

Tracking Design Smells: Lessons from a Study of God Classes

Stéphane Vaucher Foutse Khomh

GEODES / Ptidej Team

Dépt. d'Informatique

Université de Montréal

Montreal, Canada

Email: {vauchers,foutsekh}@iro.umontreal.ca

Naouel Moha

Triskell Team

IRISA / Université de Rennes 1

Rennes, France

Email: moha@irisa.fr

Yann-Gaël Guéhéneuc

Ptidej Team

Dépt. de Génie Informatique

École Polytechnique de Montréal

Montréal, Canada

Email: yann-gael.gueheneuc@polymtl.ca

Abstract—“God class” is a term used to describe a certain type of large classes which “know too much or do too much”. Often a God class (GC) is created by accident as functionalities are incrementally added to a central class over the course of its evolution. GCs are generally thought to be examples of bad code that should be detected and removed to ensure software quality. However, in some cases, a GC is created by design as the best solution to a particular problem because, for example, the problem is not easily decomposable or strong requirements on efficiency exist. In this paper, we study in two open-source systems the “life cycle” of GCs: how they arise, how prevalent they are, and whether they remain or they are removed as the systems evolve over time, through a number of versions. We show how to detect the degree of “godliness” of classes automatically. Then, we show that by identifying the evolution of “godliness”, we can distinguish between those classes that are so by design (good code) from those that occurred by accident (bad code). This methodology can guide software quality teams in their efforts to implement prevention and correction mechanisms.

Keywords—Design smells; software evolution.

I. INTRODUCTION

Quality analysis (QA) teams are concerned with identifying problematic pieces of code in the systems developed by their colleagues. Usually, QA teams first focus on the most important parts of a system (*e.g.*, its kernel) because of their limited resources in time and personnel; their remaining efforts then focus on the more risky parts of the system, using for example static code analysis to identify them.

Static code analysis can detect structural patterns in systems that are signs of poor design decisions like code and design smells [1], [2]. Code and design smells are poor solutions to recurring implementation and design problems [1]. An example of a typical and recurrent smell is the God class (GC), also called Blob [3], which defines a class that “knows too much or does too much” and centralises many functionalities. More precisely, a GC corresponds to a large controller class that depends on data stored in surrounding data classes. GCs are considered in the literature to be a bad programming practice [3], [4].

Although generally considered bad, there are cases where GCs are the most reasonable solution to a problem. For example, parsers are notoriously difficult to decompose, resulting in very large and complex classes. To the best of our knowledge, no previous work has studied the origin of smells (and in particular GCs), how they have been introduced, removed, and also how they evolve in systems. Additionally, few studies have proposed methods/techniques to prevent and correct these smells.

Guiding Metaphor: As in previous work [5], we cast our empirical study as a problem analogous to research in the field of *epidemiology*. Epidemiology is the study of factors conditioning the appearance, frequency, way of diffusion, and evolution of a disease to plan its prevention and treatments. Following the metaphor of epidemiology, researchers on software maintenance study the factors leading to software problems (*e.g.*, unstructured code).

This paper presents an exploratory analysis of the “life cycle” of GCs: (1) we use a Bayesian approach to detect the presence of GCs in systems and rank them; (2) we study the evolution of these GCs in systems; specifically, we study how GCs are introduced in and removed from the systems and how they evolve. The result of this study allows us to (3) discuss a predictive model to prevent their introduction: given a change request, how likely will a class become a GC. Finally, (4) we discuss how refactorings could be used to treat classes identified as God classes.

Our study leverages our previous work in GC detection [6], in which we developed a Bayesian-based smell detection model and tested it on two open-source systems. This model was shown to accurately detect all existing occurrences while raising few false alarms. For the present work, this model is used to evaluate different versions of two open-source systems, Xerces and Eclipse JDT. We also examine the effect of different code changes on the likelihood that classes become GCs. This study shows that a large proportion of GCs seems to be introduced as a conscious design decision by developers; and that specific maintenance activities can eliminate GCs when they are accidents.

Organisation: Section II relates our study with previous work. Section III presents our study along with the evolution of GCs in systems and the techniques used in the study. Section IV presents the process of building a model for the prevention of GC and suggestions of corrections, while Section V discusses the results of our study, along with threats to their validity. Finally, Section VI concludes the paper and outlines future work.

II. RELATED WORK

Many papers address various aspects of software evolution but few deal specifically with the evolution of code and/or design smells.

Smell Definition and Detection: Code and design smells include low-level or local problems such as code smells [1], which are usually symptoms of more global design smells such as antipatterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [1] presented 22 code smells, structures in the source code that suggest the possibility of refactorings. Code smells such as duplicated code, long methods, large classes, and long parameter lists are just a few symptoms of design smells and opportunities for refactorings. Brown *et al.* [3] described 40 antipatterns, including the well-known Blob and Spaghetti Code. Riel [4] defined 61 heuristics characterising good object-oriented programming to assess and improve manually a system design and implementation; alluding to bad programming practices, such as GCs.

Several approaches to specify and detect code smells and antipatterns have been proposed. They range from manual approaches, based on inspection techniques [7], to metric-based heuristics [8], [9], where antipatterns are identified according to sets of rules and thresholds defined on various metrics. Rules may also be defined using fuzzy logic and executed by means of a rule-inference engine [10]. Some approaches for complex software analysis use visualisation techniques [11], [12]. Such semi-automatic approaches are an interesting compromise between fully automatic detection techniques that can be efficient but lose track of the context and manual inspections that are slow and subjective [13]. However, they require human expertise and are thus time-consuming. Other approaches perform fully automatic detection and use visualisation techniques to present the detection results [14], [15].

This previous work has contributed significantly to the specification and automatic detection of code and design smells. The approach used in this study, builds on these previous works, especially [6] and [2], and offers a probabilistic method to study the evolution of smells in systems.

Smell Evolution: Recently, some work studied the impact of code smells and of one design smell (God class) on evolution-related phenomena. In a direction of research related to the impact of smells on program comprehension, Du Bois *et al.* [16] showed that the decomposition of GCs

into a number of collaborating classes using well-known refactorings can facilitate comprehension. Independently, Lozano *et al.*'s work [17] raised several research questions related to the impact of smells on maintainability and suggest different research directions. In a paper in the same proceedings as this one, Khomh *et al.* [18] studied the impact of classes with code smells on change-proneness and the particular impact of certain code smells, using Azureus and Eclipse. They showed that the risk that classes with code smells change is very high, except in a few explainable cases.

This previous work raised the awareness of the community towards the concrete impact of code smells and antipatterns. In this study, we focus on one design smell to understand its evolution and discuss its possible prevention and correction.

Software Evolution: In recent years, much work has been done on problems related to the evolution of systems. A dedicated workshop exists since 1998, the International Workshop on Principles of Software Evolution. We summarise some important lines of work on software evolution.

A direction of research studies the evolution of systems to identify co-changing artifacts. Zimmermann *et al.* [19] extended previous work [20], [21], [22], [23] to recover co-changing fine-grain entities (classes, methods, fields...). They suggest likely future changes by detecting causal couplings between entities to prevent incomplete changes. German [24] abstract co-changing files into modification requests and analyses their interrelationships and authors. Bouktif *et al.* [25] identify recurring patterns in the evolution of co-changing files.

Another direction of research provides help to understand system evolution through visualisation techniques. For example, Eick *et al.* [26] developed tools to visualise the evolution of software measures and change data, including size and effort. Ratzinger *et al.* [27] proposed an approach called EvoLens to explore evolution data across multiple dimensions. It allows visualising important relationships across module boundaries based on customisable views. Xie *et al.* [28] presented several visualisation techniques integrated in a tool called CVSViewer3D. This tool offers system-, file-, function-, and line-level views. It allows, for example, to highlight all changes made by one developer.

Yet another research direction investigates the evolution of systems to infer information about these systems. Bakota *et al.* [29] studied the evolution of clones across software versions to track those occurrences of clones that could become problematic in the future versions. They reuse the clone detection technique available in COLUMBUS and define a similarity measure to relate two code fragments in two versions of a system. Then, they studied the evolution of clones and defined four major categories of clones according to their evolution pattern: vanished, occurring, moving, and migrating clones. Demeyer *et al.* [30] defined four metric-based heuristics to identify the refactorings applied between two versions of a system.

Finally, in another research direction, researchers performed evolution analyses trying to infer design and/or architectural knowledge about a system. For example, Egyed [31] proposed an approach to check the consistency of evolving UML diagrams. In a series of papers, Xing and Stroulia [32] proposed an approach to analyse the evolution of the logical designs of systems. They proposed three types of longitudinal analyses: phasic, gamma, and optimal matching, to recover distinct evolutionary phases and their styles, thus helping in evolving the system consistently. Mens *et al.* [33] performed a metrics-based study of the evolution of Eclipse, the popular integrated development environment. Their study consisted in evaluating Eclipse against three laws of software evolution (law 1: *Continuing Change*, law 2: *Increasing Complexity*, and law 6: *Continuing Growth*) using size and complexity indicators (such as number of classes, lines of code, number of defect reports). They found that Eclipse changed and grew continuously, hence supporting laws 1 and 6. However, the increase in complexity (law 2) of Eclipse was only partially supported. Wermelinger *et al.* [34] performed a quite similar case study as [33].

We get inspiration from this previous work to study the evolution of smells that we consider to be diseases that can infect healthy software systems. Our ultimate goal is to prevent and cure such infections.

III. THE EVOLUTION OF GOD CLASSES

Following our metaphor, just as a disease can affect the health of a person, *smells* can affect the “health” of a system. Moreover, it is usually preferable to prevent an infection than to cure it afterwards. However, prevention is quite difficult without understanding how the disease can be contracted. In particular, we must know its causes (*e.g.*, smoking causes cancer) and/or its transmission mechanisms (*e.g.*, airborne infection in public places). There are limits to what prevention can do and treatments for curing a disease are also needed.

Consequently, we study factors conditioning the appearance, frequency, way of diffusion, and evolution of GCs—a recurrent and potentially harmful “disease” in object-oriented systems. Our ultimate goal is to find ways to prevent the introduction of GCs and facilitate their removal. We choose to study GCs because they occur frequently in object-oriented systems and impact negatively the quality and maintenance of systems.

As in epidemiological studies, we selected a population that contains “infected” classes. The two open-source systems, Xerces and Eclipse JDT, serve as our population. We identified GCs in the different versions of these systems using a detection model. Then, we identified and classified different evolution trends of GCs that indicate how the level of “godliness” of these classes varies throughout the life cycle of the systems. The identification of the evolution trends was performed using a classification technique based

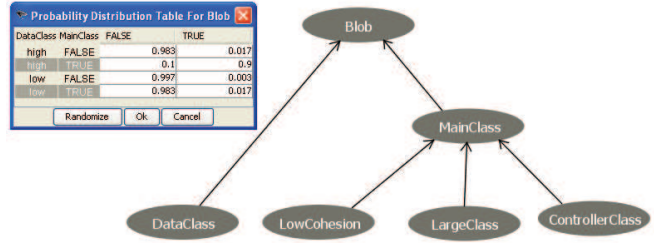


Figure 1. GC Detection Model

on dynamic time-warping (DTW). These trends are then used to understand how these classes become GCs and could be corrected. Prevention and correction mechanisms are then discussed based on these evolution trends.

In the following, we provide a short description of the two systems, our “population”. We also describe succinctly the detection model and the DTW technique. Then, we present the evolution trends of GCs. The mechanisms of prevention and correction that could stem from the evolution trends are described in the next section.

A. Population

The population of our study consists of two well-known, industrial-strength, open-source systems: Xerces and the Eclipse JDT sub-project. Xerces¹ is a family of software packages for parsing and manipulating XML. It implements a number of APIs for XML parsing, including DOM and SAX2. Implementations are available for C++ and Perl.

The Eclipse JDT sub-project² is an Eclipse plug-in that implements the infrastructure for the Java IDE of the Eclipse platform. It provides a Java model and capabilities to parse, manipulate, and rewrite Java programs. Eclipse has been developed partly by a commercial company (IBM), which makes it more likely to embody industrial practices. It has also been used in related studies, *e.g.*, [33], [34].

These systems were chosen because they are long-lived and each contains several hundred to several thousand of classes. Furthermore, they are from different problem domains and follow different development processes, two factors affecting design quality. We analysed 36 releases of Xerces from 1999 to 2006 and 22 releases of Eclipse JDT available from 2001 to 2008.

B. Detection Model

The identification of GCs in systems has been performed with the detection model presented in [6]. From a rule card describing the detection rules, we build a Bayesian belief network describing a probabilistic model of the rule card, as shown in Figure 1.

¹<http://xerces.apache.org/>

²<http://eclipse.org>

This model is based on metrics for characterising specific classes and computes the probability that these specific classes are GCs. The inputs used in the model include: (1) the size of the class (its number of methods and attributes), (2) its cohesion (using Henderson-Sellers’ LCOM5 [35]), (3) the number of associated data classes, and (4) a lexical analysis of the names of a class and its methods.

The model was calibrated by learning the conditional probability tables from manually-validated data. This data relate to Xerces v2.7.0, in which we asked two undergraduate students and two graduate students to detect occurrences of the Blob in the two systems. The pair of undergraduate students performed the task together [36]. In a previous work [6], we showed that building the model using a system and applying on another gives consistent results, thus avoiding the problem of over-fitting, *i.e.*, of false positives.

The output of this model is a real value between 0 and 1 that defines the probability that the class is a GC, which we call “Godliness”. The output probability enables us to rank classes, which cannot be identified by traditional detection techniques. More exactly, we can track the evolution of all classes throughout the existence of a system and identify when and how they degenerate into GCs.

We use the highest probability of the classes manually-validated as being a GC in Xerces v2.7.0 as threshold to tag classes in Eclipse as GCs, 45%.

C. Global Evolution Trends of God Classes

The first step in the study consisted of evaluating the number of GCs present in every version of a system to identify possible global trends. Figure 2 presents the ratio of GCs (right axis) from one version to the next as well as the growth of the system (left axis, in number of classes). The figure shows that the growth of both systems is relatively linear. The different plateaux (both curves) correspond to minor versions during which few new classes/GCs are added. The proportion of GCs is relatively stable in Eclipse (2%) but varies significantly in Xerces (10%–15%).

Table I
INTRODUCTION AND REMOVAL OF GOD CLASSES.

	Xerces	Eclipse JDT
Nb of GCs (%)	138 (18%)	144 (3%)
Nb of GCs from introduction (%)	97 (70%)	88 (61%)
Nb of GCs deleted (%)	41 (30%)	27 (19%)

Table I summarises descriptive statistics concerning the introduction and removal of GCs. Globally, Xerces and Eclipse JDT have 138 and 144 classes that were GCs at some point in their existence. A large number of these GCs were GCs from their introduction: 70% for Xerces and 61% for Eclipse JDT. Later in the life cycle of the systems, these classes have been deleted in the proportion of 30% in Xerces and 19% in Eclipse JDT.

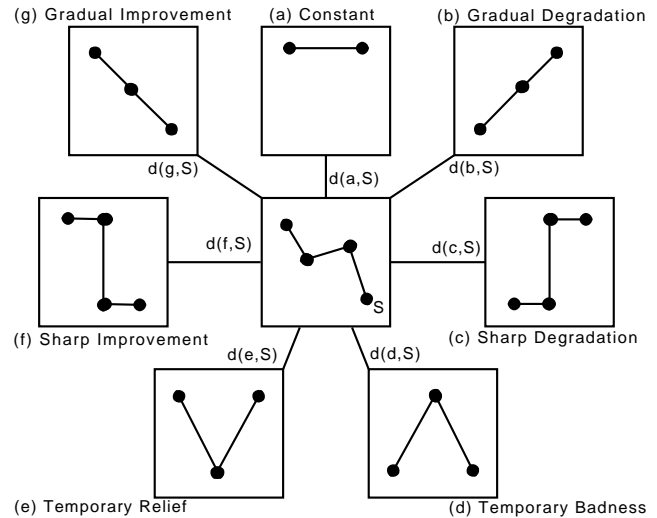


Figure 3. Evolution Trend Classification

D. Dynamic Time-Warping

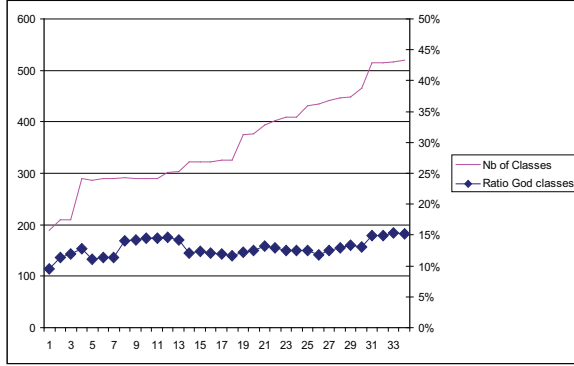
The evolution of a class with respect to its level of “godliness” can be represented as a signal $S = (s_1, s_2, s_3, \dots, s_n)$ where $s_i, 1 \leq i \leq n$ is the probability that a class is a GC at the version i . Version i represents a version in which a class appears in the system and n is the total number of versions in which the class exists and is analysed.

Our objective is to classify S according to meaningful change stereotypes and thus guide our subsequent study of the evolution of GCs. To reach this objective, we use a classification technique based on dynamic time-warping (DTW). DTW was first presented by Kruskal and Lieberman [37] to compute a time-independent comparison of pairs of signals. This technique finds a “topological” distance between two signals by modifying the time axis of each one. For example, it can align two signals with peaks at different times. This ability to modify time is important because changes in classes tend to happen irregularly, often independently of the system versions. A specific signal is classified according to its proximity to the nearest stereotype.

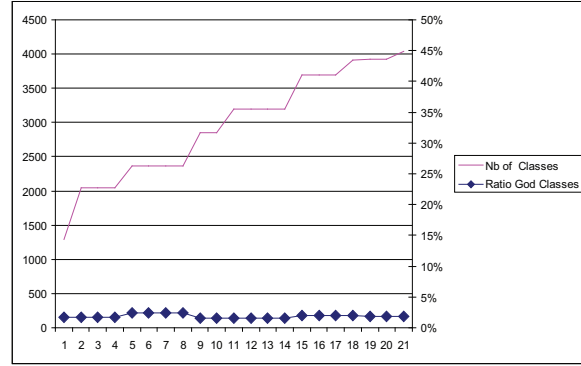
E. Classification of the Evolution Trends

We present the results of the classification of the evolution trends of GCs on Xerces and Eclipse JDT. Figure 3 presents this classification (the seven surrounding plots, from a to g) and illustrates it with an example: the evolution trend of class `org.apache.xerces.impl.XMLVersionDetector` in the center. S represents the signal to be classified, *i.e.*, the probability that class `XMLVersionDetector` is a GC, and $d(x, S)$ corresponds to the distance calculated by the DTW algorithm between the signal S and a stereotype x .

The different stereotypes are defined by two or three point configurations where every point can either have a none, medium, or high value. Low and high correspond to



(a) Number of GCs in Xerces from 1.0.1 to 2.9.0



(b) Number of GCs in Eclipse JDT from 1.0.0 to 3.4.0

Figure 2. God Class Ratios vs. Total Classes

the lowest and highest godliness probabilities possible, and medium is $(low + high)/2$. Only three points are needed because the DTW can stretch the signal as much as needed. S can have however different values. The seven stereotypes describe different common evolution trends that have been observed using the DTW-based clustering technique (described in [25]) on the GCs in Xerces and Eclipse JDT.

The *Constant* stereotype corresponds to a stable signal where the class is always tagged as a GC. *Gradual improvement* corresponds to a class that starts with a high probability of being a GC, probability which drops to a medium level before becoming low. *Sharp improvement* is similar but the transition is abrupt: the signal level drops from high to low in a single version. *Gradual degradation* and *Sharp degradation* show the same phenomenon except describing design degradation. Finally, *Temporary relief* and *Temporary badness* are stereotypes of classes that are only temporarily GCs. To classify an evolution signal S , we compute the distance with the different stereotypes. The DTW algorithm finds $d(x, S)$ as the minimal distance between x and S .

The distribution of different trends is presented in Figure 4. In Xerces, out of the 138 classes that were GCs at some point in their existence, 91 (66%) showed no significant variations and followed the constant stereotype. 16 GCs (12% corresponding to improvement trends) were corrected by developers. This is significantly fewer than the number of GCs removed by developers (41, see Table I). 22 (16% corresponding to degradation trends) classes presented different degradation symptoms.

Similarly, Eclipse JDT (containing 144 GCs) had a large number of stable GCs, 96 (63%). In this system, almost as many GCs were corrected, 21 (14% corresponding to improvement trends) as were deleted (27, see Table I). Finally, 27 (19%) classes saw their quality degrade. We consider that the classes corresponding to Temporary badness and relief are instances of both an improvement and a degradation.

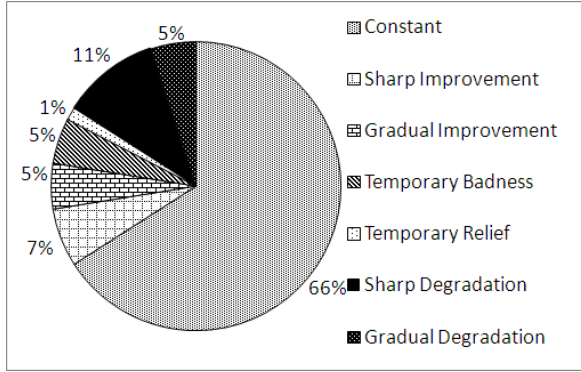
This classification process highlights three main types of evolution trends of interest: improved, degraded, and con-

stant GCs. Analysing each group can provide key insights into the nature of these complex classes, why they still exist or become more complex, and how they are improved. We now analyse in greater detail each of these evolution trends.

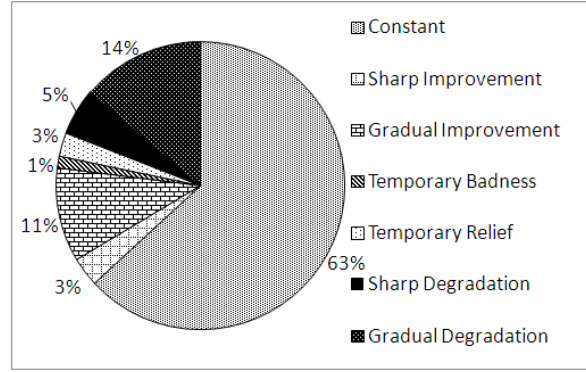
Constant: By far, this group is the largest stereotype. It contains a high number of GCs that are introduced at the very beginning in systems and that remain GCs throughout the existence of the systems. We investigated the motivation of developers for creating and adding large classes with lots of functionalities to a system. When asked, the primary developer of Xerces mentioned that these classes were as complex as the problems addressed. Independently, we verified the use of design patterns [38] in these classes as it could indicate a clear intention by developers to write clean code; the structure of the classes is no accident. We found that 82% of the classes from this group were playing roles in at least one of the following design patterns: Abstract Factory, Adapter, Observer, and Prototype.

Degradation: The GCs observed in this category have two different reasons for their degradation: either they gained new responsibilities (and grow in size) or they gained new data classes. In EclipseJDT, for the most part, these GCs are very large classes from their introduction. The main reason why their observed quality degrades is due to the addition of data classes. This can be explained by the particular use of data classes in Eclipse: often, data classes are used to communicate data between different application layers. One typical case is `org.eclipse.jface.text.Region`, a data class that describes a certain range in an indexed text store. This data class is in fact a value object used to transmit information from one system layer to another. One large class that uses it is `org.eclipse.jdt.internal.debug.ui.snippeteditor.JavaSnippetEditor`. Although there seems to be a justification for such design, any large class that interacts with a large number of these value objects, centralises behaviour, a symptom of GCs.

For Xerces, 11 of 15 sharp degradations are due to a similar situation: the quality of a class degrades because



(a) Evolution Trends Distribution in Xerces



(b) Evolution Trends Distribution in Eclipse JDT

Figure 4. Evolution Trends Distribution of God Classes

it is already large and developers add new data classes. Gradual changes, however, were all incurred by additional code. Table II summarises the changes occurring in the degraded classes. In this table, two different growth rates are presented: the average relative size increases in instructions and in methods. The number of versions indicates the duration of the gradual degradation. Sharp degradations show an average increase of 150% in instruction size and 86% in method size between a pair of versions. In the case of gradual degradations, the average change rates per version is 65% for instructions and 41% for methods. Not presented in the table, the total changes in gradual degraded classes tend to be equivalent to that of the sharp degradations. Other metrics were considered like cohesion but were not significantly impacted.

Table II
DEGRADATION GROWTH RATES IN XERCES

Degradation trend	Growth rate instructions/version	Growth rate methods/version	Nb of versions
Sharp	363.64%	162.50%	1
Sharp	138.44%	283.33%	1
Sharp	214.98%	113.64%	1
Sharp	513.33%	100.00%	1
Gradual	40.99%	16.67%	2
Gradual	142.90%	6.67%	3
Gradual	9.46%	15.38%	5
Gradual	65.04%	6.25%	2
Gradual	34.28%	155.56%	3
Gradual	63.46%	42.86%	2
Gradual	103.33%	45.00%	4

The results presented in Table II seem to indicate that, when performing a modification on a class, developers should pay attention to the size of the changes made, as they may induce a degradation of the quality of the class. This will be discussed in greater detail in Section IV.

Improvement: Brown *et al.* [3] defined GCs as a large complex class associated with many data classes. Fowler [1] suggested refactorings to correct both large classes and data classes. To correct large classes, possible refactorings

include *Extract Class*, *Extract Subclass*, and *Extract Interface*. When correcting a data class, the main concern is to limit access to its public attributes using the *Encapsulate Field* refactoring and then to add functionalities using the *Move Method* and *Extract Method* refactorings.

In our investigation of the improved GCs, we analysed the different classes to identify if and what refactorings were applied. The results are presented in Table III. The vast majority of refactorings found in Xerces were not refactorings “by the book”. In fact, four times, developers of Xerces extracted new super classes (indicated by *), a refactoring not explicitly mentioned by Fowler as a solution to GCs. Furthermore, in three cases out of five, the extracted classes became new GCs. We observed thus that the correction of GCs may induce the creation of new GCs.

Table III
REFACTORINGS IDENTIFIED IN XERCES FOR THE CORRECTION OF GOD CLASSES

Improvement	Refactoring	Nb (%)
Sharp	Move Method to data class	5 (31%)
Gradual	Move Method from GC	2 (13%)
Sharp	Extract Superclass* from GC	4 (25%)
Sharp	Extract Class from GC	1 (6%)

In the following, we suggest prevention and correction mechanisms that include a prediction model for preventing the introduction of GCs. This mechanism is based on the information gleaned from the evolution trends.

IV. PREVENTION AND CORRECTION OF GOD CLASSES

Our preliminary study indicated that there are factors that can be used to both prevent the introduction and correct GCs. We start by presenting a prediction model that, given the size of a change, indicates how the godliness of a class will evolve. Then, we present how refactorings can be used to best correct a GC.

A. Preventing the Introduction of God Classes

When developers want to implement new functionality, they should be able to offer an estimate of the work involved. In our study, we identified two issues causing quality degradation: the size of changes in methods and in instructions. We therefore built upon this observation to build a prediction model that, given a code change, predicts the likelihood that it will cause a GC. Being able to estimate the direct impact (small, medium, or large) of a change on a class is important as changes can result in a degradation of classes. A good assessment of the impact of changes could help to prevent this decay in GCs: a developer informed of the negative consequences of a change could anticipate and implement an alternative change.

We used previously tested change metrics to quantify these effects: three different instruction change metrics [39] and two public interface change metrics. We calculated the absolute and relative numbers of added, removed, and modified instructions (in terms of bytecode instructions) using a Levenshtein edit distance [40]. We also measured the absolute and relative changes in the numbers of interface methods (*i.e.*, public declared or inherited methods) as in [39]. The values were discretised into the following levels: *none*, *low*, *high*. *None* indicates that there were no changes. We used the third quartile (Q_3) of observed values as a threshold to separate the low from high values.

The predicted value is the variation on the level of symptoms exhibited. After a change, the quality of a class can degrade (its probability of being a GC increase), it can be stable, or it can improve (GC decrease).

We built and executed the prediction model on Xerces using JRip, the implementation of the RIPPER rule-extraction algorithm in Weka [41]. Tests were run using a 10-fold cross-validation. Figure 5 provides the results of the rules. RULE 1 of the model warns against modifications on borderline classes (classes with already high symptoms of GCs) because adding a large number of instructions to these classes would cause them to decay into GCs. RULE 2 moreover states that any change that does not reduce the complexity (measured in terms of size) of a class will likely increase the risk that this class becomes a GC. Finally, any other change would likely not affect the godliness of a class.

The prescriptive ability of this model is essential to prevent the introduction of a GC in a system. Although a developer cannot provide specific metrics to describe a change, she should be able to describe it sufficiently for the model to be useful. If a change is judged “bad”, the developer can test alternate changes. The RIPPER algorithm was not able to identify rules that predict change operations decreasing the level of godliness. This problem could be attributed to the simplicity of the metrics used. Future work includes a detailed study of relevant metrics and building a more powerful prediction model.

```
RULE_CARD : God Classes {
RULE 1: {
  ( Class status = Borderline ) AND
  ( Ratio of instructions
    added and/or deleted = high )
  AND ( Instruction Change Ratio = high )
  => Godliness = Increase (83 %)
};
RULE 2: {
  ( Class status = Healthy ) AND
  ( Instructions Deleted = none )
  => Godliness = Increase (66 %)
};
RULE 3: {
  ( Default => Godliness = Stable (74 %)
};
}
```

Figure 5. Rules of the Prediction Model. (Classification rates.)

B. Proposing Remedies to God Classes

We discuss how to “cure” infected classes using refactorings. In our exploration of improvements in Xerces and EclipseJDT, we found that developers use some refactorings on GCs. In EclipseJDT, most corrections came from adding behaviour to data classes, and in Xerces, we found that developers would extract classes (often into super-classes). We therefore present a process that could suggest the most appropriate refactorings to cure a specific GC.

There are three structural issues that describe a GC: its size, its cohesion, and its reliance on data class. Different refactorings can address a different set of these issues. The suggestion process could allow a developer to describe visible characteristics of a GC using metrics. For example, she might notice that a class C implements too much functionality, which is measurable using the NMD (number of methods) metric. She would like to know what is the best refactoring to apply. We present different symptoms and suitable refactorings to solve them.

Too Many Data Classes: A data class is loosely defined as a data holder without behaviour. Any corrections of this symptom consists of adding behaviour to the data class. First, a developer encapsulates the fields and then moves behaviour into the data class. The measurable effect of these refactorings on the structure of the observed classes are defined in Table IV. In the table, NPF and NPF_r are the number of public fields declared and removed. NMD is the number of declared methods (excluding accessors), where NMD_a and NMD_m are respectively the number added and moved from a GC. Finally, DC is the number of data classes. Both the number of data classes and the number of methods declared are explicitly used in the detection model. In our study of Xerces and EclipseJDT, this refactoring was found to be commonly used to correct GCs.

Too Much Behaviour in the God Class: When a class implements too much behaviour, it can be advisable to *Move Method* out. The effect is that the number of methods declared (NMD) in the GC decreases.

Table IV
CORRECTING DATA CLASSES

Refactoring	Data class	GC
1) Encapsulate	$\#NPF - NPF_r$	
2) Move method	$\#NMD + NMD_a$	$\#NMD_m - NMD_m$
3) Result		$\#DC - 1$

Table V
CLASS EXTRACTION

Refactoring	GC
1) New class/sub/superclass	$\#\text{Assoc./NOC} + 1/\text{DIT} + 1$
2) Move methods out	$\#NMD - NMD_m$
3) Move attributes out	$\#NAD - NAD_m$
Results	Cohesion is better, Size is smaller

Too Much State: When there are too many attributes, a hidden class looms that must be extracted using, for example, *Extract Class*. Fowler suggests for this class to be a new associated class or a subclass. We have however also observed that a superclass seems to be an alternative in three cases in Xerces.

The result on metrics (in Table V) is that the size of the class should decrease (as measured by the number of attributes and methods declared). Depending on the choice of attributes and methods moved out, the cohesion should also increase. If the extracted class is a super/subclass, then the position of the class in the inheritance tree should change (measured by the metrics Number Of Children or Depth of the Inheritance Tree). We are aware of the risk of transmission: the new class may be a GC or a data class.

The suggestion process consists of evaluating the main issue with the GC. If it has too many data classes, then methods should be moved there. If it is large and non-cohesive, then a developer should extract a new class. Finally, if the class has too much behaviour, methods should be extracted. While we discussed only these three types of suggestions, more could be considered and included in an automated suggestion system [42]. In future work, we plan on guiding the selection of refactorings using optimisation techniques to find a balance between the metric values characterising design and code smells.

V. DISCUSSIONS

Following an epidemiological metaphor, we conducted an exploration of how GCs are introduced and removed from software systems. While the results are observed on two different systems; the methodology, using a time-independent classification, could be reused in further research to support our findings.

GCs were identified using a detection model that was shown in previous work to identify GCs with a precision of 77% and a recall of 100% for the top ranked classes. Thus, the accuracy of the model is an issue in this type of study. To minimise this threat to validity, we manually validated the GCs that were discussed in the paper.

Both the prevention and correction mechanisms proposed are simplistic, but their purpose was to illustrate the usefulness of the exploration of the life-cycle of GCs. We believe that better prevention and correction mechanisms should be explored in future work.

A general problem to guide preventive maintenance was the lack of a taxonomy of changes that have a negative impact on software quality. We therefore used quantitative data (change metrics) instead of semantically meaningful transformations (like refactorings).

All data is available online³ for future replications.

VI. CONCLUSION

In this paper, following an epidemiological metaphor, we reported a study of the life cycle of GCs in two open-source systems, Xerces and Eclipse JDT, to determine how they came to be introduced, removed, and how they evolve. We noted that GCs are sometimes introduced by design as the best solution to a particular problem. Although they are not “good” code, these classes cannot be improved and remain relatively untouched from version to version. We found that changes, such as adding new responsibilities, can result in the degradation of GCs. The correction of a GC may also move the problem to a different class.

From this study, we showed how to develop prevention mechanisms, filters to determine whether projected changes are likely to transform a class into a GC and decrease its quality. We also formalised refactorings with their theoretical effects on GCs to suggest the most appropriate changes.

The generalisation of our study to other smells is briefly discussed and will be developed in future work. Future work will also include assessing more systems and discussing the proposed refactorings with their developers who apply them. We also plan to identify other kinds of good design practices, for example design patterns or Riel’s heuristics [4], to explain the existence of “good” GCs.

ACKNOWLEDGMENT

We gratefully thank Jean Vaucher for our many fruitful discussions and his valuable remarks. This work has been partly funded by the Canada Research Chair on Software Patterns and Patterns of Software.

REFERENCES

- [1] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1st ed. Addison-Wesley, June 1999.
- [2] L. D. Naouel Moha, Yann-Gaël Guéhéneuc and A.-F. L. Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. To appear.

³<http://www.ptidej.net/downloads/experiments/WCRE09b/>

- [3] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. John Wiley and Sons, March 1998. [Online]. Available: www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457
- [4] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [5] G. Antoniol and Y.-G. Guéhéneuc, “Feature identification: An epidemiological metaphor,” *Transactions on Software Engineering (TSE)*, vol. 32, no. 9, pp. 627–641, September 2006, 15 pages. [Online]. Available: <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/TSE06.doc.pdf>
- [6] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “A Bayesian Approach for the Detection of Code and Design Smells,” in *Proceedings of the 9th International Conference on Quality Software*, D.-H. Bae and B. Choi, Eds. IEEE Computer Society Press, August 2009.
- [7] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: using reading techniques to increase software quality,” in *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999, pp. 47–56.
- [8] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *Proceedings of the 20th International Conference on Software Maintenance*. IEEE Computer Society Press, 2004, pp. 350–359.
- [9] M. J. Munro, “Product metrics for automatic identification of “bad smell” design problems in java source-code,” in *Proceedings of the 11th International Software Metrics Symposium*, F. Lanubile and C. Seaman, Eds. IEEE Computer Society Press, September 2005. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.38>
- [10] E. H. Alikacem and H. Sahraoui, “Generic metric extraction framework,” in *Proceedings of the 16th International Workshop on Software Measurement and Metrik Kongress (IWSM/MetriKon)*, 2006, pp. 383–390.
- [11] K. Dhambri, H. Sahraoui, and P. Poulin, “Visual detection of design anomalies,” in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland*. IEEE Computer Society, April 2008, pp. 279–283.
- [12] F. Simon, F. Steinbrückner, and C. Lewerentz, “Metrics based refactoring,” in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR’01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 30.
- [13] G. Langelier, H. A. Sahraoui, and P. Poulin, “Visualization-based analysis of quality for large-scale software systems,” in *Proceedings of the 20th International Conference on Automated Software Engineering*, T. Ellman and A. Zisma, Eds. ACM Press, November 2005. [Online]. Available: <http://www.iro.umontreal.ca/labs/infographie/papers/Langelier-2005-VAQ/angelier-ase2005.pdf>
- [14] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. [Online]. Available: <http://www.springer.com/alert/urltracking.do?id=5907042>
- [15] E. van Emden and L. Moonen, “Java quality assurance by detecting code smells,” in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE’02)*. IEEE Computer Society Press, Oct. 2002. [Online]. Available: citeseer.ist.psu.edu/vanemden02java.html
- [16] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, “Does god class decomposition affect comprehensibility?” in *Proceedings of the 10th IASTED International Conference on Software Engineering*. Acta Press, 2006, pp. 346–355, 10 pages.
- [17] A. Lozano, M. Wermelinger, and B. Nuseibeh, “Assessing the impact of bad smells using historical information,” in *Proceedings of the 9th International Workshop on Principles of Software Evolution*, M. D. Penta and M. Lanza, Eds. ACM Press, September 2007, pp. 31–34. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1294948.1294957>
- [18] Foutse Khomh, M. D. Penta, and Y.-G. Guéhéneuc, “An exploratory study of the impact of code smells on software change-proneness,” in *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, G. Antoniol and A. Zaidman, Eds. IEEE Computer Society Press, October 2009, 10 pages. [Online]. Available: <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/WCRE09a.doc.pdf>
- [19] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” in *Proceedings of the International Conference on Software Engineering*, 2004, pp. 563–572.
- [20] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in *Proceedings of IEEE International Conference on Software Maintenance*, 1998, pp. 190–197.
- [21] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth, “Software evolution observations based on product release history,” in *Proceedings of IEEE International Conference on Software Maintenance*, 1997, pp. 160–.
- [22] H. Gall, M. Jazayeri, and J. Krajewski, “Cvs release history data for detecting logical couplings,” in *IWPSE*. Washington DC USA: IEEE Computer Society, 2003, pp. 13–23.
- [23] A. T. T. Ying, “Predicting source code changes by mining revision history,” Master’s thesis, University of British Columbia, October 2003.
- [24] D. M. German, “An empirical study of fine-grained software modifications,” *Journal of Empirical Software Engineering*, 2005.
- [25] Salah Bouktif, Y.-G. Guéhéneuc, and G. Antoniol, “Extracting change-patterns from CVS repositories,” in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, S. E. Sim and M. Di Penta, Eds. IEEE Computer Society Press, October 2006, pp. 221–230, 10 pages. [Online]. Available: <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/WCRE06.doc.pdf>

- [26] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *Transactions on Software Engineering*, vol. 28, no. 4, pp. 396–412, April 2002. [Online]. Available: <http://csdl2.computer.org/persagen/DLabsToc.jsp?resourcePath=/dl/trans/ts/&toc=comp/trans/ts/2002/04/e4toc.xml&DOI=10.1109/TSE.2002.995435>
- [27] J. Ratzinger, M. Fischer, and H. Gall, "EvoLens: Lens-view visualizations of evolution data," in *Proceedings of the 8th International Workshop on Principles of Software Evolution*, G. Canfora and S. Yamamoto, Eds. IEEE Computer Society Press, September 2005, pp. 103–112. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1572314
- [28] X. Xie, D. Poshypanyk, and A. Marcus, "Visualization of CVS repository information," in *Proceedings of the 13th Working Conference on Reverse Engineering*, S. E. Sim and M. D. Penta, Eds. IEEE Computer Society Press, October 2006.
- [29] T. Bakota, R. Ferenc, and T. Gyimóthy, "Clone smells in software evolution," in *Proceedings of the 23rd International Conference on Software Maintenance*, L. Tahvildari and G. Canfora, Eds. IEEE Computer Society Press, October 2007. [Online]. Available: <http://icsm07.ai.univ-paris8.fr/RPaccepted.htm>
- [30] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th conference on Object-oriented Programming, Systems, Languages, and Applications*, D. Lea, Ed. ACM Press, October 2000, pp. 166–177. [Online]. Available: <http://portal.acm.org/citation.cfm?id=353183>
- [31] A. Egyed, "Scalable consistency checking between diagrams—the VIEWINTEGRA approach," in *Proceedings of the 16th international conference on Automated Software Engineering*. IEEE Computer Society Press, November 2001, pp. 387–390. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=989835
- [32] Z. Xing and E. Stroulia, "Analyzing the evolutionary history of the logical design of object-oriented software," *Transactions on Software Engineering*, vol. 31, no. 10, pp. 850–868, October 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1542067
- [33] T. Mens, J. Fernandez-Ramil, and S. Degrandart, "The evolution of Eclipse," in *Proceedings of the 24th International Conference on Software Maintenance (ICSM'08)*. IEEE Computer Society Press, 2008, pp. 386–395.
- [34] M. Wermelinger, Y. Yu, and A. Lozano, "Design principles in architectural evolution: A case study," in *Proceedings of the 24th International Conference on Software Maintenance (ICSM'08)*. IEEE Computer Society Press, 2008, pp. 396–405.
- [35] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
- [36] M. T. P. T. M. C. Filippo Ricca, Massimiliano Di Penta and C. A. Visaggio, "Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks," in *Proceedings of the 30th international conference on Software engineering*, M. D. Wilhelm Schäfer and V. Gruhn, Eds. IEEE Computer Society Press, 2008, pp. 361–370.
- [37] J. B. Kruskal and M. Liberman, "The symmetric time-warping problem: From continuous to discrete," in *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Copmparison*, D. Sankoff and J. B. Kruskal, Eds. Addison-Wesley, 1983.
- [38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley, 1994.
- [39] S. Vaucher, H. Sahraoui, and J. Vaucher, "Discovering New Change Patterns in Object-Oriented Systems," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, Oct. 2008, pp. 37–41.
- [40] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Journal of Soviet Physics - Doklady*, vol. 10, no. 8, pp. 707–710, feb 1966.
- [41] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, 1st ed. Morgan Kaufmann, October 1999.
- [42] H. A. Sahraoui, R. Godin, and T. Miceli, "Can metrics help to bridge the gap between the improvement of oo design quality and its automation?" in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 154.