overline{X}_1$	24.72	32.67	$\overline{X}_2$	45.05	43.91
s_1	35.91	68.65	s_2	387.90	140.11
Lucene: $H_A: {}_1 - {}_2 < 0; \quad p-value: 0.00027$					
Xerces: $H_A: {}_1 - {}_2 < 0; \quad p-value: 0{,}007$					

Table 6. Entity churn comparison, Lucene and Xerces, Shotgun Surgery Code Smell

Stable transitions without smell			Stable transitions with smell		
	Lucene	Xerces		Lucene	Xerces
n_1	1497	5007	n_2	100	296
$\overline{X}_1$	27.99	32.67	$\overline{X}_2$	25.32	36.03
s_1	44.61	101.18	s_2	27.27	69.73
Lucene: $H_A: {}_1 - {}_2 < 0; \quad p-value: 0.82$					
Xerces: $H_A: {}_1 - {}_2 < 0; \quad p-value: 0.55$					

Table 5 shows the characteristics of the two samples as well as the result of the hypothesis test applied on the change data of Lucene and Xerces concerning the God Class smell. The two sample *t-tests* on the experimental data shows that the size of changes is significantly larger for classes which are infected with the God Class code smell that non-infected classes. Thus we can accept *H4*.

The calculated mean values ($\overline{X}_1;\overline{X}_2$) concerning the average code churn (within each 50th revision) shows that the change size of god classes is 1.8 times higher in Lucene and 1.3 times in Xerces.

Table 6 shows the characteristics of the two samples as well as the result of the hypothesis test applied on the change data of Lucene and Xerces concerning the Shotgun Surgery smell.

The applied two sample *t-test* on the experimental data shows that there is no significant difference concerning the size of changes on classes infected with the Shotgun Surgery smell and non-infected classes. Thus *H4* is rejected. The calculated mean values ($\overline{X}_1;\overline{X}_2$) show that the average code churn is even lower (but not significant) for classes infected with the Shotgun Surgery smell.

6. Validity

In this section we will discuss the different internal and external threats of the validity in this study.

This was an exploratory study. Although we tried to select two projects for this initial study that were representative of open source projects, the results may not be generalizable to all open source projects or to non-open source projects. More evidence of these results needs to be gathered from other open source projects and software systems with different characteristics (e.g., system size, problem domain).

The detection of code smells in this study is based on detection strategies (See section 2.). Even though detection strategies deliver many advantages compared to manual code inspections, they offer the chance for measurement error, i.e., were the right metrics and thresholds chosen?

Another confounding factor is the aggregation of changes (See section 4.3). Since we only analyzed the infection status of each 50th revision it might be possible that we missed changes in the infection status between the analyzed revisions. This could affect our second research question. For example, it is possible that some change data is assigned to the wrong group since the respective entity had a different infection status at time t_0 and t_1 then within the interval (t_0, t_1). Since we can assume that such exceptions occur uniformly distributed for both cases we can ignore this issue. However, a more fine grained aggregation or the complete abdication of it would avoid such doubts.

As described in section 4.1, we analyzed only the public class for each source file. This was done because of restrictions of the tool framework used. Even if we can assume that this class is the most important, we might be losing information. Further studies should take multiple classes per source file into account.

Since this study is focused on the influence of code smells, we did not take other possible influencing factors into account, such as the authors that worked on the system or the underlying development processes. In future studies it is necessary to find a way to deal with those factors.

7. Conclusion

In this study we offered an approach for analyzing the evolution of code smells on software systems and the impact on the frequency and size of changes. The analysis identifies different phases in the evolution of the system that point to periods of time when 'good' or 'bad' evolution is occurring. Good or bad is with respect to the increase or decrease of code smells infected components. We believe that a more fine grained approach on the entity level would provide even more insights on the effects of the code smells in the software evolution (see section 5.1). Only an analysis which takes the evolution of each class into account would provide all the information needed for a comprehensive interpretation of the system evolution. However, as shown in this paper, the observed code smell density can be used to show some patterns in the evolution of the software systems and can be used to compare behaviors.

With regard to change behavior, in this study the classes infected with code smells have a higher change frequency; such classes seem to need more maintenance than *non*-infected classes; furthermore God Classes feature bigger changes Thus, the probability is higher that maintenance tasks take more effort since more modifications to the code (i.e., modifying, adding or deleting) is required. Initially we made the assumption that classes infected with shotgun surgery would have a lower change frequency since people would be hesitant to touch them because this implies too many changes on coupled classes and thus too much effort. The results here show that, in fact, those classes are touched much more than normal classes. One possible explanation for this is that since they provide functionality for a lot of components in the system, their functionality has to be extended more often. The monitored frequency of changes and change size can be used as indicators of when refactoring on a specific entity is necessary. The results also may imply that the negative impact of shotgun surgery is lower compared to god class since the changes are not as big, but further analysis needs to be performed in order to verify the results in other systems and investigate the issues caused by the afferent coupling.

Contributions of this work include the application of strategies and code smells to the analysis of historical data on the evolution of software systems. Specifically, the analysis of two large, multi-year developments, opening the opportunity for analysis of a variety of systems and diverse analysis on the behavior of the systems infected with code smells.

Based upon this first insight into the evolution of code with respect to code smells and their influence on the change frequency and size in open source projects, it would be interesting to investigate the evolution of closed source projects for comparison as well as other open source systems with different characteristics. A long term goal would be the creation of thresholds for acceptable levels of code smell densities in different environments. For Shotgun Surgery in particular, we want to investigate afferent coupling issues to better understand its impact on the software evolution. We intend on implementing other code smells detection strategies and analyze their impact on the software system evolution as well as analyzing the effects of multiple code smells in a class.

8. Acknowledgements

This research was supported in part by NSF grant CF0438933, "Flexible High Quality Design for Software" to the University of Maryland. The authors would like to thank Dr. Forrest Shull for comments during the work on this project.

9. References

[1] Bieman, J.; Kang, B. (1995). Cohesion and reuse in an object oriented system. Proceedings of ACM Symposium on Software Reusability (SSR'95), 259-262.

[2] Chidamber, S. R.; Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):476-493, June 1994.

[3] Deligiannis, Ignatios; Stamelos, Ioannis; Angelis, Lefteris; Roumeliotis, Manos; Shepperd, Martin (2003). A controlled experiment investigation of an object oriented design heuristic for maintainability Journal of Systems and Software, v.65 n.2, p.127-139, 15 February 2003.

[4] Deligiannis, I.; Shepperd, M.; Roumeliotis, M.; Stamelos, I. (2004). An empirical investigation of an object-oriented design heuristic for maintainability. The Journal of Systems and Software 72 (2), 129-143.

[5] Demeyer, S.; Ducasse, S.; Nierstarsz, O. (2002). Object-Oriented Reengineering Patterns. Morgan Kaufmann.

[6] Fowler, Martin (1999). Refactoring. Improving the Design of Existing Code. Addison-Wesley.

[7] Johnson, R.E.; Foote, B. (1988). Designing reusable classes. Journal of Object-Oriented Programming, Journal of Object Oriented Programming 1, 2 (June/July 1988), 22-35.

[8] Koru, A. Günes (2005). Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Projects. IEEE Trans. Software Eng. 31(8): 625-642 (2005).

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

[10] McCabe, T. (1976). A measure of complexity. IEEE TSE 2(4):308-320, Dec. 1976.

[11] Mäntylä, Mika. Vanhanen, Jari and Lassenius, Casper (2003): A Taxonomy and an Initial Empirical Study of Bad Smells in Code. ICSM 2003: 381-384.

[12] Mäntylä, M.; Vanhanen, J.; Lassenius, C. (2004). Bad Smells - Humans as Code Critics. ICSM 2004: 399-408.

[13] Marinescu, R. (2001). Detecting Design Flaws via Metrics in Object-Oriented Systems. TOOLS (39): 173-182.

[14] Marinescu, Radu (2002). Measurement and Quality in Object-Oriented Design. Ph.D. Thesis, Department of Computer Science, "Politehinca" University of Timisora.

[15] Marinescu, Radu (2004). Detection Strategies: Metric Based Rules for Detecting Design Flaws. ICSM 2004: 350-359.

[16] Marinescu, Radu; Lanza, Michelle (2006). Object-Oriented Metrics in Practice. Springer.

[17] Ratiu, Daniel; Ducasse, Stéphane; Girba, Tudor; Marinescu, Radu (2004). Using History Information to Improve Design Flaws Detection. CSMR 2004: 223-232.

[18] Riel, Arthur (1996). Object-Oriented Design Heuristics. Addison Wesley.

[19] Rising, L.S.; Calliss, F.W. (1993). An experiment investigating the effect of information hiding on maintainability. 12th Ann. Int. Phoenix Conference on Computers and Communication, March, pp. 510-516.

[20] Wilde, N.; Mathews, P.; Ross, H. (1993). Maintaining Object-Oriented Software. Addison-Wesley.