# Using History Information to Improve Design Flaws Detection

Daniel Raţiu
LOOSE Research Group
University of Timişoara
Romania
ratiud@cs.utt.ro

Stéphane Ducasse Tudor Gîrba
Software Composition Group
University of Berne
Switzerland
{ducasse, girba}@iam.unibe.ch

Radu Marinescu
LOOSE Research Group
University of Timişoara
Romania
radum@cs.utt.ro

## Abstract

[1] *As systems evolve and their structure decays, maintainers need accurate and automatic identification of the design problems. 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* Data Classes*, and additionally how it adds valuable semantical information about the evolution of flawed design structures.*

**Keywords:** software maintenance, software evolution, software metrics, quality assurance, object-oriented programming

## 1 Introduction

Maintenance effort is reported to be more than half of the overall development effort and most of the maintenance effort is spent in adapting and introducing new requirements, rather than in repairing errors [2][27]. One important source of maintainability problems is the accumulation of poor or improper design decisions. This is the reason why, during the past years, the issue of identifying and correcting design problems became an important concern for the object-oriented community [13][26][9].

Various analysis approaches [6][22] 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 two of the most well known design flaws: *Data Class* and *God Class*. Marinescu [21] [22] detected these flaws 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.

We applied our detection on three systems: two in-house projects, and a large open source framework. In this paper, we present and discuss the results we obtained on the latter case study.

After we present the metaphor of our approach, we briefly describe the concept of *detection strategy* and discuss the detection of *Data Classes* and *God Classes*. We define the history measurements needed to extend the detection strategies and discuss the way we use historical information in detection strategies. We then apply the new detection strategies on a large open source case study and discuss in detail the results. Before concluding and presenting the future work, we give an overview of the related work.

[1] Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering), 2004, pp. 223–232

## 2 The Evolution of Design Flaw Suspects

Design flaws (*e.g., God Class* or *Data Class*) are like human diseases – each of them evolve in a special way. Some diseases are hereditary, others are gained during the lifetime. The hereditary diseases are there since we were born. If the physicians are given a history of our health status over time they can give the diagnostic in a more precise way. Moreover there are diseases with which our organism is accustomed and thus represent no danger for our health and we don't even consider them to be diseases any more.

In a similar fashion we use the system's evolution to increase the accuracy of design flaws detection. We analyze the history of the suspects to see whether the flaw caused problems in the past. If in the past the flaw proved not to be harmful then it is less dangerous. For example, in many cases, the generated code needs no maintenance so the system which incorporates it can live a long and serene life no matter how the generated code appears in the sources (*e.g.,* large classes or unreadable code).

The design flaws evolve with the system they belong to. As systems get older their diseases are more and more prominent and need to be more and more taken into account.

## 3 Detection Strategies

A detection strategy is a generic mechanism for analyzing a source code model using metrics. It is defined by its author [22] as *the quantifiable expression of a rule, by which design fragments that are conformant to that rule can be detected in the source code.*

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 [22] to quantify several design flaws described in the literature [26] [13].

We present below the detection strategies for *Data Class* and *God Class*. Their presentation will also clarify the structure of a detection strategy.

### 3.1 *God Class* Detection Strategy

*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" [22].

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). We introduce below the measurements used:

- Access to Foreign Data (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [22].
- Weighted Method Count (WMC) is the sum of the statical complexity of all methods in a class [5]. We considered the McCabes cyclomatic complexity as a complexity measure [23].
- Tight Class Cohesion (TCC) is the relative number of directly connected methods [3].
- Number of Attributes (NOA) [20].

### 3.2 *Data Class* Detection Strategy

*Data Classes* "are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes" [13].

To detect a *Data Class* we look for classes which have a low complexity and high exposure to their internal state (*i.e.,* a lot of either accessor methods or public attributes). The *Data Class* detection strategy in Equation 2 uses the following measurements:

- Weight of a Class (WOC) is the number of non-accessor methods in the interface of the class divided by the total number of interface members [21].
- Number of Methods (NOM) [20].
- Weighted Method Count (WMC) [5].
- Number of Public Attributes (NOPA) is defined as the number of non-inherited attributes that belong to the interface of a class [21].
- Number of Accessor Methods (NOAM) is defined as the number of the non-inherited accessor-methods declared in the interface of a class [21].

### 3.3 Detection Strategy Discussion

As shown in the Equation 1 and Equation 2 the detection strategies are based on a skeleton of measurements and thresholds for each measurement (*e.g.,* $ATFD > 40$). While the measurements skeleton can be obtained by translating directly the available informal rules (*e.g.,* heuristics or bad smells), the particular sets of thresholds are mainly chosen based on the experience of the analyst. As this experience can differ from person to person the thresholds represent the weak point of the detection strategies.

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

$$DataClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall C \in S' \\ ((\frac{WMC(C)}{NOM(C)} < 1.1) \wedge (WOC(C) < 0.5)) \wedge ((NOPA(C) > 4) \vee (NOAM(C) > 4)) \end{array} \right. \quad (2)$$

Detection strategies are not a fully automated mechanism. The result of a detection strategy is a list of suspects which requires further human intervention to verify the flaw.

## 4 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.).

**Example.** In Figure 4 we use a simplified example of the Evolution Matrix [18] to display an example of a system history. A column of the matrix represents a version of the system (there are 4 versions displayed) and a line represents a class history. A square represents a version of a class.

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*[2] 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 $Pers$.

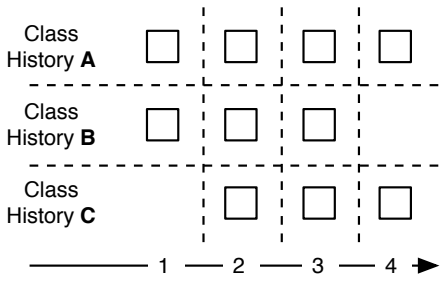


**Figure 1. Example of a system history displayed in an Evolution Matrix.**

[2]The 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

### 4.1 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 3).

$$(i > 1) \qquad Stab_i(C, M) = \begin{cases} 1, & M_i(C) - M_{i-1}(C) = 0 \\ 0, & M_i(C) - M_{i-1}(C) \neq 0 \end{cases} \quad (3)$$

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 4).

$$(n > 2) \qquad Stab_{1..n}(C, M) = \frac{\sum_{i=2}^{n} Stab_i(C, M)}{n-1} \quad (4)$$

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).

### 4.2 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 5).

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

$$(n > 2) \qquad Pers_{1..n}(C, F) = \frac{\sum_{i=1}^{n} Suspect_i(C, F)}{n} \quad (5)$$

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.

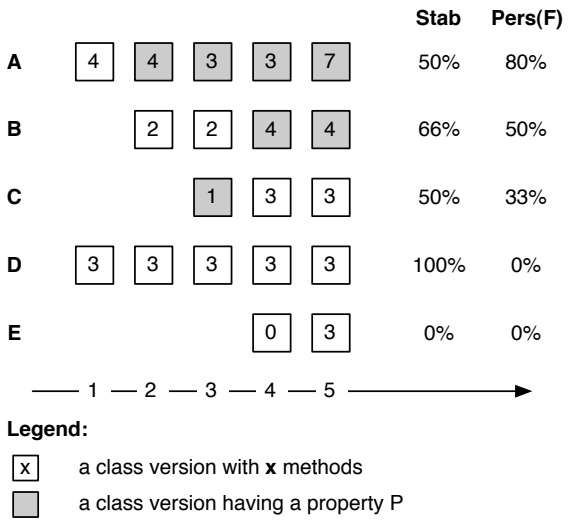


**Figure 2. Examples of the computation of $Stab$ and $Pers$.**

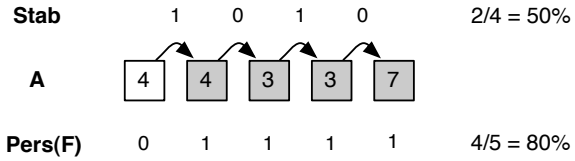


**Figure 3. Detailed computation of $Stab$ and $Pers$ for class history A.**

**Example.** Figure 2 presents 5 class histories and the results of the $Stab$ and $Pers$ measurements. Figure 3 shows in details how we obtain the values for the two measurements. We can interpret the persistent flaws in one of the following ways:

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 [4] 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 [4] 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.

## 5 Detection Strategies Enriched with Historical Information

We use the history measurements to enrich the *God Class* and *Data Class* detection strategies [3]. Due to space limitations of this paper, we only indicate the way we define *StableGodClass* and *PersistentGodClass* (see Equation 6 and Equation 7), 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.

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

$$PersGC(S_{1..n}) = S' \left| \begin{array}{l} S' \subseteq GodClass(S_n), \\ \forall C \in S' \\ Pers(C, GodClass) > 95\% \end{array} \right. \quad (7)$$

***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 [26]. However, not all *God Classes* raise problems for maintainers. The stable *God Classes* are a benign

[3] The detection strategies used in this paper are based on the work from [25]

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 lifetime. 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.

***Data Classes* and Stability.** *Data Classes* are lightweight classes which contain only few functionality. While *God Classes* carry on the work, *Data Classes* are only dumb data providers whose data is used from within other (possible many) classes. Modifications within *Data Classes* require a lot of work from programmers, as the principle of *locality of change* is violated. Thus, regarding the efforts implied by their change, programmers are less likely to change *Data Classes*. Based on this, we infer that the more relevant functionality a *Data Class* contains the higher are its chances to become the subject of a change. From this point of view, "classic" *Data Classes*, which are nothing else but dumb data carriers with a very light functionality, should be rather stable.

***Data Classes* and Persistency.** Persistent *Data Classes* represent those classes which were born with this disease. They break from the beginning of their life the fundamental rule of object-oriented programming which states that data and its associated functionality should stay together. Particularizing the reasons which could lead to persistent *Data Classes* we obtain that:

1. The class is used only as a grouping mechanism to put together some data. For example such a class can be used where is necessary to transmit some unrelated data through the system.

2. There was a design problem as *Data Classes* do not use any of the mechanisms specific to object-oriented programming (*i.e.*, encapsulation, dynamic binding or polymorphism). The data of these classes belong together but the corresponding functionality is somewhere else.

*Data Classes* which are not persistent are classes which got to be *Data Class* during their life. A class can become a *Data Class* either by requirements change in the direction of functionality removal or by refactorings. Functionality removal while keeping the data is unlikely to happen. Furthermore by properly applying the refactorings as defined in [13] we can not get classes with related data but no functionality (*i.e.*, the malign set of *Data Classes*). The only *Data Classes* we are likely get are those which belong to the harmless category where the class is used only as a modularization mechanism for moving easily unrelated data.

# 6 Experiment

## 6.1 Experimental Setup

We applied our approach on three case studies: two in-house projects and Jun[4], a large open source 3D-graphics framework written in Smalltalk. The Jun project, started in 1996, is still under development and we have access to more than 500 of its versions. 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. There were 814 different classes which were present in the system over this part of its history. Within these classes 2397 methods were added or removed through the analyzed history.

[4] See http://www.srainc.com/Jun/ for more information.

| Suspect | P | NP | S | U |
|---|---|---|---|---|
| JunHistogramModel | * | | * | |
| JunLispCons | * | | * | |
| JunLispInterpreter | * | | * | |
| JunSourceCodeDifference | * | | * | |
| JunSourceCodeDifferenceBrowser | * | | * | |
| JunChartAbstract | * | | | * |
| JunOpenGLGraph | * | | | * |
| JunOpenGLDisplayModel | * | | | * |
| JunOpenGLShowModel | * | | | * |
| JunUNION | * | | | * |
| JunImageProcessor_class | * | | | |
| JunOpenGLRotationModel | * | | | |
| JunUniFileModel | * | | | |
| JunVoronoi2dProcessor | * | | | |
| JunMessageSpy | | * | * | |
| JunOpenGLDisplayLight | | * | * | |
| Jun3dLine | | * | | * |
| JunBody_class | | * | | * |
| JunEdge | | * | | * |
| JunLoop | | * | | * |
| JunOpenGL3dCompoundObject | | * | | * |
| JunOpenGL3dObject_class | | * | | * |
| JunPolygon | | * | | * |
| JunBody | | * | | |
| 24 | 14 | 10 | 7 | 12 |

**Table 1.** *God Classes* **detected in version 200 of Jun case-study and their history-related properties**

Legend: P – Persistent; NP – Not Persistent;
S – Stable; U – Unstable

We first applied the detection strategies and then the suspects were both manually inspected at the source-code level and looked-up in Jun's user manual. Based on the manual inspection we determined which suspects were false positives, however we did not have access to any internal expertise.

### 6.2 Results Analysis

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.

The analysis shows how, by adding information related to the evolution of classes, the accuracy of the detection results was improved on this case-study, both for *God Classes* and *Data Classes*. Additionally, it shows that the history of the system adds valuable semantical information about the evolution of flawed design structures.

**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.

When taking a closer look at the Table 1 which present the *God Class* suspects we observe that more than 20% of them are 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".

- *JunLispInterpreter* 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.

- *JunSourceCodeDifferenceBrowser* is used to support the evolution of Jun. It belongs to the effort of the developers to support the evolution of the library itself.

Continuing to look at the Table 1 we notice that 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*.

- *JunOpenGLDisplayLight* is a GUI class (*i.e.,* a descendant of UI.ApplicationModel), which suddenly grew in version 195.

- *JunMessageSpy* is a helper class which is used for displaying profiling information for messages. It also belongs to the 'Goodies' category. This class was detected as not persistent only because of some noises which interfered in the measurements.

**Harmful *God Classes*.** The *God Classes* which were both *not-persistent* and *unstable* are the most dangerous ones. Looking at the Table 1 we can easily see that 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.

- *JunOpenGL3dCompoundObject* implements the composition of more 3D objects. Its growth is the result of a continuous accumulation of complexity from version 75 to version 200.

- *JunBody* models an element which represents a single solid. Between version 100 and 150 its complexity grew by a factor of 3.

- *JunEdge* element represents a section where 2 faces meet. It had a continuous growth of WMC complexity between versions 10 and 155.

| Suspect | P | NP | S | U |
|---|---|---|---|---|
| JunParameter | * | | * | |
| JunPenTransformation | * | | * | |
| JunVoronoi2dTriangle | * | | * | |
| JunVrmlTexture2Node | * | | * | |
| Jun3dTransformation | * | | | |
| JunVrmlIndexedFaceSetNode20 | * | | | |
| JunVrmlTransformNode | * | | | |
| JunHistoryNode | | * | | |
| JunVrmlMaterialNode | | * | | |
| JunVrmlNavigationInfoNode | | * | | |
| JunVrmlViewPointNode | | * | | |
| 11 | 7 | 4 | 4 | 0 |

**Table 2.** *Data Classes* **detected in version 200 of Jun case-study and their history-related properties**

Legend: P – Persistent; NP – Not Persistent;
S – Stable; U – Unstable

**Harmless *Data Classes*.** We consider non persistent *Data Classes* to be less dangerous. The manual inspection revealed that the accessor methods of these classes are not used from exterior classes. They are used only as local wrappers for their instance variables or as instance initializers from their meta-classes.

| Suspects | Total | False Positives |
|---|---|---|
| Classic *God Class* Suspects | 24 | |
| Harmless *God Classes* | 5 | 0 |
| Harmful *God Classes* | 7 | 0 |
| Not Classified *God Classes* | 12 | |
| | | |
| Classic *Data Class* Suspects | 11 | |
| Harmful *Data Classes* | 7 | 4 |
| Harmless *Data Classes* | 4 | 0 |
| Not Classified *Data Classes* | 0 | |

**Table 3. Summary of the results**

**Harmful *Data Classes*** The most dangerous *Data Classes* are those which were designed like that from the beginning. The manual inspection revealed that 3 out of 7 of these *Data Classes* (*i.e.,* almost 50% of persistent *Data Classes*) are used from within other classes.

- *JunVoronoi2dTriangle* has its accessors used from within JunVoronoi2dProcessor which is a persistent *God Class*.

- *Jun3dTransformation* which is used from JunOpenGL3dObject_class which is a not persistent *God Class*.

- *JunParameter* is used from within JunParameterModel.

The other 4 persistent *Data Class* suspects proved to be false positives as the manual inspection of the suspects proved that their accessors are used only from within their class or for initialization from their meta-class.

### 6.3 Results Summary

Table 3 summarises the results of the case-study analysis, showing explicitly the accuracy improvement compared with the single-version detection strategies.

From the total of 24 classes which were suspect of being *God Classes* using the classic detection strategies, there were 5 of them detected to be harmless and 7 of them detected to be harmful. After the manual inspection, no false positives were found. There were 12 suspects being *God Classes* which were not classified as being either harmful or harmless. This category of suspects require manual inspection as the time information could not improve the classic detection.

The classic *Data Class* detection strategies detected 11 suspects. 7 of these were detected as being harmful and the other 4 detected to be harmless. After the manual inspection, we found 4 false positives out of 7 of the *Data Classes* detected as being harmful.

## 7 Variation Points of the Approach

During the experiments we had to choose among many different possibilities to deal with the time-information. We consider necessary a final discussion centered on the possible variations of this work. Thus, the purpose of this section is to put the chosen approach in a larger context driving in the same time the directions for future work.

**Variation of the $Stab$ measurement.** We consider a class unstable if *at least* one method was added or removed regardless of the kind of method. Therefore, we ignored the changes produced at a lower level (*e.g.*, bug-fixes) and do not distinguish the quality of changes. A possible solution here would be to sum the number of changes in successive versions and complement the $Stab$ measurement with size information.

Also, $Stab$ measurement considers all changes equal, regardless of the period of time in which they appeared. Another possible extension here is to consider just the latest changes by weighting each change with the distance from the last version analyzed.

**Variations in the number of analyzed versions.** The more number of versions we consider in our analysis, the more we favor the capture of the small changes in the $Stab$ measurement.

**Variations in the starting point of the analyzed history.** We found out that persistent and stable *God Classes* are harmless because they usually implement a collateral standalone feature. These classes are part of the initial design, as persistency means the existence of the flaw almost since the very beginning of the class life. Therefore, if we consider the analyzed period to be from the middle of their actual life, we cannot detect whether they were really persistent. Therefore, for persistency we need to consider the history from the beginning of the system's history.

**Variations of threshold values.** The thresholds represent the weakest point of the detection strategies because they are established mainly using the human experience. In this work and in [25] the time information is used to supplement the lack of precision for particular sets of thresholds.

## 8 Related Work

During the past years, different approaches have been developed to address the problem of detecting design flaws. Ciupke employed queries usually implemented in Prolog to detect "critical design fragments" [6]. Tourwe *et al.* also explored the use of logic programming to detect design flaws. vanEmden *et al.* detected bad smells by looking at code patterns [28]. As mentioned earlier, Marinescu defined detection strategies which are measurement-based rules aimed to detect design problems [22] [21]. Visualization techniques have also been used to understand design [7].

These approaches have been applied on one single version alone, therefore missing useful information related to the history of the system. Our approach is to complement the detection of design fragments with history information of the suspected flawed structure.

Pioneering work to measure systems history has been carried out by Lehmann who analyzed the evolution of the IBM OS/360 system [19]. In [24], the authors explored the use of measurement when analyzing the software evolution. Demeyer *et al.* analyzed the history of three systems to detect "refactorings via change metrics" [8]. Krajewski defined a methodology [17] for analyzing the history of software systems.

These approaches used measurements as an indicator of changes from one version to another. We consider history to be a first class entity and define history measurements which summarize the entire evolution of that entity.

Visualization techniques in combination with measurements were also used to understand history information [18] [16] [15]. As opposed to visualization, our approach is more automatic and it reduces the scope of analysis, yet we believe that the two approaches are complementary.

In [12] the authors used information outside the code history and looked for feature tracking, thus gaining semantical information for understanding. Our approach differs as we only make use of the code history. Another approach was developed by Gall *et al.* to detect hidden dependencies between modules [14], but they considered the module as unit of analysis while we base our analysis on detail information from the code.

Jun has been the target of evolution research [1], however the focus of the article was to use the history information to describe the development process and the lessons learnt from developing Jun.

## 9 Implementation - Van and Moose

We made the experiments using *Van*, our history analysis tool which is based on the *Moose* [11] reengineering platform. Van is an implementation of the *HisMo* history meta model which is an extension of the *FAMIX* [10] language independent meta model.

In Figure 4 we present schematically the relationship between HisMo and FAMIX. HisMo recognizes the history as being a first-class entity which is formed by multiple versions, each version having a one-to-one relationship with a FAMIX entity.

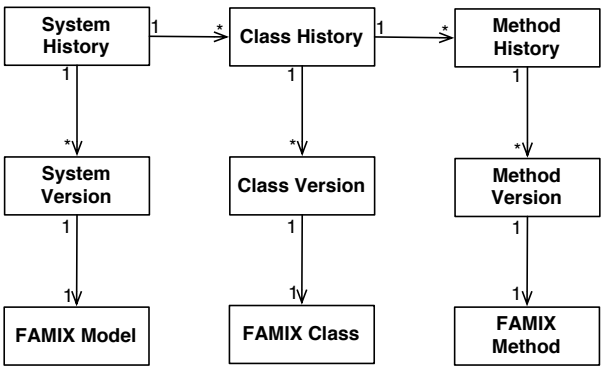


**Figure 4. The HisMo meta model is an extension of the FAMIX meta model (This a reduced schema of the meta model)**

## 10 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* and *Data 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:

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.

The $Stab$ measurement as defined previously, does not take into consideration the sizes of the change. We would like to investigate how this indicator of stability could be improved by considering the number of methods changed between successive versions. Other interesting investigation issues are: the impact on stability of other change measurements (*e.g.,* lines of code) and the detection of periodic appaearing and disappearing of flaws.

## Acknowledgments

Ducasse and Girba gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Tools and Techniques for Decomposing and Composing Software” (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and “RECAST: Evolution of Object-Oriented Applications” (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

Ratiu would like to thank CHOOSE, and Girba would like to thank European Science Foundation for the financial support.

## References

[1] A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A. Takashima, and Y. Yamamoto. A case study of the evolution of jun: an object-oriented open-source 3d multimedia library. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2001.

[2] K. Bennett and V. Rajlich. Software maintenance and evolution:a roadmap. In *ICSE - Future of SE Track*, pages 73–87, 2000.

[3] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, Apr. 1995.

[4] F. P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, Apr. 1987.

[5] 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.

[6] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.

[7] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE ’99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA ’2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000. 26 accepted papers on 142 (18%).

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[10] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 — the FAMOOS information exchange model. Technical report, University of Bern, Aug. 1999.

[11] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[12] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), to appear*, Nov. 2003.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[14] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[15] M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[16] M. Jazayeri, H. Gall, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM '99 Proceedings (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society, 1999.

[17] J. Krajewski. QCR - A methodology for software evolution analysis. Master's thesis, Information Systems Institute, Distributed Systems Group, Technical University of Vienna, Apr. 2003.

[18] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[19] M. M. Lehman and L. Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[20] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[21] R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of TOOLS*, pages 173–182, 2001.

[22] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timişoara, 2002.

[23] T. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[24] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2002.

[25] D. Ratiu. Time-based detection strategies. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, Sept. 2003.

[26] A. J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[27] I. Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[28] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, Oct. 2002.