

Evolution Enriched Detection of God Classes

Daniel Rațiu* Radu Marinescu* Stéphane Ducasse** Tudor Gîrba**

* LOOSE Research Group, Research Institute e-Austria Timișoara

**Software Composition Group, University of Berne, Switzerland

{ratiud,radum}@cs.utt.ro
{ducasse,girba}@iam.unibe.ch

Abstract

Current approaches for automatic detection of design problems are not accurate enough because they analyze only a single version of a system and consequently they miss essential information as design problems appear and evolve over time. Our approach is to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. Our means is to define measurements which summarize how persistent the problem was and how much maintenance effort was spent on the suspected structure. We apply our approach on a large scale case study and show how it improves the accuracy of the detection of *God Classes* and additionally how it adds valuable semantical information about the evolution of flawed design structures.

Introduction. Various analysis approaches [4][8] have been developed to automatically detect where the object-oriented design problems are located, yet these approaches only make use of the information found in the last version of the system (*i.e.*, the version which is maintained). For example, we look for improper distribution of functionality among classes of a system without asking whether or not it raised maintenance problems in the past.

We argue that the evolution information of the problematic classes over their life-time can give useful information to system maintainers. We propose a new approach which enriches the design problems detection by combining the analysis based on a single version with the information related to the evolution of suspected flawed classes over time.

We show how we apply our approach when detecting one of the most well known design flaw: *God Class*. Marinescu [7] [8] detected this flaw by applying measurement-based rules on a single version of a system. He named these rules *detection strategies*. The result of a detection strategy is a list of *suspects*: design structures (*e.g.*, classes) which are suspected of being flawed. We enlarge the concept of detection strategies by taking into account the history of the suspects (*i.e.*, all versions of the suspects). We define history measurements which summarize the evolution of the suspects and combine the results with the classical detection strategies.

Detection Strategies. Detection strategies allow us to work with metrics on a more abstract level, which is conceptually much closer to our real intentions in using metrics (*e.g.*, for detecting design problems). The result of a detection strategy is a list of suspects (*i.e.*, suspected design structures). Using this mechanism it became possible [8] to quantify several design flaws described in the literature [9] [5]. We present below the detection strategy for *God Class*.

$$GodClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall C \in S' \\ (ATFD(C) > 40) \wedge ((WMC(C) > 75) \vee ((TCC < 0.2) \wedge (NOA > 20))) \end{array} \right. \quad (1)$$

God Class “refers to those classes that tend to centralize the intelligence of the system. An instance of a god-class performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes“ [8].

To detect a *God Class* we look for classes which use a lot of data from the classes around them while either being highly complex or having a large state and low cohesion between methods. The *God Class* detection strategy is a quantified measurement-based rule expressing the above description (see Equation 1). The metrics used for detecting *God Class* are taken from the literature: ATFD - Access to Foreign Data [8], WMC - Weighted Method Count [3], TCC - Tight Class Cohesion [1] and NOA - Number of Attributes [6].

History Measurements. We define a *history* to be a sequence of versions of the same kind of a particular entity (*e.g.*, class history, system history, etc.). By a version we understand a snapshot of an entity at a certain point in time (*e.g.*, class version, system version, etc.).

We refine the detection of design flaws by taking into consideration how *stable* the suspects were in the past and how long they have been suspected of being flawed. We name *persistent*¹ the entities which were suspects a large part of their life-time (*i.e.*, more than 95% of their life time). Thus we further introduce two measurements applied on the history of a suspect: *Stab* and *Pers*.

Measuring the Stability of Classes. We consider that a class was stable with respect to a measurement M between version $i - 1$ and version i if there was no change in the measurement. Thus we define $Stab_i$ measurement applied on a class history C with respect to a measurement M and related to version i (see Equation 2).

$$(i > 1)$$

$$Stab_i(C, M) = \begin{cases} 1, & M_i(C) - M_{i-1}(C) = 0 \\ 0, & M_i(C) - M_{i-1}(C) \neq 0 \end{cases} \quad (2)$$

Furthermore, as an overall indicator of stability, we define the $Stab_{1..n}$ measurement applied on a class history C as a fraction of the number of versions in which a class was changed over the total number of versions (see Equation 3).

$$(n > 2) \quad Stab_{1..n}(C, M) = \frac{\sum_{i=2}^n Stab_i(C, M)}{n - 1} \quad (3)$$

The classes functionality of the system is defined in their methods. In this paper, we consider that a class was changed if at least one method was added or removed. Thus, we will use *Stab* with respect to the number of methods of a class (NOM).

Measuring the Persistency of a Design Flaw. We define the *Pers* measurement of a flaw F for a class history C with n versions, *i.e.*, 1 being the oldest version and n being the most recent version (see Equation 4).

$$(i \geq 1)$$

$$Suspect_i(C, F) = \begin{cases} 1, & C_i \text{ is suspect of flaw } F \\ 0, & C_i \text{ is not suspect} \\ & \text{of flaw } F \end{cases}$$

$$(n > 2) \quad Pers_{1..n}(C, F) = \frac{\sum_{i=1}^n Suspect_i(C, F)}{n} \quad (4)$$

Measuring the persistency of a flaw we can find in what measure the birth of the flaw is related with the design stage or with the evolution of the system. We can interpret the persistent flaws in one of the following ways:

¹The adjective persistent is a bit overloaded. In this paper we use its first meaning: *existing for a long or longer than usual time or continuously*. Merriam-Webster Dictionary

1. The developers are conscious about these flaws and they could not avoid making them. This could happen because of particularities of the modeled system - the essential complexities [2] or the need to meet other quality characteristics (*e.g.*, efficiency).
2. The developers are not conscious about the flaws. The cause for this can be either the lack of expertise in object-oriented design or the trade-offs the programmers had to do due to external constraints (*e.g.*, time pressure).

The flaws which are not persistent are the result of the system's evolution. These situations are usually malign because they reveal the erosion of the initial design. They have two major causes:

1. The apparition of new (usually functional) requirements which forced the developers to modify the initial design to meet them.
2. The accumulation of accidental complexity [2] in certain areas of the system due to the changing requirements.

We can observe that, from the maintainers' point of view, we are interested mainly in the second aspects regarding to the apparition of flaws in the two cases presented above, as in the first situations we can not correct these design flaws because they are enforced by the modeled system.

Detection Strategies Enriched with Historical Information. We use the history measurements to enrich the *God Class* detection strategy. Due to the space limitation we define only *StableGodClass* and *PersistentGodClass* (see Equation 5 and Equation 6), the rest of the detection strategies used further being defined in a similar fashion. The only difference is that while stability is measured for a class in isolation, the instability for a class is measured relatively to the other classes within the system.

$$StableGC(S_{1..n}) = S' \left| \begin{array}{l} S' \subseteq GodClass(S_n), \\ \forall C \in S' \\ Stab(C) > 95\% \end{array} \right. \quad (5)$$

$$PersGC(S_{1..n}) = S' \left| \begin{array}{l} S' \subseteq GodClass(S_n), \\ \forall C \in S' \\ Pers(C, GodClass) > 95\% \end{array} \right. \quad (6)$$

God Classes and Stability. *God Classes* are big and complex classes which encapsulate a great amount of system's knowledge. They are known to be a source of maintainability problems [9]. However, not all *God Classes* raise problems for maintainers. The stable *God Classes* are a benign part of the *God Class* suspects because the system's evolution was not disturbed by their presence. For example, they could implement a complex yet very well delimited part of the system containing a strongly cohesive group of features (*e.g.*, an interface with a library).

On the other hand, the changes of a system are driven by changes in its features. Whenever a class implements more features it is more likely to be changed. *God Classes* with a low stability were modified many times during their life-time. Therefore, we can identify *God Classes* which raised maintenance problems during their life from the set of all *God Classes* identified within the system. The unstable *God Classes* are the malign sub-set of *God Class* suspects.

God Classes and Persistency. The persistent *God Class* are those classes which have been suspects for almost their entire life. Particularizing the reasons given above for persistent suspects in general, a class is usually born *God Class* because one of the following reasons:

1. It encapsulates some of the essential complexities of the modeled system. For example, it can address performance problems related to delegation or it can belong to a generated part of the system.

2. It is the result of a bad design because of the procedural way of regarding data and functionality, which emphasis a functional decomposition instead of a data centric one.

It is obvious that *God Classes* which are problematic belong only to the last category because in the first category the design problem can not be eliminated.

God Class suspects which are not persistent, obtained the *God Class* status during their lifetime. We argue that persistent *God Classes* are less dangerous than those which are not persistent. The former were designed to be large and important classes from the beginning and thus are not so dangerous. The later more likely occur due to the accumulation of accidental complexity resulted from the repeated changes of requirements and they degrade the structure of the system.

Experiment. We applied our approach on three case studies: two in-house projects and Jun², a large open source 3D-graphics framework written in Smalltalk. As experimental data we chose every 5th version starting from version 5 (the first public version) to version 200. In the first analyzed version there were 160 classes while in the last analyzed version there were 694 classes.

Next we present and analyze the concrete results of applying our approach on the Jun case-study. The history information allowed us to distinguish among the suspects provided by single-version detection strategies, the *harmful* and *harmless* ones. This distinction among suspects drives the structure of the entire section.

Harmless God Classes. The *God Classes* which are *persistent* and *stable* during their life are the most harmless ones. They lived in symbiosis with the system along its entire life and raised no maintainability problems in the past.

More than 20% of the *God Classes* were persistent and stable (5 out of 24). These classes in spite of their large size, did not harm the system's maintainability in the past so it is unlikely that they will harm it in the future. Almost all of these classes belong to particular domains which are weakly related with the main purpose of the application. We can observe this even by looking at their names. In Jun these classes belong to a special category named "Goodies". For example, one of these classes is *JunLispInterpreter* which is a class that implements a Lisp interpreter, one of the supplementary utilities of Jun. This is an example of a *GodClass* that models a matured utility part of the system.

Some of the *God Classes* were stable during their lifetime even if they were not persistent. The manual inspection revealed that some of these classes were born as skeletons of *God Class* classes and waited to be filled with functionality at later time. Another part of them was not detected to be persistent because of the noise which interfered during the analysis. We can consider these classes to be a less dangerous category of *God Classes*. An example of such classes is *JunOpenGLDisplayLight* which is a GUI class (*i.e.*, a descendant of *UI.ApplicationModel*), that suddenly grew in version 195.

Harmful God Classes. The *God Classes* which were both *not-persistent* and *unstable* are the most dangerous ones. Almost 30% of the *God Classes* are harmful (7 out of 24). They grew as a result of complexity accumulation over the system's evolution and presented a lot of maintainability problems in the past. The inspection of non-persistent unstable *God Classes* reveals that they all belong to the core of the modeled domain, which is in this case graphics. For example, one of these classes is *JunOpenGL3dCompoundObject* which implements the composition of more 3D objects. Its growth is the result of a continuous accumulation of complexity from version 75 to version 200.

Conclusions and Future Work. In this paper, we refined the original concept of *detection strategy*, by using historical information of the suspected flawed structures. Using this approach we showed how the detection of *God Classes* can become more accurate. We applied our approach on a large case study and presented the results and accompanied them by detailed discussions.

Our approach refines the characterisation of suspects, which lead to a twofold benefit:

²See <http://www.srainc.com/Jun/> for more information.

1. *Elimination of false positives* by filtering out, with the help of history information, the harmless suspects from those provided by a single-version detection strategy.
2. *Identification of most dangerous suspects* by using additional information on the evolution of initial suspects over their analyzed history.

In order to consolidate and refine the results obtained on the Jun case-study, the approach needs to be applied on further large-scale systems. We also intend to extend our investigations towards the usage of historical information for detecting other design flaws.

References

- [1] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, Apr. 1995.
- [2] F. P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, Apr. 1987.
- [3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.
- [5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [6] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.
- [7] R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of TOOLS*, pages 173–182, 2001.
- [8] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timișoara, 2002.
- [9] A. J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.