Time-Based Detection Strategies

Author:
Daniel Raţiu

Diploma Thesis

Faculty of Automatics and Computer Science of the
"Politehnica" University of Timişoara

Timişoara,
September 2003

Advisors:
Dipl. ing. Tudor Gîrba
Dr. ing. Radu Marinescu

We must use time as a tool, not as a crutch.

John Fitzgerald Kennedy (1917 - 1963)

Acknowledgments

I am greatly indebted to Dr. Radu Marinescu with whom I had the fortune and priviledge to work during a large period of my student years. Discussing and working with him have been an invaluable source of life and scientific experience.

Special thanks to Tudor Gîrba, my diploma project mentor, with whom I had a lot of interesting and useful talks on the subjects presented in this work. Every time I was slipping away from the right track he succeeded to bring me back. I must also thank him for hosting me during the time I spent in Bern.

Many thanks to Pr. Dr. Stéphane Ducasse for making possible my collaboration with *Software Composition Group* from Bern and for supporting me during the month I staid there.

I want to thank my recent room mates, Andrei Firoiu and Gabriel Ersze, for their kindness, helping hand and for the funny moments we had together. I especially thank Andrei for reviewing a large part of this diploma.

The *Loose Research Group* members gave me useful feedback during our meetings. Special thanks to Bogdan Horje for the fresh ideas he gave and to Mircea Trifu with whom I spent many late-night working sessions during the faculty projects.

Last but not least I thank to my parents, to whom I owe what I am, who offered me full support and encouragements during my student-hood. Special thanks to my sweetheart Diana for her daily support and to her parents for the joyful weekends and holiday we had.

Timişoara, Daniel Raţiu
September 12, 2003

Contents

Chapter 1

Introduction

The explosion in size of the programs, together with costs reduction and tight deadlines imposed by software market, are some of the challenges of today's software industry. Object-oriented technology which was at first regarded as the magic solution to these problems proves to be of limited use. In order to satisfy their clients and preserve the business value of their already produced software, company programs must evolve. Thus, yesterday's builders of large object-oriented systems are now faced with the problems of modifying them.

In the assessment of software quality, the permanent audit of maintainability gained an important place. The current state of affairs shows that the maintainability problems have their roots in poor design of the systems. In order to make a connection between design problems and source code Fowler [Fow99] provides us a list of 'bad smells' which are informal descriptions of how design problems manifest at the source level. Even if these descriptions became a popular vocabulary among object-oriented developers, they lack the tool support necessary to make them automatically detectable.

The state of the art on quality assurance deals mostly with studying a certain version of a program. When analyzing a single version of a system we are faced with various problems like *noise* in the input data and the *inability* to detect quality flaws which manifest in time.

1.1 Goals and Approach

The **goal** of this work is to combine the information obtained from analysis of software systems evolution with the current ways of detecting object-oriented design flaws.

We divided this goal in two parts: the first is to *improve* the current way in which the design flaws are detected and the second is to *extend* the set of

detectable design flaws with new flaws that inherently require time-based information in order to be discovered.

In order to **approach** the aforementioned goal we define a new time-based mechanism for detecting design flaws, which combines current *detection strategies* [Mar02] with time information, the result being "Time Based Detection Strategies". In order to evaluate our new mechanism we have built a tool which uses these strategies as means for detecting a set of design flaws.

As we already mentioned we intend to *improve* and *extend* the detection of design flaws. Therefore, in this work we will introduce two distinct approaches which address each of the sides of our goal. We called these approaches: *Quality refined through evolution* and *Quality defined through evolution*.

Quality refined through evolution takes into consideration the evolution of the program entities (e.g. classes, methods) affected by a particular design problem over multiple versions of the same program. Taking time into account we aim to reduce the inherent noise which appears when input data is represented by only one version. We bring new information into the system by extending the input data to the whole history of the program.

Quality defined through evolution introduces a new approach for detection of the design flaws which inherently requires time information. Using this new approach, we are able to detect several quality flaws that were not detectable till now by state of the art approaches which use only a snapshot of system's evolution.

1.2 Diploma Roadmap

The body of this work contains the following chapters:

In Chapter 2 we will introduce the theoretical foundations which represent the basis of the current work. Here we will give the definition of quality, of its characteristics and we will focus on sub-characteristics of maintainability [91291]. After that we will introduce the software metrics used as a tool for assessing the software quality. Following this, we will present some of the fundamental views on software evolution like "Laws of evolution" [Leh96] and "Software Aging" [Par94] and we will end with a presentation of the costs of maintenance in the software process.

Chapter 3 will present the current state of affairs in the two main fields of this diploma: *software quality* and *software evolution*.
In the context of quality we will present a new set of metrics developed in [Mar99] and the *detection strategies* concept [Mar02]. Following this, we will survey the modalities of reusing the design knowledge during the software development process instantiated through good habits to be followed and pitfalls

to be avoided. In the context of software evolution we will present the *Evolution Matrix* [Lan01a] and the *History Meta-Model* used as tools for dealing with evolution. We will conclude this chapter by presenting the roadmap in software evolution as described in [Ben01].

In Chapter 4 we present the concept of *"Time based detection strategy"* and how it can be used to extend the classic detection strategies and increase their accuracy. This chapter has two sections which correspond to the directions of the current work: *Quality defined through evolution* and *Quality refined through evolution*. In the first section we propose a way to enhance the current detection process with time information in order to improve its accuracy. The second section presents a new approach to find the design problems inherently related to time.

Chapter 5 presents the results of the experiments performed by using V-Soda[1] the tool we developed in the context of this diploma thesis. Out of the empirical results we drive several conclusions for each set of time-based detection strategies. In this chapter we investigate mainly the accuracy of our approach and secondly its scalability.

In Chapter 6 we will draw some of the general conclusions along with perspectives for future extensions of the current work.

The appendices which follow are summarized below:

Appendix A presents the implementation specific issues like the *particularities of Smalltalk* which affected our implementation and experiments, the *particularities of Moose* [DLT00], the implementation of the new detection strategy operators in addition to the existing ones and the algorithm used for detecting the 'entities that change together'.

In Appendix B we give the definition of all metrics used during this work. We will also discuss in more detail the difference in intent and implementation of related metrics.

Appendix C is intended to be a high-level and goal-driven description of *V-Soda*, the tool used for the experiments done in this work.

[1] *Soda* - Smalltalk Object-oriented Design Analyzer - is a module in a larger system for Versions Analysis - *Van*

Chapter 2

Theoretical Foundations

The goal of this chapter is to introduce some of the theoretical foundations on which this diploma lays. The whole chapter is organized mainly along the main fields of the current work - *software quality* and *software evolution*. Each of these two sections has a whole part dedicated to them. We end this chapter with a presentation of the link between maintainability and the total cost of the software.

2.1 Software Quality

We will start this section by presenting some definitions of *quality*. The presentation continues with the enumeration of quality characteristics emphasizing the place of maintainability among them. At the end we focus only on maintainability by further decomposing it into sub-characteristics and by discussing its relations with the other characteristics.

2.1.1 Definitions of Quality

In the following we reproduce two definitions of 'quality' - the first is intended to be a more general one and is taken from Merriam-Webster Dictionary and the second is meant to be more formal and is taken from an ISO standard.

Definition (Quality): [Merriam-Webster Dictionary] [. . .] **2** a : degree of excellence [. . .] b : superiority in kind [. . .]

Definition (Quality): The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs. [91291]

The first definition is stronger as it implies that quality means excellence and superiority. The second definition is more pragmatic as it links the quality with the ability to satisfy someone's needs without the need of excellence. Out of

the above definitions we observe that 'quality' can have different meanings for different persons as it depends on the receiver thus arising the need for dividing the quality into finer grained characteristics as presented in the following.

2.1.2 Software Quality Characteristics

In the following we will present the quality characteristics as defined by the *ISO 9126* standard.

> **Functionality:** A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
>
> **Reliability:** A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
>
> **Usability:** A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.
>
> **Efficiency:** A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
>
> **Maintainability:** A set of attributes that bear on the effort needed to make specified modifications.
>
> **Portability:** A set of attributes that bear on the ability of software to be transferred from one environment to another.

We can easily notice that many of the above characteristics cannot be fully met in the same time. For example the efficiency and portability - the more portable a software the more environment independent it should be while the efficiency is directly linked with the specific resources offered by a specific environment. We will see below in Paragraph 2.1.3 how maintainability can be in conflict with other characteristics.

2.1.3 Maintainability

The sub-characteristics of maintainability The ISO9126 standard [91291] also defines a set of sub-characteristics of each of the aforementioned main characteristics. In the following we will present only the sub-characteristics of *maintainability*.

> **Analysability:** Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

Changeability: Attributes of software that bear on the effort needed for modification, fault removal or for environmental change.

Stability: Attributes of software that bear on the risk of unexpected effect of modifications.

Testability: Attributes of software that bear on the effort needed for validating the modified software.

Maintainability vs. other characteristics Here we give some examples about how can maintainability sub-characteristics come in conflict with the other characteristics of quality.

Changeability vs. Portability: Portability depends on the ability to be transferred from one system to another. This directly affects the changeability as the more portable the software is the more effort must be invested to implement certain kinds of changes which directly interact with its operational environment.

Analysability vs. Efficiency: The more analyzable a software is, the more additional information it must contain and thus the farther it is from the underlying hardware so less efficient it becomes.

Changeability vs. Usability: Whenever a system is designed to be easily changed some additional complexity is introduced and this can be reflected negatively in the usability of the existing set of features.

Different views on maintainability There are other classifications of the software characteristics and sub-characteristics. Some of them will be given in the following together with the prepositions they depend on.

- Depending on **when** they are observed: *'Quality Attributes Discernible at Runtime'* vs. *'Quality Attributes Not Discernible at Runtime'*
- Depending on **who** observes them: *User's View* vs. *Developer's View* vs. *Manager's View* [91291] or *'External Factors'* vs. *'Internal Factors'* [Mey97].
- Depending on **what** is observed: *'Functional Attributes'* vs. *Non-functional Attributes*

Maintainability can thus be framed as a *non-functional*, *internal* and *non-runtime* attribute which mainly interests the *developer*.

2.2 Software Metrics

Throughout the research on software quality assurance a large number of software metrics have been defined spanning from metrics applied mainly on procedural code to metrics specific for Object-Oriented software systems. The advent of the metrics was rendered to the need of measuring the software. Below we present some of the most well known used metrics from the two categories.

2.2.1 Procedural metrics

Procedural metrics measure the characteristics of the software at the level of a procedure. They don't take into consideration the overhead introduced by the data. From this category the most important metrics are referring to the system complexity (e.g. McCabe's cyclomatic number) and size (e.g. lines of code).

Cyclomatic Complexity

This is one of the oldest and most known complexity metric defined by McCabe in [McC76]. The main idea of this metric is that the complexity of a program directly depends on the number of paths through it. For each program there is a graph (G) associated with it called 'program control graph'. Each node of the graph corresponds to a piece of code which is executed sequentially and each arc corresponds to branches taken in the program. At end, a supplementary arc must be added to the graph which connects the node which represent the exit point of the procedure with the entry of it. In this way we obtained a strongly connected graph. The cyclomatic number of this graph can be computed as shown below:

$$v(G) = e - n + 2$$

Where `e` and `n` represents the number of edges and the number of nodes of the graph

We reproduce below the set of properties of the cyclomatic complexity as they were originally presented in [McC76].

1. v(G)≥ l.
2. v(G) is the maximum number of linearly independent paths in G; it is the size of a basis set.
3. Inserting or deleting functional statements to G does not affect v(G).
4. G has only one path if and only if v(G) = I.
5. Inserting a new edge in G increases v(G) by unity.
6. v(G) depends only on the decision structure of G.

For the drawbacks related to this metric used on object-oriented code in general and on Smalltalk in particular please refer to Paragraph B.1.3.

Lines of Code

This is one of the oldest measures of software. It started as a measure of the programs written in fixed-format assembly languages. As the syntax of assembly evolved and no fixed-format was compulsory any more this measure became more problematic as one could write more than one statement per line thus the idea of a 'line of a program' lost from its initial importance. With the advent of high-level languages where the expressibility increased and one could express the same thins in many different ways, the LOC metric lost again from its importance. [HS95, p.87]

Definition: A line of code is any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. [CDS86]

The greatest advantage of this metric is its simplicity. On the other hand this metric doesn't take in consideration the complexity of a line of code and thus it is of limited use.

For a comparison between LOC (Number of Lines of Code) and NOM (Number of Methods) in the context of measuring the stability metrics please refer to Section B.2.2.

2.2.2 Object Oriented Metrics

In the field of Object-Oriented systems other categories of metrics which take into consideration the particularities of this field appeared. In the following we present some of the categories which are specific to the Object-Oriented systems and which we used the most in our work. All categories definitions and metrics presented as examples with their definitions and viewpoints were taken as they were originally presented in [Chi94]. After presenting each metric example with its viewpoints we will map the viewpoints on the sub-characteristics of maintainability presented above as a set of comments.

Complexity Metrics

Their role is to express the complexity of a class.

Example: WMC - Weighted methods per class [Chi94]

Definition: Consider a class C_1, with methods $M_1, \dots M_n$ that are defined in the class. Let $c_1, \dots c_n$ be the complexity of methods. Then:

$$WMC = \sum_{i=1}^{n} c_i$$

[Chi94]

The complexity is not defined in the above definition. However the most common measure for the above complexity is the *cyclomatic complexity*. (in this work we used the 'cyclomatic complexity' as a particular measure for the complexity too)

Viewpoints [Chi94]:

- The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
- The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.
- Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse.

Comments: It is obvious that this metric is connected with the *analysability*, *changeability* and *testability* of the software under scrutiny. The more complex a class is the less analyzable, testable and changeable it is. This metric can give predictions about other characteristics of the quality too. (e.g. *reusability*).

Coupling Metrics

"Two classes are coupled when methods declared in one class use methods or instance variables of the other class." [Chi94]

Example: CBO - Coupling between object classes [Chi94]

Definition: CBO for a class is the count of the number of other classes to which it is coupled. [Chi94]

Viewpoints [Chi94]:

- Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.
- In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.
- A measure of coupling is useful to determine how complex the testing of various parts of the design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Comments: This metric has a strong connection to *maintainability* due to its links with the modularity. The more modular a system is the easier it is to be analyzed, changed and tested.

Cohesion Metrics

"If an object class has different methods performing different operations on the same set of instance variables, the class is cohesive. This view of cohesion is centered on data that is encapsulated within an object and how methods interact with data."[Chi94]

Example: LCOM - Lack of Cohesion in Methods [Chi94]

Definition: Consider a Class C_1 with n methods $M_1, M_2, \ldots, M_n$. Let $\{I_j\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1, \ldots, I_n\}$. Let P=$\{(I_i, I_j) \mid I_i \bigcap I_j = \emptyset\}$ and Q=$\{(I_i, I_j) \mid I_i \bigcap I_j \neq \emptyset\}$. If all n sets $\{I_1\}, \ldots, \{I_n\}$ are $\emptyset$ then let $P = \emptyset$.

$$LCOM = \begin{cases} \mid P \mid - \mid Q \mid & \text{if } \mid P \mid \geq \mid Q \mid \\ 0 & \text{otherwise} \end{cases}$$

[Chi94]

Viewpoints [Chi94]:

- Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
- Lack of cohesion implies classes should probably be split into two or more sub-classes
- Any measure of disparateness of methods helps identify flaws in the design of classes.
- Low cohesion increase complexity, thereby increasing the likelihood of errors during the development process.

Comments: As shown above there is a strong link between LCOM and the encapsulation. Encapsulation is directly connected with the *stability*.

Inheritance Related Metrics

These metrics measure different properties of the inheritance hierarchy.

Example: DIT - Depth of Inheritance Tree [Chi94]

Definition: Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length

from the node to the root of the tree. [Chi94]

Viewpoints [Chi94]:

- The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Comments: The maintainability is linked with the complexity measurement aspect of this metric. The more complex a class is the less analyzable it will be.

There are metrics suites defined for predicting the software maintainability [Li,93]

2.3 Detection Strategies

The software metrics are too fine grained means to achieve the identification of quality problems. Thus, a *higher level*, *goal driven* way of combining metrics with logical operators and thresholds, the *detection strategy* concept, was introduced in [Mar02].

Definition: A detection strategy is the quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code.

The meanings of 'Detection Strategies': The expression *detection strategies* has some different meanings which are presented in the following."Detection strategies" can denote: a *concept* - it was presented in the above definition (e.g. A 'detection strategy' is made of composition operators and filtering operators), a general *tool* (e.g. We used the 'detection strategies' for identifying the design flaws) and a *library* - a set of concrete instances of the concept (e.g. Quality refined through evolution extends the currently defined 'detection strategies')

2.4 Software Evolution

2.4.1 Definitions of Evolution

Definition: Evolution [Webster's Dictionary] **[. . .] 2** a : a process of change in a certain direction : UNFOLDING b : the action or an instance of forming

and giving something off : EMISSION c (1) : a process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state : GROWTH (2) [...]

In the part (a) of the above definition, evolution is regarded as a 'change in a certain direction'. In the IT field the direction is given by the users of the programs. Over time the humans evolve and so do their needs which according to the part (c) of the definition become more and more complex. The software must evolve itself to satisfy the user's new requirements - thus it also becomes more and more complex.

2.4.2 Views of the Software Evolution

In the following we will present some views on the software evolution as they appear in [Leh96], [Par94] and [Mey97].

Laws Of Software Evolution

In the following are presented the laws of evolution which were established by Lehman [Leh96] as a result of his more than twenty years of work. After each evolution law we make some comments based on the explanations found in [Leh96]. The laws refer to the software system itself, to the relation of it with its users, developers and the producing organization.

Law 1 (Continuing Change): An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.

This law makes a parallel between the evolution of software and the evolution of living organisms. In order to survive, a software must keep up with the changing of both its operational environment and of the requirements of its users.

Law 2 (Increasing Complexity): As a program is evolved its complexity increases unless work is done to maintain or reduce it.

This is one of the sources of maintenance problems. As a software evolves, its complexity increases more and more. The problem is that the amount of "accidental complexity" [Bro87] increase in a higher rate than the size of the system.

Law 3 (Self Regulation): The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.

This law makes the link between the developers team and the organization were they activate. The target goals of the organization go beyond the goal of finishing the system under development. Organization's goals are controlled by the checks and balances it have to deal with and which influence the as positives

or negatives feedbacks the system under construction.

Law 4 (Conservation of Organizational Stability (invariant work rate)): The average effective global activity rate on an evolving system is invariant over the product life time.

The progress in the development of a software depends not only on the founds allocated but on other factors too such as the software's complexity, the knowledge of the developers, etc. If the management intends to increase suddenly the progress, the software will manifest an inertial behavior. If more new people became involved in the project they require time for training and for system understanding. Furthermore the communication overhead can lower down the progress rate.

Law 5 (Conservation of Familiarity): During the active life of an evolving program, the content of successive releases is statistically invariant.

The developers of a system must be familiar with the end goals of it. As new requirements arrive the end goal changes. The more new requirements are to be implemented for the subsequent version, the more new information must be acquired by the system's producers thus the more intensive their work will be. Statistical data showed that there are more critical levels of knowledge which "if exceeded trigger behavioral change".

Law 6 (Continuing Growth): Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

The first law relates the system with its environment. This law links the functional characteristics of the software with the user needs. The roots of this law come from the fact that not all the initial requirements can be accommodated. The requirements which were initially ignored can become real bottlenecks in the future.

Law 7 (Declining Quality): E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

The initial constraints in which the system is built will change over time, thus the need for the software changing itself. As the time passes another products become available, so the users develop new expectations form the product.

Law 8 (Feedback System): E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

The evolution is driven by the feedback coming from many sources, including

the users, organization itself and stakeholders. In order to successfully implement a change all the sources must be taken into account.

Software Aging

In the paper "Software Aging" [Par94] we are introduced the main problems related to the decaying of software. In this paper an analogy is made between the aging of humans and of software. In the following we summarize the causes, effects and the solutions related to the aging of software.

Causes of aging *Lack of movement* (i.e. if the software is not updated, the user will try to replace it) was identified to be the first cause of aging. The second cause is *Ignorant surgery* which represents the chains of modifications done by maintainers which do not fully understand the initial structure of the system.

The effects of aging Three main effects are associated to software aging: *inability to keep up*, *reduced performance* and *decreasing reliability*. As the complexity of system increases and more patches are applied the system inertia increases as well and the first effect of it is the inability to keep up with the changing requirements. While more and more patches are applied to the system its complexity increases artificially and the immediate result is a decrease in performance. A side effect of the modifying patches is the apparition of new bugs and thus the reliability is affected.

Reducing the costs of the software aging *Design for change* is the design for success and is seen as the first technique which must be applied in order to reduce the costs of aging. This approach implies that the developer should consider the possible future changes from the design time. However it is stressed that this technique is limited by the humans incapacity to predict the future. The second technique is related to the documentation process. In order to facilitate the system understanding in the future, the developers should *keep records* of their work. The author emphasizes on the fact that the documentation process is usually ad-hoc and neglected. The last solution proposed is related to the *second opinions* offered during the reviews.

Software geriatrics The first measure to be taken is *stopping the deterioration* of the architecture. This can be done by recreating the structure each time when a change is applied. In addition to this, the *retroactive documentation* replaces the old disparate pieces of documentation with a unitary view over the already existing system. *Retroactive incremental modularization* assumes that the structure is remodulated according to the new changes which the system will face. *Amputation* aims to lower down the system's size and complexity by getting rid of the parts that were modified too many times. The last and the most radical measure is **system restructuring** and is applied when the system gets out of control.

The Endless Need for Evolution

The fact that successful systems must continuously evolve is also emphasized in [Mey97, p. 106]: "Good systems have the detestable habit of giving their users plenty of ideas about all the other things they could do. [...] The new requirements evolve from the initial ones in a continuous way." Software evolution is an endless process driven by the evolution of humans needs. An implemented change triggers new changes in an continuous manner.

From Evolution to ... Revolution

As showed in [vGB01] currently there is no way to stop the erosion of the software design. Even if only the *optimal strategy* is used for integrating the new requirements the structure of the system decays. The software design erodes to a point where the evolution of that part of software is not possible any more. Then, the stakeholders need to take more radical decisions than changing only a small part of it (i.e. major restructuring or even system rewriting). In this context the software development moves from *evolution* to *revolution*. All modern software engineering techniques (e.g. programming languages, development methodologies, refactoring) can't do anything but to delay this major turning point.

2.5 The Costs of Maintenance

In this section we present the importance of the maintainability of a software in terms of costs.

The maintainability of a system directly affects its capacity of evolution. As shown in [Som01, p.10] the costs of the software evolution exceed the costs of development by a factor of 3 or 4. This implies that the maintenance plays a major role in the software life. The distribution of the maintenance effort during the evolution phase is shown in Figure 2.1 [Som01, p.606].

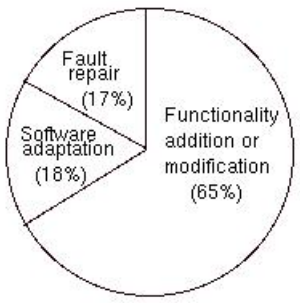

Figure 2.1: Maintenance effort distribution

We can observe that a relatively small percent of the maintenance effort is dedicated to bug fixes. The major part is dedicated to the addition of new functionality or modification of the existing one. In contrast to bug fixing where the changes are usually small, the addition of new functionality affects the structure of the existing system. The structure of the system is directly affected by the quality of the system's design. Thus a large part of the costs of the software is influenced by the design quality.

Chapter 3

State Of The Art

This chapter resumes a part of the work which has been done in the two fields which represent the main axes of this diploma. We will start by presenting the state of the art in software quality and we will continue with software evolution. After each section we make some comments related to the drawbacks of that approach and the improvements which can be done.

3.1 Software Quality

The definition of quality which was presented in the last chapter: "Totality of characteristics of an entity that bears on its ability to satisfy stated and implied needs." [ISO 8204 standard] The keyword here is *implied needs*, as the implied needs are not every time obvious. During the evolution of the system new needs come up while the initial implied ones are resolved. *By studying the evolution we can see how the implied needs were resolved and what are the new needs.*

3.1.1 Software Metrics

In this section we will present some of the software metrics defined more recent, emphasizing the improvements beside the classic metrics.

A Multi-Layered System of Metrics

In [Mar99] a new, more systematic approach for defining a system of metrics related to the 'Reuse of Ancestors' in a *Multi-Layered System of Metrics*.

Definition: Multi-Layered System of Metrics (MLSM) A multi-layered system of metrics (MLSM) is a set of interdependent metric definitions organized in layers, in which the definition of each layer excepting the lowest is based upon the definitions of the lower layers in the system. [Mar99]

When using this approach, three steps are required - the identification of layers, the definition of the layers and the definition of layer-specific metrics. For example, for the 'Reuse of Ancestors' system of metrics the following layers were identified:

- *Method-Ancestor level*. In fact the real reuse of ancestors takes place at this level, therefore this is the base (bottom) level of the system.
- *Class-Ancestor level*. When speaking about reuse we usually think in terms of classes reusing other classes.
- *Class level*. Often we need to assess the total reuse for a class (e.g. in order to evaluate its self-sufficiency). For this we need a higher-level definition of reuse.

[Mar99]

The improvements brought by this approach is that it facilitates the construction of more complex metrics because one can define the complex metrics based on others with a lower complexity.

The shortcomings of software metrics

There are two intrinsic problems with Software Metrics:

1. The first is that they represent a too fine grained approach to dealing with design problems. Even more complex metrics have been built, a metric taken into isolation can not offer more than a small view over the system analyzed. Thus a mechanism for uniformly combining the metrics into a more complex structures is required.

2. The second problem is that they lack of interpretation. The metrics are only mapping entities with meaningless numbers. A value of a number in one context (e.g. a programming language, a project) can have a completely different meaning beside another contexts. While the choice of which metric to be applied to a certain system compulsory depends on the goals of the measurement, like in Goal-Question-Metric model [BR88] the problem of the interpretation of numbers remains alive.

These problems can be successfully approached by using the *detection strategies* mechanism which is summarized below.

3.1.2 Detection Strategies

Detection Strategy, as a mechanism is a mean to overcome the two main drawbacks of metrics. They offer a higher level of expressibly due to the composition and filtering operators. The interpretation also is considered by giving the suspects which are conforming with a certain rule. In contrast to *software metrics*

whose usage represent a *bottom-up* approach, the *detection strategies* can be seen as a *top-down*, *goal-driven* approach to the design problems detection.

In [Mar02] a suite of detection strategies for object-oriented systems were defined. A part of our work will enrich with time information two of them - *DataClass* and *GodClass* - and give a completely new approach for *Shotgun-Surgery*.

In order to combine different properties *detection strategies* are based on 'Data Filtering' and 'Composition Operators'.

Definition (Data Filter) A data filter is a mechanism (a set operator) through which a subset of data is retained from an initial set of measurement results, based on the particular focus of the measurement. [Mar02]

The filters can be divided in two categories: marginal and interval filters according to Table 3.1 [Mar02].

Type of Data Filter	Limit Specifiers		Filter Examples
Marginal	Semantical	Relative	TopValues(10)
			BottomValues(5%)
		Absolute	HigherThan(20)
			LowerThan(6)
	Statistical		Box-Plot
Interval	Specification		Filter Example
	Composition of two marginal filters with semantical limit specifiers of opposite polarities		Between(20,30) := HigherThan(20)∧ LowerThan(30)

Table 3.1: Classification of Data Filters

The marginal filters can be further divided into *semantical* filters which contain a *threshold* value and a *direction* and into *statistical* filters where the threshold is determined by using statistical methods from the initial data set. The semantical filters contain two categories: *absolute semantical filters* (i.e. HigherThan, LowerThan) and *relative semantical filters* (i.e. TopValues, BottomValues).

Definition (Composition Operators) The operators used to compose a set of metrics into an articulated rule are called composition operators. [Mar02]

The composition operators can be seen from two viewpoints: *Logical Viewpoint* when operators represent the logical joints of the detection strategy expression and *Set Viewpoint* when computing the result of applying the detection

strategy to an input set.

In the following we will present examples of the three detection strategies expressions. The first two detection strategies we aim to enrich with time-based information in the first part of this work. For the third one we intend to provide an alternative expression using the evolution. All the metrics used in the below examples are described in Appendix B.

Example 1 - DataClass

```
DataClass := ((WOC, BottomValues(33%)) and (WOC, LowerThan(0.33)))
                and ((NOPA, HigherThan(5)) or (NOAM, HigherThan(5)))
```

Example 2 - GodClass

```
GodClass := ((ATFD, TopValues(20%)) and (ATFD, HigherThan(4)))
                and ((WMC, HigherThan(20)) or (TCC, LowerThan(0.33)))
```

Example 3 - ShotgunSurgery

```
ShotgunSurgery := ((CM, TopValues(20%))
                    and (CM, HigherThan(10))) and (CC, HigherThan(5))
```

Detection Strategies Drawbacks

The main problem with the detection strategies is represented by their *thresholds* as proved in the following. A detection strategy encapsulates the information in two ways - in its skeleton expression (i.e. the metrics and the operators) and in its thresholds. The skeleton expression can be quite easily defended by the definer of a certain detection strategy based on the relations between metrics with the specific flaw - the Goal-Question-Metric (GQM) [BR88] method can be applied to detection strategies as well. However, the thresholds are chosen based on the experience and intuition and vary from project to project.

Recently, a more systematic approach for finding thresholds was developed as a diploma thesis [Mih03]. In this approach the information fed into the system consists of two sets of entities: the first set consisting of entities with a specific flaw and the second set consisting of flawless entities. The thresholds are automatically tunned until the system identifies most of the flawed entities as flaws while ignores the other ones.

The first part of this current work aims to use the evolution as an additional source of information for finding the flaws. In order to achieve this, we find

correlations between the entities with a specific flaw and their evolution over time (See Section 4.1).

3.1.3 Patterns, Anti-Patterns and Code Smells

In order to capture the previous knowledge in software design and development various methods for transmitting the design expertise appeared.

Design patterns As defined in [GHJV95] a *design pattern* is a 4-tuple consisting of:

1. Pattern Name - "it is a handle we can use to describe a design problem, its solutions, and consequences in a word or two."
2. The Problem - "describes when to apply the pattern. It explains the problem and its context".
3. The solution - "describe the elements that make up the design, their relationships, responsibilities, and collaborations".
4. Consequences - "are the results and trade-offs of applying the pattern".

Thus, the design patterns offer solutions to recurrent problems which appear in a given context.

Anti-patterns Sometimes it is easier to describe a "negative solution", the source of a certain failure using an *Anti-Pattern* instead of describing a good practice. In the following we cite from a tutorial on anti-patterns [McC].

- Anti-Patterns are **Negative** Solutions that present more problems that they address
- Anti-Patterns are a natural extension to design patterns
- Anti-Patterns bridge the gap between architectural concepts and real-world implementations
- Understanding Anti-Patterns provides the knowledge to prevent or recover from them

Thus, Anti-patterns address the pitfalls of software development.

Code Smells The "code smell" metaphor was introduced in [Fow99] and is related to finer grained structures of a program. While *design patterns* express relations between classes of a system at the design level, the *code smells* relate with properties of a single class or method and are focused on the implementation level. Their main intention is to describe when and what to refactor.

Our work is in the context of finding the "code smells" by using the information about the system's evolution. Each time when we provide a detection detection procedure for a code smell we start with the definition of it. Thus, all

the definitions of the used code smells can be found in Chapter 4.

3.2 Software Evolution

In this chapter we present some of the later developments in the field of software evolution. We will start with a visualization method of the evolution of classes, continue by presenting a meta-model which support the evolution and end by presenting an envisioned road-map for the future of this domain.

3.2.1 Evolution Matrix

In the following paragraphs we will present the 'Evolution Matrix' as it was introduced in [Lan01a]. All the pictures are taken from the aforementioned paper.

On each row of the evolution matrix we have different versions of the same class. Each column represents a whole version of the system. For each class we have the characteristics represented with the help of a rectangle - See 3.1 (the picture was taken from [Lan01a]). The width of the rectangle represents the 'Number of instance variables' of the class and the height represents the 'Number of methods' of the class. By using the evolution matrix one can visualize

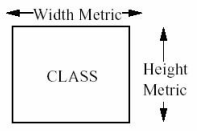

Figure 3.1: A graphical representation of classes using metrics

the characteristics at different levels of granularity. If he is mostly interested in an overview of the system, the evolution matrix can give information about the size of the system, the addition/removal of classes and growth and stagnation phases in the evolution. If we are interested in a finer grained analysis, we can concentrate on a single row of the evolution matrix. This way we can see how did the class evolve during its life-time. In this article, a categorization of classes based on the evolution matrix is proposed. In this work we will use this categorization in Section 5.3.5. In the following we take a deeper look at the class categories which were proposed in [Lan01a].

- Pulsars - the classes which increase and decrease in size repeatedly during their life-time. The pulsars can reveal the functionality addition, subtraction or the refactorings done inside classes.[Lan01a]
- Supernovas - are classes which "suddenly explode in size". The supernovas can reveal major refactorings, very light classes which can grow

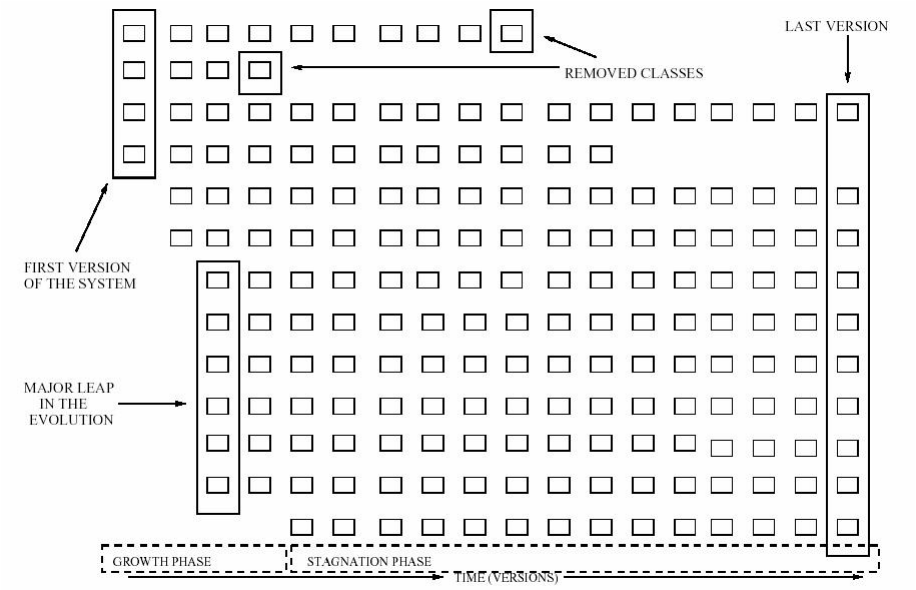

Figure 3.2: Evolution matrix at system level

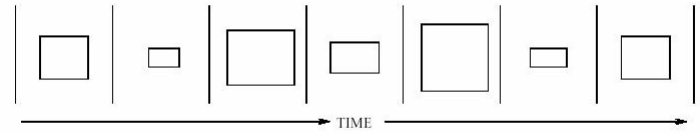

Figure 3.3: A graphical representation of a pulsar

very rapidly or classes which are in the system for a long time and are waiting to be filled with functionality [Lan01a]

- White Dwarfs - are classes which lost their functionality over their life and became very light-weighted [Lan01a].
- Red Giant - are huge classes which were permanent 'god class' classes. [Lan01a]
- Stagnant - are those classes which don't change over a long period of their life-time. They can reveal dead code, well designed classes or parts of the system where no work was done. [Lan01a]
- Dayfly - are classes which appear only for a version of the system's evolu-

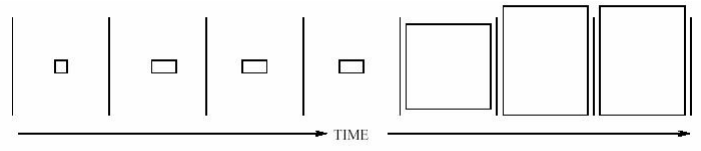

Figure 3.4: A graphical representation of a supernova

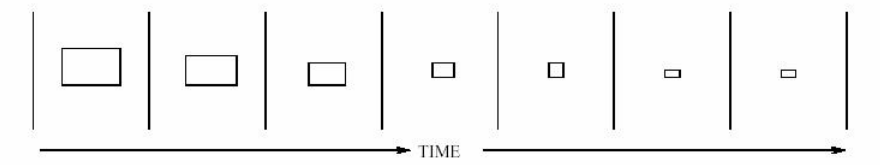

Figure 3.5: A graphical representation of a white dwarf

tion. [Lan01a]

- Persistent - are classes which have been inside the system since its birth.

Strengths and Weaknesses The major strength of software visualization in general is that this technique displays in an analogical way the information related to the software's characteristics. Instead of presenting numbers or lists of elements, this approach reveals the characteristics through the size, color and orientation of the entities under scrutiny. On the other hand, the major weakness of the visualization is its incapacity to cope with large amount of data - due to the limits of the displays and of humans visual field.

3.2.2 The History Meta-Model

In [Gir03] is addressed the problem of representing the time during the analysis of evolution. Entities which appear as elements in classic, version based meta-models such as *class* or *method* don't encapsulate time. Giving them time-based characteristics like those presented in [Lan01a] (e.g. pulsar class, supernova class) is inaccurate. A solution to this problem, proposed in [Gir03], is represented by "the concept of a *history* which consists of all the version based snapshots of a certain entity". As shown in the above picture this meta-model goes in parallel with classic ones but in addition it incorporates the time in its

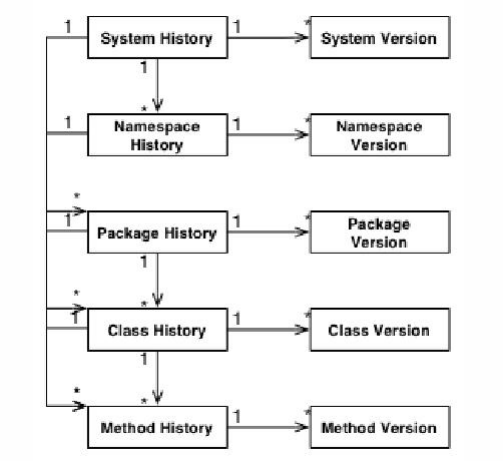

Figure 3.6: The History Meta-Model

component entities.

In this work we use *Hismo* as infrastructure for dealing with evolution.

3.2.3 A Road-Map of Software Evolution

Our survey on the 'state of the art' of software evolution ends with the presentation of the road-map of this domain [Ben01]. The purpose of this presentation is to make it possible to position the current work within the broad areas of research in software evolution.

The authors make a clear difference between the *maintenance* and *evolution* even if they are usually used interchangeably. The *maintenance* is "used to refer to generally post-delivery activities" and "*evolution* represents a particular phase in the staged model".

The aforementioned document takes two approaches in building the road-map: the first is to project the current state of the art into a newly defined model of software evolution called *stage model* and the second is to present a more far sighted vision on the development of software evolution in the future. As our aim is to position ourselves into the current work, we will concentrate only on the first part of the article.

A *Staged Model* for Software Maintenance

The authors define this new model based on the observation that the activities done during software evolution vary greatly in time. Thus, the maintenance seen

as a single post-delivery activity is not satisfying any more. The maintenance can be divided into five steps as presented in the following. [Ben01]

- *Initial development* During this stage the first version of the system is built. Even if this version may be incomplete "it already possesses the architecture that will persist thought the rest of the life of the program. [...] Another important outcome of the initial development is the knowledge that the programming team acquires: the knowledge of the application domain, user requirements, role of the application in the business process, solutions and algorithms, data formats, strengths and weaknesses of the program architecture, operating environment, etc."[Ben01]
- *Evolution* "takes place only when the initial development was successful. The goal is to adapt the application to the ever-changing user requirements and operating environment. [...] Both software architecture and software team knowledge make evolution possible. They allow the team to make substantial changes in the software without damaging the architectural integrity. Once one or the other aspect disappears, the program is no longer evoluable and enters the stage of servicing." [Ben01]
- *Servicing* During this phase "only small tactical changes (patches, code changes and wrappers) are possible. [...] For all practical reasons, the transition from evolution to servicing is irreversible." [Ben01]
- *Phase-out* "During phase-out, no more servicing is being undertaken, but the system still may be in production. The users must work around known deficiencies." [Ben01]
- *Close-down* "During close-down the software use is disconnected and the users are directed toward a replacement." [Ben01]

The above steps are also represented in Figure 3.7. A common problem is represented by the boundaries between stages. Each stage has its own specific problems to be addressed and most of them are presented in the following.

Research Topics

INITIAL DEVELOPMENT The main problem of this stage is to make the software easy to change in subsequent phases - i.e. "the cost of making the change is proportional to the size of the change, not to the size of the overall software system" [Ben01] The current research focuses on the problem of describing the architecture in a formal manner through *architecture description languages* (ADL), on ways of transmitting the good practices in design through *design patterns* and on developing "neutral" technologies (i.e. "the technology can be changed easily without consequential affects on the software design"). [Ben01]

EVOLUTION The research is centered on the following topics: software architecture, system understanding, features interaction, raising the abstraction level

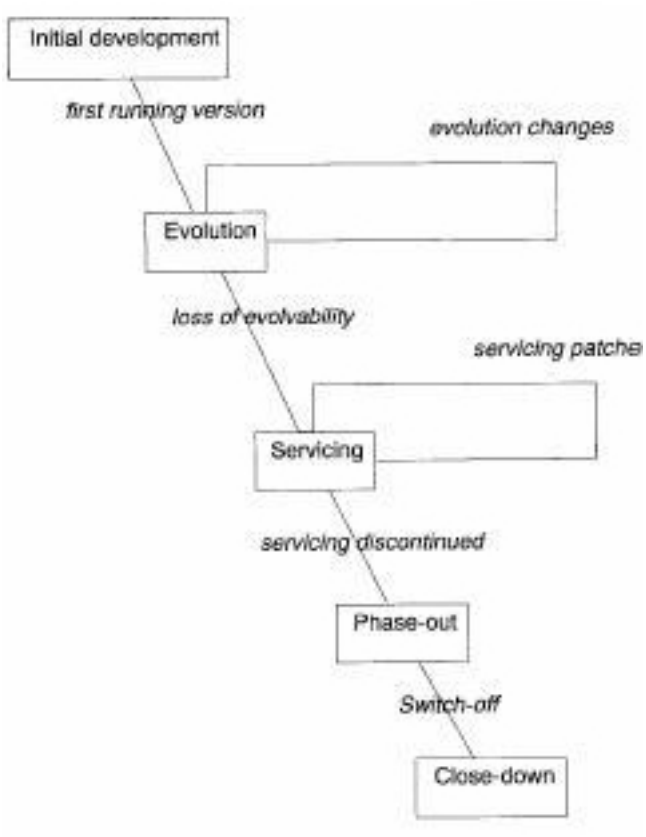

Figure 3.7: The 'Staged Model' for software maintenance

in which evolution is expressed and the business issues of the evolution stage. The research on *software architecture* aims at developing architectures that will evolve themselves and allow the unanticipated change. The research on *feature interaction* aims at developing methods for enhancing certain features of the system while maintaining the other features and the system invariants unchanged. The state-of-the-art work on raising the abstraction level in which the evolution is expressed follows the following directions: "raising the abstraction level of the language used", "separating the declarative issues like business objects from control issues","representing domain objects in domain analysis and domain specific languages", "partitioning architectures into independently evolving subsystems".[Ben01]

SERVICING The key research topics are centered on: program comprehension, automated tool support to improve the code, the problems related to the software centered for the mass market, and servicing the components. For program

comprehension the visualization tools have an important role to play. Further tools can be used for impact analysis, regression testing, concept identification and configuration management and software control. The improvement of the code can be done through migration from old to new languages, restructuring of code and data to eliminate the unnecessary complexity, documentation tools to manage comments, etc. [Ben01]

PHASE-OUT AND CLOSE-DOWN No technical research topics were identified as during these phases the maintenance of software is almost inexistent. [Ben01]

This diploma is concerned with quality assurance by identifying the flaws at the design level. Thus, our work is mainly placed in the *evolution* stage of the maintenance where is a great need for keeping the architectural integrity which implies a great interest in the quality of design. The taxonomy developed in Section 5.3.6 helps to express the evolution of design flaws in a more abstract level. Raising the level at which the restructuring are done above the level of source code (e.g. [DT03]) implies beforehand finding the flaws where the restructuring is triggered from. Even if in this work we concentrate on quality assurance, the results obtained through our developed methods can be used in other fields of maintenance as well (e.g. *Entities That Change Together* 4.2 can help in partitioning the system into independent subsystems)

Secondly, the results of our work can be useful in the *servicing* stage as the flaws which we detect can also be interesting both for implemented local changes (e.g. the impact of change can be given by ShotgunSurgery).

Chapter 4

Time Based Detection Strategies

The shortcomings of the currently defined detection strategies were presented in Section 3.1.2. This chapter will present the newly defined detection strategies which encapsulate time.

This section is divided in two main parts:

- Part I deals with the extensions added to the currently defined detection strategies. This section will be in the context of the *Quality refined through evolution* part of the thesis.
- Part II defines entirely new time-based detection strategies. This will belong to the frame of the *Quality defined through evolution*.

4.1 Quality Refined Through Evolution

In this part we will study the evolution of DataClass and GodClass design flaws. However this approach is not inherently restricted only to these two flaws; they are only examples chosen to prove our concepts. For comments on further improvements please refer to Section 6.3.1.

In the next paragraphs we present the definitions of *DataClass* and *GodClass* design flaws, as found in the literature, together with explanation on the reasons of why they were chosen in this work.

Definition: *Data Class* "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". [Fow99]

We can regard the *DataClass* classes as the object-oriented equivalents of the classic *structures from C language*. The structures were born in the period when the main gain of programming languages was to offer the possibility to organize the data. They provide neither encapsulation nor information hiding.

Definition: *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".[Mar02]

In [Rie96, pp. 32-41] two types of *God Class* problems were found: *Behavioral God Class* and *Data God Class*. In our work we will concentrate only on the behavioral form. In order to increase the fluency of the text, throughout this diploma we will use the word *GodClass* as shortcut for *Behavioral God Class*.

Behavioral Form of God Class has as result "the creation of a god object that performs most of the work, leaving details to a collection of trivial classes". A symptom of this form of God Class flaw is the apparition of many accessor methods in the classes of the system because these classes become 'dumb data carriers'. This flaw is common among programmers who change the programming paradigm from action-oriented to object-oriented programming.

Data Form of God Class has as result that many classes access the data from a big data carrier class (the 'Data God Class'). This flaw is common when a legacy system is migrated to object-oriented design. The difference between this form of God Class and the Data Class flaw is that the former contain and centralize a much larger quantity of data from the system.

Comparing the definitions of these two design flaws we can easily observe that they are complementary. *GodClass* classes access the data which is exposed with so much generosity by *DataClass* classes. *GodClass* classes tend to be a pile of methods joint together around a few data. These can often appear as the result of migrating legacy procedural systems in Object-Oriented environments.

The occurrence of these flaws is due to the old procedural way of regarding the data and functionality organization. They are the most outstanding results of the programming paradigm change and violate the basic intention of object-oriented technology (i.e. that data and functionality should stay together).

4.1.1 Suspects Detection Refined Through Time Information

This part aims to reduce the number of *false positives* introduced by the currently defined detection strategies. The mean of achieving a better detection is represented by the information related to the occurrence of the suspects over

time. Here we can divide the possibilities of apparition as shown in the following:

- The suspects constantly appear over the system's history. Depending on the type of the suspect, this could add more credibility to the entity to be a real flaw or only a *false positive*. See Section 5.3.5.
- A suspect only appears in the last version of the system. This is a good candidate for being a false positive flaw as it may represent only a transient part of the entity's evolution.
- A suspect appears only a few times during the evolution. This may indicate that some noises interfered in the version based detections.
- An entity was a suspect during some period but in the last versions it changed its state to non-suspect. This shows us that it is possible that some refactorings took place in order to eliminate the bad-smell.

4.1.2 The Shape Of Change

Aims at combining the Lanza's taxonomy described in Section 3.2.1 with the current detection strategies' suspects. The two taxonomies are orthogonal and by combining them we add meaningful information regarding the evolution of suspect entities.

For example we will characterize the evolution of a *God Class* as IdleGodClass, PulsarGodClass or SupernovaGodClass. *Idle* shows whether a class is instable or not - to measure the stability we use *LOC stability* and *NOM stability* metrics. (Appendix B.2) *PulsarGodClass* shows that there were maintainability problems with that class and *SupernovaGodClass* reveals that the size of the class exploded during a few versions.

We are focusing on the maintainability characteristic of the quality. Thus, having a *PulsarGodClass* is a problem because the *Pulsar* characteristic reveals that the entity had some maintainability problems. On the other hand, having an IdleGodClass reveals that that entity remained in constant size so there were no maintainability problems with it.

4.2 Quality Defined Through Evolution

Analyzing the system evolution gives more information beside single version based analysis. The previous part used the evolution to *improve* the current analysis procedure. In this part we use the time as a `tool` to find *other* well known design flaws which can not be found directly by analyzing only a version.

When analyzing the evolution we can look at concrete "historical facts" (i.e. events in the system) which happened along the time and thus we can go beyond speculations which could be derived when looking only to a snapshot of

the system. The accurate detection of flaws whose definition contain expressions like *"every tine", "commonly changed", "different versions", "evolution", "new releases"* inherently require multiple versions analysis.

The detection of three of the proposed "code smells" is based on finding *entities whose properties change together*. The presentation below will start by defining the concept of "entities that change together", stating its importance and explaining its detection procedure.

Entities that Change Together

Definition: *Entities that Change Together* are those sets of homogeneous entities with a specific property which changes in the same time.

RATIONALE: Knowing the entities whose properties change together represents an important asset while maintaining a software system. They help both in *system understanding* by revealing the links behind entities and in *quality assurance* by identifying hidden malign dependencies from the analyzed system. Furthermore the results obtained this way can be used, as described below, as a basis for assessing the quality of modularization of the system.

This is a step forward beside the current approach presented in Section 3.1.2 which try to anticipate possible correlations between entities by analyzing only a version of the system.

By using the time we can uncover semantic couplings between entities which gives hints about the *changeability* problems. The kind of couplings we expect are:

- code and functionality duplication (e.g. exception handling) - Shotgun Surgery.
- unidirectional dependencies between entities (e.g. inheritance) - Parallel Inheritance Hierarchies, Shotgun Surgery.
- bidirectional dependencies between entities (e.g. dependency cycle) - Shotgun Surgery.
- the usage of side effects (e.g. global variables, public-static attributes) - Shotgun Surgery.

Beside couplings we try to identify lack of cohesion in a class with *Divergent Change*.

DETECTION PROCEDURE: Using the evolution of the system we can detect the entities that changed together. We apply an incremental procedure to increase the number of entities from a group. At first step we find groups of two elements; in the subsequent steps we combine the groups found in the previous step to obtain larger groups.

For the following three "code smells" we will instantiate this general approach by looking at changes of different properties of the entities.

4.2.1 Detection of Shotgun Surgery

Definition: *Shotgun Surgery* appears when "every time you make a kind of change, you have to make a lot of little changes to a lot of different classes".[Fow99]

RATIONALE: The "Shotgun Surgery" can be seen as an antonym for "Locality of Change" which directly influences the maintainability. *Shotgun Surgery* violates the *modular continuity* [Mey97] criterion for modular design.

Furthermore, according to *Common Closure Principle (CCP)* [Mar] the entities that change together should belong together (i.e. to the same package) so when we find large groups of entities that are changing together they can become suspects of violations of CCP.

DETECTION PROCEDURE: The same time variations of NOM (Number of Methods) or LOC (Number of Lines Of Code) gives hints about the semantic links between classes. By finding the entities which are semantically linked to many other, we find in fact the entities with *ShotgunSurgery* flaw.

4.2.2 Detection of Parallel Inheritance Hierarchies

Definition: *Parallel Inheritance Hierarchies* occur when "every time you make a subclass of one class, you also have to make a subclass of the another".[Fow99]

RATIONALE: *Parallel Inheritance Hierarchies* show another kind of links between classes. The inheritance is used as a reuse mechanism so "parallel inheritance" means "same time reuse". In statically typed languages where the inheritance is a mechanism for sub-typing we can regard it as "same time specialization". When the inheritance is used as a reuse mechanism and the groups of results contain entities that don't belong to the same module, they can be suspects for violations of the "Common Reuse Principle"[Mar].

We must emphasize that there are cases where the parallel inheritance hierarchies are not harmful (e.g. in the case when one hierarchy belongs to tests, the link between the hierarchies of Model-View-Controller)). There are, however cases when the parallel inheritance hierarchies can be seen as a special case of *Shotgun Surgery* and in this cases they are harmful.

DETECTION PROCEDURE: The classes can be identified by looking at the subclasses of a certain class that are added together (i.e. the same time variation of the Number Of Children (NOC) B metric).

4.2.3 Detection of Divergent Change

Definition: *Divergent Change* occurs "when one class is commonly changed in different ways for different reasons".[Fow99]

RATIONALE: This is an indirect measurement of the class cohesion. A low cohesive class could indicate that the class has features that belong to more than one abstraction. This would violate the *Class Consistency Principle* [Mey97]. If the cohesion is low then the *Split Class* [Fow99] refactoring might be the solution.

DETECTION PROCEDURE: These groups of methods can be identified by looking at the methods of a certain class that change together.

4.2.4 Detection of Speculative Generality

Definition: *Speculative Generality* occurs when users implemented "all sorts of hooks and special cases to handle things that aren't required".[Fow99]

RATIONALE: The increase in the number of classes in a system increases the system's complexity. When there are useless classes the system becomes cluttered.

An abstract base class is intended to factor some common functionality. If the class has only a child then obviously there is no functionality factored. When this situation spreads over a long time from system's evolution that is a sign that the designer did not anticipated well the further evolution. This violates Heuristic 3.7 and 3.9. [Rie96]

> HEURISTIC 3.7: Eliminate irrelevant classes from your design.
>
> HEURISTIC 3.9: Do not turn an operation into a class. Be suspicious of any class whose name is a verb or derived from a verb. Especially those which have only one piece of meaningful behavior (i.e. do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class.

This flaw is a side effect of "design for change" paradigm. When the paradigm is overused it produces excessively-flexible code which could turn out to be difficult to understand. The higher the flexibility the less simple the classes are, and the less maintainable the code is.

As shown in [Par94] and presented in Section 2.4.2 the success of the "design for change" is limited by humans ability to predict the future. When the future is mis-predicted the investment made is useless and thus we have *Speculative Generality*.

DETECTION PROCEDURE: The evolution of the NOC (Number of Children) gives hints about the functionality added. If NOC for an highly abstract class

is one and it remains so for a long time we have a suspect of **'Speculative Generality'** [Fow99].

4.3 Extensions

Here we will present a short term extension of this work. The reason for not including these extensions in Section 6.3 is that at the moment on writing this document the work on them was in progress.

Detection of Fragile Base Class Possible Problems

DESCRIPTION: *Syntactic Fragile Base Class Problem* - "is about binary compatibility of compiled classes with new binary releases of superclasses.[...] The idea is that a class should not need recompilation, just because purely syntactic changes to its superclasses 'interfaces have occurred". [Szy98]

DESCRIPTION: *Semantic Fragile Base Class Problem* - "How can a subclass remain valid in the presence of different versions and evolution of the implementation of its superclasses" [Szy98]

RATIONALE: The highly instable parts of the system give us some hints about the recurrent violations of *Open-Closed Principle*. The more abstract a class is and the more higher in the inheritance tree it is placed, the more problematic Open-Closed violations are. In an environment where are more programmers which develop the system, the recurrent modifications of a base class could easily lead to the *fragile base class problem*.

DETECTION PROCEDURE: To detect the *fragile base class* we look at the hight of a class in the inheritance hierarchy and at the stability of it.

Chapter 5

Validation

The purpose of this part is to show how we validated the mechanisms that we developed in the current work. We start this chapter by presenting *V-Soda*, the tool that we used for making the experiments. In the second section we will present the experimental setup. The third and fourth sections present the experiments we did for the two parts of our diploma: *Quality refined through evolution* and *Quality defined through evolution*.

5.1 Tool Support

We did our experiments with the help of **Smalltalk Object-oriented Design Analyzer** (Soda) which is integrated into a larger environment named **Versions Analyzer** (Van). Thus we will refer to our tool through the name of **V-Soda**. In Appendix C we present the list of use-cases for *V-Soda*. The front-end of our tool, the User Interface, is built over a framework which allows to easily work with detection strategies.

In order to minimize the problem of the detection strategy thresholds presented in Section 3.1.2 we built our tool to favor the most frequently cases which appear when running the detection strategies - the possibility to easily change the thresholds.

The front-end of *V-Soda* consists of two modules which will be presented in the following: *Detection Strategies Manager* and *Entities That Change Together Manager*.

5.1.1 Detection Strategies Manager

This module integrates all the operations necessary for working with detection strategies. It allows the user to manage the set of detection strategies (i.e. to create a new detection strategy, to modify the existing ones and to save them

into a library) and to run them over the analyzed system.

Detection Strategies Manager is centered on the most common case in using the detection strategies - analyzing a system. As shown in Section C.2 during the system analysis the user usually runs iteratively a particular detection strategy, modifies the thresholds and inspects the partial results. An image of this UI is given in Figure 5.1.

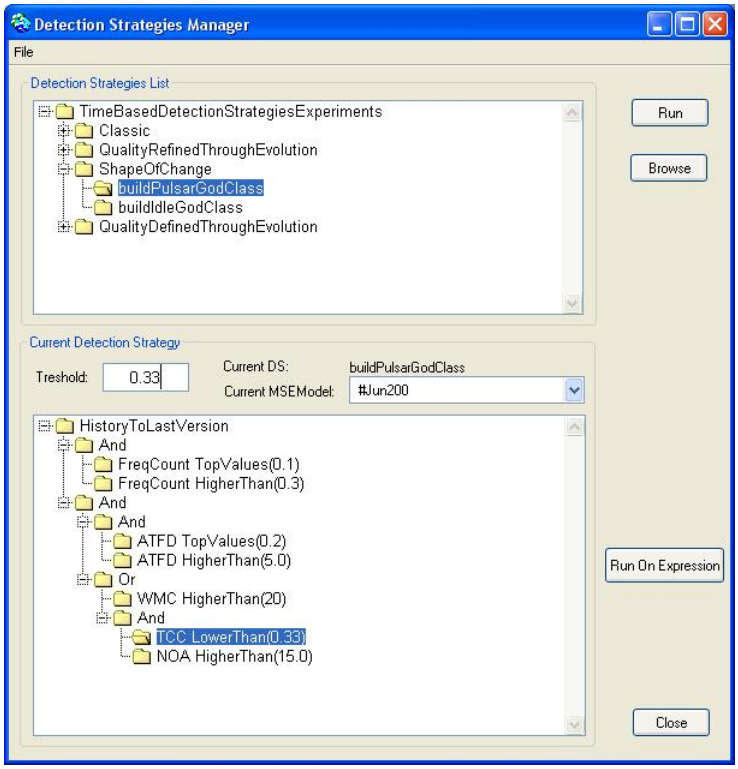

Figure 5.1: Detection Strategy Manager

5.1.2 Entities That Change Together Manager

This is the user front-end when working with 'Entities that Change Together'. We can observe that the user can easily modify the number of versions during which the specific change occurs and the number of entities that change in the same time. The results obtained by running an operator are displayed in the right panel. By selecting a group the system displays in the below list-box the names of the versions in which the change occurred. In Figure 5.2 is presented this user interface.

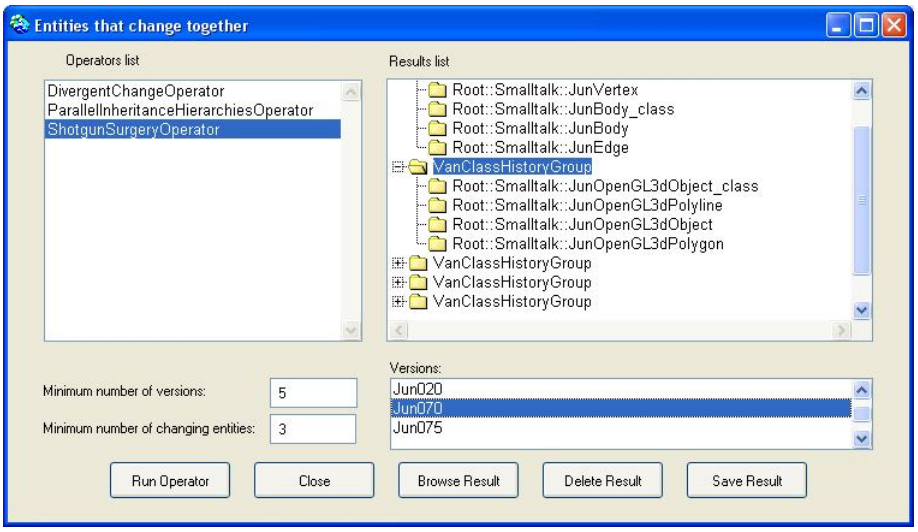

Figure 5.2: Entities that Change Together Manager

5.2 Experimental setup

5.2.1 The Case-Studies

In the next paragraphs is presented a brief information about the case-studies used for performing the experiments. At the beginning of each paragraph there is a general description of the real world purpose of the program used as case-study and of the sampling basis for the analyzed versions. Next follow tables with metrics that describe the characteristics of experiments in a deeper detail. All metrics used in these tables are described in Appendix B.

Conan Case Study *Conan* is a tool for Concept Analysis developed as support for a PhD thesis. The classic detection strategies are applied on the latest version available at 25th April 2003 (i.e. 1.102). The time-based detection strategies are applied on 22 versions collected from 5 to 5 versions, starting at version 1.02 and ending at version 1.102. In Table 5.1 are presented some characteristics of *Conan*.

Classifier Case Study **Classifier** is a software developed during a diploma project. Version based detection strategies were applied on the latest version available at 25th April 2003 (i.e. 1.89). The time based detection strategies are applied on 17 versions collected on 5 versions basis, starting with version 1.10. The characteristics of **Classifier** are presented in Table 5.2.

Metric	Version 1.02	Version 1.102
Number of Classes	143	223
Number Of Non Library Classes	104	150
Number of Non UI and Test Classes		80
NOCH	258	
NONLCH	172	
ENC	124	
LENC	16.9	
ENNLC	46	
LENNLC	5.4	
ANOC	205.55	

Table 5.1: An overview on the size and evolution of *Conan* case-study

Metric	Version 1.10	Version 1.89
Number of Classes	208	254
Number Of Non Library Classes	92	108
Number of Non UI and Test Classes		50
NOCH	265	
NONLCH	108	
ENC	62	
LENC	9.26	
ENNLC	18	
LENNLC	4.14	
ANOC	235.47	

Table 5.2: An overview on the size and evolution of *Classifier* case-study

Jun Case Study **Jun** is a high-level application framework for 3D graphics, available as a free, open source package. It is developed by SRA's Object-Oriented Group (Jun's Home Page). Table 5.3 presents the characteristics of Jun. A number of 40 versions of this case-study were analyzed, starting with version 5 and ending with version 200. The classic detection strategies were applied on version 200.

5.2.2 Experiments goals

Quality Refined Through Evolution Our goal is to *prove that the system evolution adds an important quantity of information to the current way of detecting the quality flaws which helps us to find more accurately the real design flaws* (i.e. to reduce the number of *false positives*).

Metric	Version 5	Version 200
Number of Classes	261	974
Number Of Non Library Classes	160	694
NOCH	1109	
NONLCH	814	
ENC	713	
LENC	75.92	
ENNLC	534	
LENNLC	54.89	
ANOC	626	

Table 5.3: An overview on the size and evolution of *Jun* case-study

Quality Defined Through Evolution *We want to prove that it is feasible and meaningful to detect the flaws by using the entities whose properties change together.* We are mostly interested here if the links between the entities that change together that we found with our mechanism are real links (i.e. the entities are related) or are false positives (i.e. the fact that the specific entities changed in the same time was a pure coincidence).

5.2.3 Experiments methodology

In the previous chapter we presented our approach to combine the *software evolution* and *detection strategies*. Here we present the methodology used for doing the experiments by describing the steps we did while measuring the system and the rationals behind them. All the metrics used during the measurements are described in Appendix B.

Quality Refined Through Evolution

In this part our approach is to vary the thresholds while adding the time as additional information. We assume that the skeletons of the detection strategies are correct (i.e. the metrics used in detection strategies are the best possible for identifying the corresponding flaws) and only thresholds represent a problem. This way when we apply the most relaxed thresholds we have no *false negatives*.

At first, we try to find manually the thresholds which give the least *false positives* and *false negatives* and after that we change the thresholds but we look to the *time* information to see if the experiments confirm our assumptions presented in the previous chapter.

The real design flaws were found by manually inspection of the suspects given when using the more relaxed thresholds. When this inspection was not possible due to the dimension of the analyzed case-study (i.e. Jun) we assumed that the first experiment which has the most constrained values for thresholds has the highest accuracy in respect to *false positives* and the last experiment

with the least constrained thresholds introduces no *false negatives*. The time measurements refer to the variation of entity's properties (i.e. the variation of the *property of being a suspect* versus the variation of *stability*) during the system's evolution.

Quality Defined Through Evolution

Our approach is to find the entities that change together during a period of their life. The two variables at our hand are: the size of the group of entities and the number of versions in which the change occurred for a specified group. The number of versions in which changes appear should be higher than 10% from the analyzed history otherwise we consider that the change appeared randomly.

The *noise* can interfere due to the coincidences occurred during the system's evolution. Thus the entities that are detected to change together might have no links between them, the results being *false positives*. The *false negatives* can appear when the number of analyzed versions is insufficient and we can not detect the flaws by looking to the system's history. As shown in the description of our tool from Appendix C.3, in addition to considering the period in time between changes, we approach the problem of *false positives* by allowing the user to delete the irrelevant results. For each of the operators we can customize the number of entities that should change together and the number of versions in which the specified entities changed. The bigger the number of versions is the stronger the link between the resulted entities.

Finally, after we got the results, we manually inspect in what measure the classes from the set of results are linked. The link tests are done by using the source browser (reference finder) from VisualWorks. We resume ourselves to looking if the objects from a specified class are instantiated in the other classes from the resulted group. If we find that a class is instantiated in another, we consider that the two classes are really linked and that this result was not obtained by accident. Otherwise we consider that the link between those classes appeared by accident thus the result is a *false positive*. A more comprehensive list of links between classes is given in the next paragraph.

The links between classes We identified the following types of links between classes (our definitions are symmetrically - i.e. we can interchange A and B):

- Instantiation - two classes A and B are linked if in class A we instantiate an object of type B
- Inheritance - two classes A and B are linked if B inherits from A
- Method invocation - two classes A and B are linked if A has a method which calls a method of B.
- Polymorphic link - two classes A and B are linked if from class A we call a method of class C, where C is the super-type (ancestor) of B.

The yesterday weather phenomenon The *yesterday weather phenomenon* [GDM03] can be responsible for an important part of the *noise*. We eliminate this possible noise by looking at the distance between the version where the first change was recorded and the version where the last change was recorded. The larger the distance in time between changing versions, the more interesting the results should be as changing in subsequent versions can be the result of concentrated work on a specific part of the system.

5.3 Quality Refined Through Evolution

We will show in the following how we can reduce the noise by using the system history as an additional source of information. Different sets of tables are given for each of the analyzed projects. At first there are presented the suspects as detected on a single version of the system (i.e. the latest) by using different values for thresholds. This is followed by time-enriched results which are computed using the system's history.

The purpose of the timed analysis is to show that we can correlate the thresholds used in classic detection strategies with different time-based information (i.e. the number of versions in which a suspect has a DataClass or GodClass property; the stability and instability in terms of LOC and NOM metrics).

A short characterization for the found suspects is given after each set of tables. This informal characterization was obtained by manual inspection of the analyzed source code. Each suspect is given a number of *stars* which is proportional to the degree in which we considered the described suspect to be a real flaw. The empirical results are correlated with manual inspection in the summarizing comments which end the experiment.

After each set of experiments (i.e. DataClass and GodClass experiment sets) we try to draw some conclusions about how the evolution of suspects used as extra source of information could compensate the lack of precision in some of the chosen thresholds. These conclusions were obtained by generalizing the partial empirical results described above. In the end we make a comparison between the evolution of DataClass and GodClass suspects as they represent two extremities of the object-oriented design flaws (for more comments please refer to Section 5.3.5). A taxonomy that characterize the evolution of these flaws will end this section.

Driving Questions In order to narrow the goals stated above, during the experiments we will try to answer the following specific questions:

1. How does the time affect the *DataClass* and *GodClass* status for a class ?
2. In what measure the suspects are stable/instable in comparison with the other classes ?
3. In what measure the real flaws are stable/instable in comparison with the other suspects ?

4. What is the shape of change of the suspects ?
5. Can we correlate the shape of change with a certain type of suspects ?

Detection Strategies Used The above questions will be answered by using the detection strategies enumerated below:

1. *Classic* - DataClass, GodClass.
2. *Time Based* - TimedDataClass, TimedGodClass
3. *Stability* - DataClassNOMStability, DataClassLOCStability, GodClassNOMInstability, GodClassLOCInstability
4. *Shape Of Change* - PulsarGodClass, SupernovaGodClass, IdleDataClass, IdleGodClass.

5.3.1 DataClass Experiments

The rule of the *DataClass* detection strategy is given in the following. The notation used is *SOD* language defined in [Mar02].

```
DataClasses:=
    ((WOC, BottomValues(33%)) and (WOC, LowerThan(0.33)))
     and ((NOPA, HigherThan(5)) or (NOAM, HigherThan(5)))
```

The thresholds used above are only for explanation purpose. They are taken as they were originally defined in [Mar02]. In our experiments we use other sets of thresholds in order to make links between them and the time information.

Conan Case Study

DataClass Suspects	**Apparition Freq**
Experiment 1	
ConanNavigatorBrowser (UI)	1
ConceptAnalysisUI (UI)	1
ConceptPatternType	14/19
PropertyKey	1
Experiment 2	
AbstractConceptAnalysisElement	1/3
ConanNavigatorBrowser (UI)	1
ConceptAnalysisLatticeEdge	1
ConceptAnalysisUI (UI)	1
ConceptPatternType	14/19
PropertyKey	1
Experiment 3	
AbstractConceptAnalysisElement	1/3
ConanNavigatorBrowser (UI)	1
ConanNodeEnumerator	1
ConceptAnalysisLatticeEdge	1
ConceptAnalysisObject	6/7
ConceptAnalysisUI (UI)	1
ConceptPatternType	14/19
PropertyKey	1

Experiment 1
DataClass := ((WOC, BottomValues(20%)) and (WOC, LowerThan(0.51)))
and ((NOPA, HigherThan(3)) or (NOAM, HigherThan(3)))

Experiment 2
DataClass := ((WOC, BottomValues(20%)) and (WOC, LowerThan(0.70)))
and ((NOPA, HigherThan(3)) or (NOAM, HigherThan(3)))

Experiment 3
DataClass := ((WOC, BottomValues(40%)) and (WOC, LowerThan(0.80)))
and ((NOPA, HigherThan(2)) or (NOAM, HigherThan(2)))

Figure 5.3: Evolution of DataClass property in *Conan* case-study

A Short Characterization of Suspects

- ConceptAnalysisLatticeEdge(****) very basic functionality (printOn, hash, =) and accessors.

DataClass Suspects	**LOC Stability**	**NOM Stability**
PropertyKey	1	1
ConceptAnalysisLatticeEdge	19/20	1
ConceptPatternType	17/18	17/18
AbstractConceptAnalysisElement	19/20	19/20
ConanNodeEnumerator	19/20	19/20
ConanNavigatorBrowser	6/7	1
ConceptAnalysisUI	17/20	1
ConceptAnalysisObject	7/10	3/4

Figure 5.4: Stability of DataClass suspects in *Conan* case-study

- PropertyKey(****) same as above.
- ConceptPatternType(****) same as above. The apparition among DataClasses is due to refactoring the old instance variables and adding accessors in version 1.40.
- AbstractConceptAnalysisElement(***) only accessor methods. A lot of children use this data. This class is a result of a refactoring in version 1.75. It has always been a light-weight class.
- ConanNodeEnumerator(***) a lot of selects and tests over the contained data. The functionality is more like filtering.
- ConanNavigatorBrowser(***) a UI class - with lots of accessors but they are only wrappers for instance variables.
- ConceptAnalysisUI(**) a UI class - lots of accessor methods wrappers for instance variables used as UI aspects. Again, this class distributes its data to other classes which are interested in its data.
- ConceptAnalysisObject(**) some accessor methods and some functionality. Has a lot of children. Refactored in version 1.20.

Comments on the suspects found From Table 5.3 we can observe that each time we relax the thresholds we find new suspects which do not have the DataClass property from the very beginning of their lifetime (e.g. *AbstractConceptAnalysisElement* and *ConceptAnalysisObject* which indeed during the manual inspection proved to be less specific DataClass flaws). On the other hand we found that *ConceptAnalysisLatticeEdge* which proved to be a real DataClass and which was considered a false negative when using the most constrained thresholds set has had this property over its entire life.

If we ignore UI classes (due to their particularities 5.3.2) we find that during the Experiment 2 and 3 the only entire life-time suspect without being a real flaw is *ConanNodeEnumerator*.

We can notice from the table showing the stability that, with small exceptions (i.e. *ConceptPatternType*), the increasing of LOC stability is correlated with the degree in which a suspect is a real flaw.

Classifier Case Study

DataClass Suspects	**Apparition Freq**
Experiment 1	
CCCompositeEdge	1
GroupUI (UI)	1
Experiment 2	
AbstractGroupUI (UI)	1
AbstractOperation	1
CCCompositeEdge	1
CCCompositeNode	1
CCFAMIXCompositeNodePlugin	1
ClassifierGroup	1
GroupUI (UI)	1
MapType	1
MSESinglePropertyQuery	1

Experiment 1
DataClass := ((WOC, BottomValues(20%)) and (WOC, LowerThan(0.7)))
and ((NOPA, HigherThan(3)) or (NOAM, HigherThan(3)))

Experiment 2
DataClass := ((WOC, BottomValues(20%)) and (WOC, LowerThan(0.80)))
and ((NOPA, HigherThan(2)) or (NOAM, HigherThan(2)))

Figure 5.5: Evolution of DataClass property in *Classifier* case-study

A Short Characterization of Suspects

- CCCompositeEdge(****) - some accessors and very light weighted.
- CCCompositeNode(****) - light weighted and has many accessors.
- MapType(****) - lots of non-conventional accessors. Quite light-weighted.
- ClassifierGroup(***) - the class is not very light weighted.
- MSESinglePropertyQuery(***) - quite light weighted class with lot of initializers.

DataClass Suspects	**LOC Stability**	**NOM Stability**
MapType	1	1
CCCompositeEdge	1	1
CCCompositeNode	13/16	7/8
ClassifierGroup	5/8	3/4
MSESinglePropertyQuery	6/7	6/7
AbstractGroupUI (UI)	0	5/16
GroupUI (UI)	1/7	4/7
AbstractOperation	1	1
CCFAMIXCompositeNodePlugin	15/16	15/16

Figure 5.6: Stability of DataClass suspects in *Classifier* case-study

- AbstractGroupUI(**) - UI class. Lots of accessor methods which are wrappers for the aspects.
- AbstractOperation(**) - has a lot of children. Many accessor methods. Not light weighted. Is the result of a refactoring process as it appears only in the version 1.89.
- GroupUI(**) - lots of wrapper methods.
- CCFAMIXCompositeNodePlugin(**) - small class which inherits a lot of functionality.

Comments on the found suspects The frequency counting of the suspects' occurrence does not give extra information here for finding the real flaws. However, they confirm the general trend for DataClass suspects of having this property for almost all of their life. The low dynamics of this case-study (see Table 5.2) can be an explanation for the fact that no refactorings which could generate DataClass suspects were done.

The stability counting reveals that the stability of a class is directly related with its status as a flaw. The AbstractOperation suspect which, as shown above, appeared only in the last version of the system and in this way is explainable its high stability even if it is not a real flaw.

Jun Case Study

DataClass Suspects	**Apparition Freq**
Experiment 1	
1 Class History	$< 95\%$
12 Class Histories	$\geq 95\%$
Experiment 2	
4 Class Histories	$< 95\%$
41 Class Histories	$\geq 95\%$
Experiment 3	
9 Class Histories	$< 95\%$
51 Class Histories	$\geq 95\%$

Experiment 1
DataClass := ((WOC, BottomValues(10%)) and (WOC, LowerThan(0.5)))
and ((NOPA, HigherThan(6)) or (NOAM, HigherThan(6)))

Experiment 2
DataClass := ((WOC, BottomValues(20%)) and (WOC, LowerThan(0.70)))
and ((NOPA, HigherThan(4)) or (NOAM, HigherThan(4)))

Experiment 3
DataClass := ((WOC, BottomValues(20%)) and (WOC, LowerThan(0.80)))
and ((NOPA, HigherThan(4)) or (NOAM, HigherThan(4)))

Figure 5.7: Evolution of DataClass property in *Jun* case-study

Comments on the found suspects The size of Jun did not allowed an exhaustive inspection of all the suspects which we found. However, we tried to make relative comparisons between the DataClass suspects.

Table 5.7 shows that the ratio of DataClass suspects which have this property for a shorter period of their life doubles when we relax the thresholds. (Experiment 1 compared with Experiment 3)

When we compare the LOC stability for DataClass suspects we observe that 100% of the classes from the more constrained set have the stability higher than 80% of their life-time compared with only 81% of the suspects from the most relaxed set.

When comparing the more constrained suspects with the less constrained ones we find that the tighter the constraints the more stable the suspects.

DataClass Suspects	**LOC Stability**
The Suspects From Experiment 1	
6/13 = 46% of DataClass Suspects	> 95%
9/13 = 69% of DataClass Suspects	> 90%
13/13 = 100% of DataClass Suspects	> 80%
The Suspects From Experiment 2	
23/45 = 48% of DataClass Suspects	> 95%
32/45 = 71% of DataClass Suspects	> 90%
41/45 = 91% of DataClass Suspects	> 80%
The Suspects From Experiment 3	
27/60 = 45% of DataClass Suspects	> 95%
38/60 = 63% of DataClass Suspects	> 90%
48/60 = 80% of DataClass Suspects	> 80%

Figure 5.8: LOC stability of DataClass suspects in *Jun* case-study

DataClass Suspects	**NOM Stability**
The Suspects From Experiment 1	
7/13 = 54% of DataClass Suspects	> 95%
9/13 = 69% of DataClass Suspects	> 90%
13/13 = 100% of DataClass Suspects	> 80%
The Suspects From Experiment 3	
32/60 = 53% of DataClass Suspects	> 95%
40/60 = 67% of DataClass Suspects	> 90%
54/60 = 90% of DataClass Suspects	> 80%

Figure 5.9: NOM stability of DataClass suspects in *Jun* case-study

5.3.2 Adding Time to DataClass Detection Strategy

As shown in the experiments above, the majority of DataClass suspects(i.e. more than 90% from the total number of suspects) have this property over more than 90% of their lifetime. This implies that DataClasses are flaws born from the very beginning of the class entities (i.e. a DataClass class was more likely designed to be a DataClass instead evolving into a DataClass). This way we could identify 10% of the *false positives* as classes which had the DataClass property on a shorter period of their lifetime. Of course, these classes have transformed themselves into DataClasses through refactorings (either there was removed some functionality or some data was added) and some of them could be real flaws but the empirical data showed that it is more likely that these suspects to be false positives. A possibility would be that some of the data were moved up from children to parents (e.g. **Conan** - AbstractConceptAnalysisElement,

AbstractConceptAnalysisObject) .

Here is an issue (specific to Smalltalk - because all data are private and we have no 'protected' keyword) about who use this data: foreign classes or children. If the data is used by children is less painful comparing with the other case.

We could find some *false negatives* among the suspects which were not detected when using the most constrained thresholds but whose DataClass property span over their entire lifetime. As DataClass flaws are the descendants of classic structures from C language, used only as a way to organize the data, they tend to remain unchanged for their entire life.

Furthermore, the typical DataClasses proved to be very stable (thus the majority of DataClass suspects are *IdleClass*). This is quite normal as by definition a DataClass is a light and dumb class which has almost no responsibility. The complexity of a suspect is given in the current definition using the WOC metric. This metric it assumes that complexity is given by the size of the non-trivial interface of the class.

A recurrent problem while identifying the DataClasses is represented by UI classes. In their case there is a lot of inherited functionality provided by the MVC framework. All that children have to supply is the data required for UI aspects (which is usually quite a lot of data) and some basic functionality to maintain the UI consistent with changes fired up by the user.

Out of the presented results we can see that the *time* adds a significant amount of information to the *classic DataClass detection strategies*. The evolution is therefore meaningful to be incorporated into a new way of detecting the DataClass suspects. This way we obtain the **TimedDataClass** and **StableDataClass** detection strategies. The results obtained from applying them with the most relaxed set of thresholds for the classic *DataClass* was summarized in Table 5.10.

The definitions of the newly defined detection strategies, as given in SOD [Mar02], are:

```
TimedDataClass:=
     (((WOC, BottomValues(33%)) and (WOC, LowerThan(0.50)))
            and ((NOPA, HigherThan(5)) or (NOAM, HigherThan(5)))
  HistoryToVersionMatrixCollection
     (ApparitionFreqCount, HigherThan(95%)))


StableDataClass:=
     (((WOC, BottomValues(33%)) and (WOC, LowerThan(0.50)))
             and ((NOPA, HigherThan(5)) or (NOAM, HigherThan(5)))
  HistoryToLastVersionAdapter
     (LOCStability, HigherThan(90%))
```

Case Study	Detection Strategy Name	No. of suspects	No. of flaws	No. of False Pos	No. of False Neg
Classifier (Experiment 2)	DataClass	9	3	6	0
	TimedDataClass	9	3	6	0
	StableDataClass	4	2	2	1
Conan (Experiment 3)	DataClass	8	3	5	0
	TimedDataClass	5	2	3	1
	StableDataClass	5	3	2	0
Jun (Experiment 3)	DataClass	60	?	?	?
	TimedDataClass	51	?	?	?
	StableDataClass	38	?	?	?

Figure 5.10: Time Based DataClass conclusions

Conclusions: From Table 5.10 we can observe that using StableDataClass reduces a lot the number of suspects. However StableDataClass introduces more *false negatives* than TimedDataClass. The first is very vulnerable to minor modifications (e.g. initializations) while the later is vulnerable to refactorings (i.e. it doesn't catch the evolutionary DataClass flaws which are the results of some refactorings).

The fact that more than 90% of the DataClasses have this property from their birth (i.e. that the number of the *EvolutionaryDataClass* - see Section 5.3.6 - classes is small) is at first sight somehow surprising as we would expect that more DataClasses are the result of some refactorings. However, if we consider that the Object-Oriented technology offers support for the data centric decomposition of a domain problem we find that the data abstractions identified at the design time, which correspond to the nouns found in the modeled domain, are very stable and thus the movement of the data between classes is less likely to happen. The massive functionality extraction from a class while the data remain on the same place is also quite unlikely to happen.

We enumerate the major refactorings [Fow99] (i.e. we ignore those triggered by the usage of a multi-paradigm language - e.g. *Replace record with Data Class* - and most of those related to generalization (because it is arguable whether a class up in the hierarchy can be DataClass or not) - e.g. Extract Superclass), which could lead to DataClass classes: *Move method*, *Move field*, *Extract class*, *Pull Up Method*. We will look at the rationale which drive these refactrings:

- Move Method - "I usually look through the methods on a class to find a method that seems to reference another object more than the object it lives on". Moving methods triggers DataClasses only when they did not belong at all to the starting class thus at the design time there was a huge misunderstanding of the class's purpose - and this is quite unlikely to happen.

- Move Field - "I consider moving a field If I see more methods on another class using the field than the class itself."[Fow99] From this description it is obvious that this refactoring ca not lead to DataClasses as the field is moved in the direction of functionality.

- Extract Class - "Such a class is one with many methods and quite a lot of data. A class that is too big to understand easily."[Fow99] The results of this refactoring is unlikely to be DataClass classes.
- Pull Up Method - "The easiest case of using Pull Up Method occurs when the methods have the same body, implying that there's been a copy and paste"[Fow99]. This refactoring is usually a means to factorizing the behavior of two children into their superclass and thus also unlikely to produce DataClass classes.

What we wanted to prove above was that in a decent object-oriented system it is indeed quite unlikely to obtain DataClass classes through refactoring and this confirms our empirical results.

5.3.3 GodClass Experiments

In the following we will present the *GodClass* expression that we used for our experiments (the language used is *SOD* [Mar02]). This expression is modified beside the original one which was presented in Section 3.1.2.

```
GodClasses :=
    ((ATFD, TopValues(20%)) and (ATFD, HigherThan(4)))
  and
       ((WMC, HigherThan(20))
    or
       ((TCC, LowerThan(0.33) and (NOA, HigherThan(3)))))
```

Because of the large number of methods and the small number of attributes from a class (e.g. especially for meta-classes), the TCC (Tight Class Cohesion) metric is always very small. If the number of attributes for a class is 0 then TCC will be 0 (for the computation details of TCC please refer to Section B.1.6) and according to the above definition many classes will be reported as *GodClass* suspects.

Conan Case Study

A Short Characterization of Suspects

- ConceptAnalysisConcept(****) - a big class with many methods.
- ConcetpAnalysisContext(****) - a very big class.
- InternalMSEClassBehavior(***) - many of the methods are heavy even if their cyclomatic complexity is not very high.
- MSEInvocationObject_class(***) - heavy weighted class. No class variables are declared so the class is only a collection of independent methods.

GodClass Suspects	**Apparition Freq**
Experiment 1	
ConanApp	1/3
ConceptAnalysisConcept	1
ConceptAnalysisContext	1
Experiment 2	
ConanApp	1
ConanNavigatorBrowser	1
ConceptAnalysisConcept	1
ConceptAnalysisContext	1
MSEInvocationObject_class	7/8
InternalMSEClassBehavior	5/17

Experiment 1
GodClasses := ((ATFD, TopValues(10%)) and (ATFD, HigherThan(30))) and ((WMC, HigherThan(50)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(10)))))

Experiment 2
GodClasses := ((ATFD, TopValues(10%)) and (ATFD, HigherThan(20))) and ((WMC, HigherThan(30)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(10)))))

Figure 5.11: Evolution of GodClass property in *Conan* case-study

- ConanApp(***) - a few heavy-weighted methods are present and a lot of light weighted methods.
- ConanNavigatorBrowser(*) - UI class. A lot of wrapper methods for instance variables which use late initialization. These conditionals increase the WMC by 30%.

Comments on the found suspects We found two very big classes: ConceptAnalysisConcept and ConcetpAnalysisContext which after manual inspection proved to be big from their birth. However these two classes are not conforming to the general intuition that GodClass suspects are more a result of system's evolution.

On the other hand we found two classes (i.e. InternalMSEClassBehavior and ConanApp) which became GodClass suspects by evolution. These classes have different structures: ConanApp has NOM = 44 and WMC = 52, while InternalMSEClassBehavior has in NOM = 32 and WMC = 33 (both measurements were done in version 1.102). This anomaly (i.e. after manual inspection we found InternalMSEClassBehavior to be more GodClass suspect than

GodClass Suspects	LOC Instability	NOM Instability
ConceptAnalysisContext	3/7	1/3
ConceptAnalysisConcept	3/7	2/7
InternalMSEClassBehavior	8/17	6/17
MSEInvocationObject_class	1/8	1/8
ConanApp	7/9	7/9
ConanNavigatorBrowser	1/8	0

Figure 5.12: Stability of GodClass suspects in *Conan* case-study

ConanApp) is due to the fact that the relevance of the *cyclomatic complexity* [McC76] decreases generally in object oriented languages and especially in Smalltalk. (see Section B.1.3).

MSEInvocationObject_class represents another kind of classes. It has only 19 methods (version 1.102) but the majority of them are quite complex. Without any instance variable, it acts as a bunch of methods put together, more like functions in procedural code.

The stronger GodClass suspects proved to be as expected more instable than the others.

Classifier Case Study

A Short Characterization of Suspects

- ClassifierDrawing(***) - heavy-weighted class. Its ATFD metric has low value even if the WMC is very high.
- AbstractGroupUI(**) - a big UI class with lots of methods, many of them being wrappers for aspects variables. Part of the complexity is due to the late initialization.
- GroupUI(*) - UI class, not heavy-weighted.
- AbstractOperation(*) - not very heavy-weighted.

Comments on the found suspects The most prominent GodClass suspect (i.e. ClassifierDrawing) got this feature during its lifetime.

Again we have here two UI classes. One of them (i.e. AbstractGroupUI) wraps a very complex dialog. Due to the fact that this project is quite small the UI classes have a greater importance here. If we ignore the two graphical interface classes we can see the following:

- ClassifierDrawing has a higher instability beside AbstractOperation which corresponds to the degree in which it is a GodClass flaw. When we used

GodClass Suspects	**Apparition Freq**
Experiment 1	
AbstractGroupUI	1
Experiment 2	
AbstractGroupUI	1
AbstractOperation	1
ClassifierDrawing	5/17
GroupUI	5/8

Experiment 1
GodClasses := ((ATFD, TopValues(10%)) and (ATFD, HigherThan(30))) and ((WMC, HigherThan(50)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(10)))))

Experiment 2
GodClasses := ((ATFD, TopValues(10%)) and (ATFD, HigherThan(20))) and ((WMC, HigherThan(30)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(10)))))

Figure 5.13: Evolution of GodClass property in *Classifier* case-study

GodClass Suspects	**LOC Instability**	**NOM Instability**
ClassifierDrawing	10/17	5/17
AbstractGroupUI	16/17	11/17
GroupUI	3/4	3/8
AbstractOperation	4/17	3/17

Figure 5.14: Stability of GodClass suspects in *Classifier* case-study

the more constrained thresholds we got ClassifierDrawing as a *false negative* but analyzing its instability we could compensate the noise when computing the McCabe's cyclomatic complexity (Section B.1.3).

- ClassifierDrawing was detected to be a pulsar which amplifies its instability.

AbstractGroupUI and GroupUI were detected as supernovas. The manual inspection proved that both of this classes grew in the first part of their life due to the features added to the GUI.

Fraction of GodClasses	**Apparition Freq**
Experiment 1	
5/14 = 36%	> 95%
1/14 = 7%	> 70% & < 95%
8/14 = 57%	< 70%
Experiment 2	
11/20 = 55%	> 95%
2/20 = 10%	> 70% & < 95%
7/20 = 35%	< 70%
Experiment 3	
20/37 = 54%	> 95%
5/37 = 14%	> 70% & < 95%
12/37 = 32%	< 70%

Experiment 1
GodClasses := ((ATFD, TopValues(5%)) and (ATFD, HigherThan(70))) and ((WMC, HigherThan(90)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(15)))))

Experiment 2
GodClasses := ((ATFD, TopValues(10%)) and (ATFD, HigherThan(50))) and ((WMC, HigherThan(70)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(15)))))

Experiment 3
GodClasses := ((ATFD, TopValues(10%)) and (ATFD, HigherThan(30))) and ((WMC, HigherThan(50)) or ((TCC, LowerThan(0.15) and (NOA, HigherThan(10)))))

Figure 5.15: Evolution of GodClass property in *Jun* case-study

Jun Case Study

GodClass Suspects	**NOM Instability**
10/14 = 71% of GodClass Suspects	Top(5%)
13/14 = 93% of GodClass Suspects	Top(10%)
14/14 = 100% of GodClass Suspects	Top(25%)

Figure 5.20: NOM instability of Experiment 1 suspects in *Jun* case-study

Fraction of GodClass Suspects	**LOC Instability**
12/37 = 32%	Top(5%)
20/37 = 56%	Top(10%)
27/37 = 73%	Top(25%)
30/37 = 81%	Top(35%)

Figure 5.16: LOC instability of Experiment 3 suspects in *Jun* case-study

GodClass Suspects	**NOM Instability**
15/37 = 41% of GodClass Suspects	Top(5%)
21/37 = 57% of GodClass Suspects	Top(10%)
27/37 = 73% of GodClass Suspects	Top(25%)
31/37 = 84% of GodClass Suspects	Top(35%)

Figure 5.17: NOM instability of Experiment 3 suspects in *Jun* case-study

Fraction of GodClass Suspects	**Shape of Change**
0 = 0%	Idle (ChangeFreqCount LowerThan(0.05))
7/14 = 50%	Pulsars (Freq HigherThan(0.1))
11/14 = 79%	Supernova1 (HigherStep HigherThan(7))

Figure 5.21: Shape of change of Experiment 1 suspects in *Jun* case-study

Due to the size of this Case Study no manual inspection was feasible so all the results are drawn from the measurements.

Comments on the found suspects As shown in Table 5.7 almost half of the classes were not born as GodClass suspects. We can also observe that the stronger suspects have a higher probability to be GodClasses by evolution.
Comparing the other set of four tables we can easily observe that the more constrained the thresholds for GodClass are the more instable the class is. Furthermore we notice that stronger GodClass suspects are in a greater proportion Pulsars and Supernovas and none of them is Idle.

Fraction of GodClass Suspects	Shape of Change
10/37 = 27%	Idle (ChangeFreqCount LowerThan(0.05))
8/37 = 22%	Pulsars (Freq HigherThan(0.1))
18/37 = 49%	Supernova1 (HigherStep HigherThan(7))

Figure 5.18: Shape of change of Experiment 3 suspects in *Jun* case-study

GodClass Suspects	LOC Instability
9/14 = 64% of GodClass Suspects	Top(5%)
12/14 = 86% of GodClass Suspects	Top(10%)
14/14 = 100% of GodClass Suspects	Top(25%)

Figure 5.19: LOC instability of Experiment 1 suspects in *Jun* case-study

5.3.4 Adding the Time to GodClass Detection Strategy

There is a great ratio (i.e. between 30% and 50%) of the detected suspects which have gained the GodClass property during their lifetime showing that this flaw is in significant measure an evolutionary flaw. In this way we can isolate some of the GodClass *false positives* which do not represent a maintainability problem. It is somehow strange that almost half of the GodClasses were born with this disease meaning that they were *designed* to be very large and important classes. GodClasses which have this property over their entire lifetime might encapsulate some of the *essential complexity* [Bro87] of the modeled domain.

The classes under scrutiny have a very high level of instability both (LOC and NOM). The real GodClasses are usually among the first 20% of the most instable classes. This is in great measure due to the fact that GodClasses have a great amount of the system's knowledge thus they are centers of changing. By measuring the instability we can eliminate some of the *false positives* which appear partially due to the inaccuracies introduced by the McCabe's cyclomatic complexity computed for Smalltalk systems. For an in depth discussion on the drawbacks of the cyclomatic complexity please refer to Section B.1.3.

It is interesting that GodClass suspects' history can be composed of a period when the class was a *pulsar* and a period when it was a *supernova* [Lan01a].

Out of the measurements presented above we think it is meaningful to add the time to the current way of detecting the design flaws. Table 5.22 summarizes the results obtained so far. We can notice that the number of false positives decreased with a rate between 30% and 100%. The price payed was the increase of the *false negatives* detected. However the *false negatives* obtained represent no maintainability problems as they proved to be very stable during their life-time. Thus we can extend the classic *GodClass* detection strategy to *Timed-GodClass* and *InstableGodClass* which have the following expressions given in

SOD [Mar02]:

```
TimedGodClass:=
        ((ATFD, TopValues(20%)) and (ATFD, HigherThan(10)))
                  and ((WMC, HigherThan(40)) or (TCC, LowerThan(0.15))
      HistoryToVersionMatrixCollection
        (ApparitionFreqCount, LowerThan(70%)))


InstableGodClass:=
        (LOCInstability, HigherThan(33%))
                  or (LOCInstability, TopValues(10%))
      HistoryToLastVersionAdapter
        ((ATFD, TopValues(20%)) and (ATFD, HigherThan(10)))
                  and ((WMC, HigherThan(40)) or (TCC, LowerThan(0.15))
```

Case Study	Detection Strategy Name	No of suspects	No of flaws	No of False Pos	No of False Neg
Classifier (Experiment 2)	GodClass	4	1	3	0
	TimedGodClass	1	1	0	0
	InstableGodClass	4	1	3	0
Conan (Experiment 3)	GodClass	6	2	4	0
	TimedGodClass	1	0	0	2
	InstableGodClass	4	2	2	0
Jun (Experiment 3)	GodClass	37	?	?	?
	TimedGodClass	12	?	?	?
	InstableGodClass	20	?	?	?

Figure 5.22: Time Based GodClass conclusions

Conclusions: TimedGodClass finds the evolutionary GodClass suspects. The suspects which were GodClass suspects from their birth could represent the inherent complexity of the modeled domain. If they are stable they represent no maintainability problem so they are not malign classes. InstableGodClass gives the maintainability problems for classes from the system so the more instable a GodClass the more dangerous.
These two aspects are quite independent and they can be naturally combined. In the following we present two possible combinations:

- TimedGodClass Or InstableGodClass - will most comprehensively catch the problems related to GodClasses. While it reduces the number of *false negatives* it will increase the number of *false positives*. ‘
- TimedGodClass And InstableGodClass - captures the two dangerous aspects related to GodClasses. A class which evolved into a GodClass is less likely to represent a essential piece of complexity of the system. If

this class is also instable it can be a real threaten to system maintainers. This combination reduces the number of *false positives* but increases the number of *false negatives*.

5.3.5 A Comparison Between TimedGodClass and TimedDataClass

MOTIVATION: As DataClass and GodClass design flaws are somehow complementary (i.e. DataClass being small and dumb data carriers, GodClass being large classes which tend to centralize the knowledge of the system) in the following we will show a comparison between their evolutionary behavior. This comparison is structured as comparative answers to the questions which drove this part of the work.

QUESTION: How does the time affect the *DataClass* and *GodClass* status for a class ?

`Answer for DataClass:` We found that *DataClass* is a characteristic mostly (i.e. more than 90% of the time) linked to the birth of the class. This shows somehow surprisingly that *DataClass* is a flaw highly correlated with the initial design.
`Answer for GodClass:` Many of the *GodClass* (i.e. almost 30%) have this characteristic gained by evolution. This indicates that many *GodClass* are due to the accumulation of complexity/knowledge during the system's evolution. Some of the birth *GodClass* might be false positives as they could model some essential complexity[Bro87] of the modeled domain.

QUESTION: In what measure the suspects are stable/instable in comparison with the other classes ?

`Answer for DataClass:` As our intuition tell us, the DataClass classes proved to be stable in comparison with the other classes. For a comprehensive explanation of the way how we computed the stability please refer to A.2.
`Answer for GodClass:` As shown in the *Jun* case-study the majority of GodClasses are among the *Top 10%* of the most instable classes of the system.

QUESTION: In what measure the real flaws are stable/instable in comparison with the other suspects ?

`Answer for DataClass:` The more *DataClass* a suspect, the lighter and more dumb it is. Thus our intuition tells that it should be more stable and this was confirmed by the experiments.
`Answer for GodClass:` The more *GodClass* a suspect the more heavier and less stable it is.

QUESTION: What is the shape of change of the suspects ?

`Answer for DataClass:` *DataClass* are characterized in the great majority of time by being *Idle*.
`Answer for GodClass:` Both *Supernova* and *Pulsar* shapes of change appear recurrently among the *GodClass*. Here we made an interesting observation that more constrained GodClass suspects have these shapes of change much more emphasized in comparison with the other suspects.

QUESTION: Can we correlate the shape of change with a certain subtype of suspects ?

`Answer for DataClass:` Due to their high stability the great majority of the DataClass suspects are *Idle*.
`Answer for GodClass:` *Idle* GodClasses reveal the inherent complexity of the modeled domain, raising no maintainability problems. On the other hand *Pulsar* GodClasses are those who are malign because their shape reveals that there were modified a lot during their lifetime and thus the abstractions they model were not properly chosen. *Supernova* GodClasses are in the middle. If the functionality was added very early in their life and no major modifications were made after that then they are comparable to *Idle* GodClasses. The later the functionality was added, the more probable is that was a result of a refactoring and thus the class is malign.

5.3.6 A Taxonomy For Time Based Suspects

Out of the experiments we did on the behavior of classic detection strategies during their lifetime, we propose the following taxonomies. The first refers to the period in which a suspect has that property. The next refers to the shape of change of the suspects.

1. **Design DataClass** - are those entities which were born DataClass and retain this quality over their entire life. This kind of suspects represents the majority of the DataClasses and are very interesting because they empirically connect the DataClass flaw with the system design stage.

2. **Evolutionary DataClass** - are entities which became DataClass suspects as a result of the system's evolution. They are the effects of some refactorings applied to the system.

3. **Design GodClass** - are the entities which have been GodClass suspects from the very beginning of their lifetime. These could happen if the GodClass encapsulates an *essential complexity*[Bro87] and in this way the suspect is very stable over its life. The other possibility would be that the suspect is a real design flaw and in this case it will be highly instable over its lifetime.

4. **Evolutionary GodClass** - are those which are the results of the system evolution. They are the outcomes of the complexity accumulation in the system due to the refactorings made.

The next classification is a result of combining the Lanza's taxonomy [Lan01a] with Marinescu's detection strategies [Mar02].

1. **IdleDataClasses** - are the most common kind of DataClass suspects. In his case they are identical with the *Design DataClasses*.

2. **IdleGodClasses** - are a subset of *Design GodClass*. These ones are harmless as they caused no maintainability problems. They can be inherent complexities of the modeled domain.

3. **PulsarGodClasses** - are the most dangerous ones because they reveal maintainability problems reflected by addition and subtraction of functionality.

4. **SupernovaGodClasses** - classes which suddenly grew as a result of the functionality addition. This growth can be the result of some refactorings applied and can be correlated with the apparition of other design flaws (e.g. DataClass).

5.4 Quality Defined Through Evolution

Time based detection strategies are also useful means to detect new problems. The problems which inherently require the time information for their detection were presented in Section 4.2.In this section we will evaluate how effective this approach is for the detection of time-based flaws.

We will present our results for the flaws[1] stated in Section 4.2 in a set of tables. All tables have the same layout: on the first column we have the group of entities found as results, on the second column we have the number of entities that forms a group and finally on the last column we have the versions in which the changes occurred. Each of the entities from each result group have a star associated with it if it was found to be referred from another entity and a plus assigned if it refers another entity. If in a result group appear both the class and its class side then both receive a star when they are referred (i.e. we make no distinction between a call to the class side and an instantiation). At the beginning of each table we present the smaller sets of entities that changed together in more versions. In the lower parts of the tables we increase the size of the groups as we decrease the number of versions in which the change happened. To manually gather cross-reference information we used the VisualWorks source code browser. After each table we discussed the results in a paragraph of comments.

[1] Using the our case-studies we could not find any meaningful result for *DivergentChange* operator.

As shown in Section 4.2, the concrete meaning of *entities that change together* varies with the problem which is aimed to be solved (e.g. for 'Parallel Inheritance Hierarchies' the property that changes together is 'Number of children').

Driving Questions During the experiments we will try to answer the following questions:

1. How many of the links found between entities are accidental ?
2. Can we narrow down the number of accidental links ?

The difference between the first version in which the change appear and the last version shows the period of the system's life-time between which the concurrent changes occurred. The larger the period, the more credible the connection is. Entities that change together should deal with recurrent problems - recurring referring to time. If changes occur only over a short period then the problem found can not be considered recurrent any more.

5.4.1 Shotgun Surgery Experiments

We considered here the ShotgunSurgery as the same time variation of NOM. For a comparative discussion about LOC and NOM please refer to Paragraph B.2.2.

Conan Case-study

Comments: The manual inspection was done on versions 1.60 (because TechniqueForConceptAnalysis class is not present in 1.102) and 1.102. Out of the results presented in Table 5.23 the majority of the found groups have links between their members. However we found that many of the groups whose change period span over a shorter amount of time (they are also marked in the table above with 'No' in the 'Number of changing entities' field) are not coupled confirming the speculation that the period of time is important for the elimination of the 'Yesterday Weather' noise which was described in Paragraph 5.2.3.

Jun Case-study

Comments: The manual inspection was done on Jun195. The result of the manual inspection over the first set of results proved that JunOpenGL3dObject reffers all the other classes. From the second set of classes JunBody and JunBody_class instantiates all other classes. In the third set, the class JunOpenGL3dObject_class refers all other classes. As showed above, in this case-study the general trend is that a class from each group instantiates all other classes. That class is very probable to be the trigger of changes from the other classes. We observe that all the above changes span over a long period of the analyzed project and as we found links between all the entities from all groups, our supposition that the spanning period tends to validate that the same time changes are not coincidences is confirmed.

Shotgun Surgery - Conclusions

We must stress here that we are aware that not only class instantiations are valid links between classes because other, lighter weighted relations can exist (e.g. from the methods of a class we can call methods of another class). In the above presented experiments we focused only on one type of links between classes and we ignored the other ones (a more comprehensive list of possible links is presented in Paragraph 5.2.3). The result of this is that there might be many linked entities which we erroneously considered to be completely independent. This implies that the real results are better than in our experiments.

We can integrate the ShotgunSurgeryOperator in the detection strategy mechanism as shown in the following:

```
ShotgunSurgeryDS := (MNV HigherThan(4)) and (MNE, HigherThan(3))
                    HistoryToResultGroup
                      (ResultsVersionDifference, HigherThan(20))
```

The variables which appear in the detection strategy are:

- **Minimum Number Of Changing Versions (MNV)** - the minimum number of versions in which the entities were modified together
- **Minimum Number Of Entities That Change Together (MNE)** - the minimum number of entities from the result group
- **ResultsVersionDifference** - the difference in time between the first time when the change appeared and the last time.

5.4.2 Parallel Inheritance Hierarchies Experiments

Conan Case-study

Comments: We encountered two kind of noises related to Smalltalk class hierarchy - the fact that it is single rooted and that it is parallel with the meta-class hierarchy (for more details about the concerns related to the Smalltalk language please refer to Section A.1).

The results show us a good thing - that at the beginning of the system's life, the tests were kept consistent with the source code.

Jun Case-study

Comments: The thresholds represent two extreme cases: the first set of thresholds look for relatively few entities that change many times and the second set searches larger groups of entities that change fewer times (we obtained here 5 groups but we presented only the most relevant three). From the above results we can easily see the MVC framework at work. Thus the empirical results show that the links between entities are not at random, resulting that in this case the noise is at minimum.

Parallel Inheritance Hierarchies - Conclusions

It is obvious from the results above that the resulted groups are not at random. In face the results revealed the usage of two frameworks: SUnit framework and MVC. However we could not find problems with the above parallel inheritance hierarchies as neither of them indicate a form of "ShotgunSurgery" (for the description of the malign case of "Parallel Inheritance Hierarchies" please refer to Section 4.2.2).

Even if in the above experiments we could not find malign usages of "Parallel Inheritance Hierarchies" we demonstrated that our approach is scalable and accurate. We strongly believe that the only problem against finding flaws is represented by the chosen case-studies which don't have this kind of flaw.

A similar detection strategy like that defined in the ShotgunSurgery conclusions section of this chapter can be defined for including the distance between versions in the flaws detection process.

5.4.3 Speculative Generality Experiments

Jun Case-study

Comments: We restricted our analysis to using only 25 versions (i.e. starting with Jun005 and ending with Jun125). We found that *JunSequence* was born in the first analyzed version as an abstract class and until version Jun125 it had only one child.

Speculative Generality - Conclusions

We could successfully identify an instance of the *SpeculativeGenerality* flaw. Again, the approach proved to be scalable and accurate. The detection strategy that we used is presented in the following:

```
SpeculativeGeneralityDS:=
        (NOC For Abstract Classes, LowerThan(2))
      HistoryToVersionMatrixCollection
          (RelativeFreq Count, HigherThan(0.99))
        and
          (AbsoluteFreq Count, HigherThan(5))
```

5.4.4 Conclusions

The system history show what happened during the life-time of the system. This approach goes beyond speculations about possible flaws inherent when only a single version is analyzed. The major problem of our technique, the *false positives* proved to be easy to overcame when increasing the number of versions in which the change appeared and the distance in time between the first version and the last version in the case of 'entities that change together' like strategies.

Entities Groups	**Number of changing entities**	**Number of versions and versions number**
	2	4
ConanApp * InternalMSEClassBehavior_class +		1.75, 1.85, 1.95 1.100, 1.102
	2	3
InternalMSEClassBehavior * InternalMSEClassBehavior_class +		1.60, 1.85, 1.95
ConceptAnalysisContext InternalMSEClassBehavior_class		1.60, 1.80, 1.95
ConceptAnalysisContext + BinaryMappingSet *		1.55, 1.60, 1.90
InternalMSEClassBehavior_class ConceptAnalysisConcept	No	1.80, 1.95, 1.100
InternalMSEClassBehavior BinaryMappingSet		1.55, 1.60, 1.85
InternalMSEClassBehavior ConceptAnalysisObject	No	1.55, 1.60, 1.70
MSEMethodObject_class MSEAttributeObject_class		1.25, 1.30, 1.70
ConanApp * InternalMSEClassBehavior +		1.70, 1.85, 1.95
ConceptAnalysisConcept * ConceptAnalysisContext +		1.40, 1.80, 1.95
TechniqueForConceptAnalysis Predicate		1.15, 1.40, 1.55
	3	3
TechniqueForConceptAnalysis + ConceptAnalysisObject ConceptAnalysisContext *	No	1.40, 1.55, 1.60
MammalAnimalsTest + InternalMSEClassBehavior ConceptAnalysisContext *		1.55, 1.60, 1.95

Figure 5.23: ShotgunSurgery - Conan case-study experimental results

Entities Groups	**Number of changing entities**	**Number of versions and versions number**
	4	5
JunOpenGL3dCompoundObject *+ JunOpenGL3dPolygon ** JunOpenGL3dObject * JunOpenGL3dObject_class +++*		020, 070, 075 095, 100
JunEdge ** JunVertex ** JunLoop ** JunBody +++* JunBody_class +++*		010, 110, 140, 145, 150
JunOpenGL3dPolygon * JunOpenGL3dPolyline * JunOpenGL3dObject * JunOpenGL3dObject_class +++*		010, 020, 070 075, 095

Figure 5.24: ShotgunSurgery - Jun case-study experimental results

Entities Groups	**Number of changing entities**	**Number of versions and versions number**
	2	3
Object TestCase		1.20, 1.35, 1.40

Figure 5.25: Parallel Inheritance Hierarchies - Conan case-study experimental results

Entities Groups	**Number of changing entities**	**Number of versions and versions number**
	4	7
ApplicationModel ApplicationModel_class View View_class		065, 075, 080, 090 160, 165, 170
ApplicationModel ApplicationModel_class Object Object_class		010, 050, 065, 090 100, 140, 160, 165 175, 200
	6	3
Object Object_class ApplicationModel ApplicationModel_class JunOpenGLDisplayModel JunOpenGLDisplayModel_class		140, 165, 175
Object Object_class ApplicationModel ApplicationModel_class Controller Controller_class View View_class		065, 160, 165
JunOpenGLDisplayModel JunOpenGLDisplayModel_class Controller Controller_class View View_class		080, 114, 165

Figure 5.26: ParallelInheritanceHierarchies - Jun case-study experimental results

Entities Groups	**NOC LowerThan**	**Relative Freq Count HigherThan**	**Absolute Freq Count HigherThan**
	2	0.99	10
JunSequence			

Figure 5.27: Speculative Generality - Jun case-study experimental results

Chapter 6

Conclusion and Future Work

The first section of this chapter summarizes the work done during the diploma project. In the following section we evaluate the results obtained by using the analysis mechanism using our tool named *V-Soda*. In the end of the chapter we draw possible directions for future work.

6.1 Summary

As presented in Section 1.1 the *main goal* of this work is to combine the information obtained from evolution of software systems with the current ways of detecting object-oriented design flaws. We divided the above goal in two parts: the first is to *improve* the current way of detecting design flaws and the second is to *extend* the set of design flaws which can be detected with new flaws that inherently require time information.

In order to achieve our aims we defined a new, evolution based mechanism for detecting flaws, named "Time Based Detection Strategies". In order to check our new mechanism, we built a tool named *V-Soda*[1] which uses it for design flaws detection.

The "Time Based Detection Strategies" were used in two main directions which are described below.

1. **Quality Refined Through Evolution** aimed to improve the detection of design flaws by adding evolution information to the classic, version-based detection mechanism. We focused on the detection of *GodClass* and *DataClass* design flaws and identified their specific behavior over

[1] **Soda** is the shortcut for 'Smalltalk Object-oriented Design Analyzer'. **V-Soda** is the short-name for Versions-based Soda

time. We were interested in which degree the suspects of these flaws were stable/instable during their life and we summarized our results with a comparison of the behavior in time of *GodClass* and *DataClass* suspects. In order to describe the behavior of these two flaws over time we developed a higher level language by using the taxonomy developed in [Lan01a]. (Section 3.2.1).

2. **Quality Defined Through Evolution** uses the evolution of a system as the main tool for finding several time-based flaws. We proposed new methods to identify some of the 'bad smells' defined in [Fow99] and which require information about the evolution of the system. The 'bad smells' which can be detected with the help of our mechanism are: *Shotgun Surgery*, *Parallel Inheritance Hierarchies*, *Divergent Change* and *Speculative Generality*.

6.2 Evaluation of Contribution

This section is meant to evaluate the concepts proposed, the general approach and the tool support. Each of the paragraphs below will concentrate on a specific characteristic of our work.

Effectiveness of the concepts: *Time-based detection strategies* proved to be effective in the process of design flaws detection. The results from our experiments presented in Chapter 5 showed that the time information integrated in the old detection strategy mechanism enhances its capabilities of detecting the 'bad smells'.

Accuracy of the approach: We are interested here in the number of 'false positives' and 'false negatives' found during the detection process. In the first part of the validation chapter (see Chapter 5) - 'Quality Refined Through Evolution' (Section 5.3) - the number of suspects found after an analysis was dropped down with 30% in the case of *DataClass* flaw and with 50% in the case of *GodClass* flaw. The price paid for the reduction of number of suspects is represented by the new *false negatives* which appeared. In the second part of the validation chapter (see Chapter 5)- 'Quality Defined Through Evolution' (Section 5.4) - our manual inspection proved that more than 60% of the found suspects have obvious links between them.

Scalability of the approach: We successfully succeeded to Analyze systems spanning from 50 to 400 classes (for further details about our case-studies please refer to Section 5.2.1). Furthermore, we used in our analysis a considerable number of versions (i.e. between 20 and 40). The experiments were done on relatively slow systems (e.g. AMD Athlon 600MHz with 384Mb of RAM) which proves that using our approach we can analyze systems of industrial size.

Usability of the developed tool Our main aim in building the tool was to overcome the problem of thresholds which is inherent to the detection strategy mechanism (for more details on this subject please refer to Section 3.1.2). The graphical interface enables the user to easily change the thresholds of detection strategies used. By making the thresholds easy to customize we reduce the importance of their particular values. Using our tool the user can change the thresholds in real-time based on his/her feedback from the resulted suspects. The user has also the possibility to inspect the results given by applying a sub-expression of the detection strategy and in this way s/he can track the results.

6.3 Future Work

In this section we will present the future plans for extending the current work. We identified several directions in which the extensions can be made. In the first two sub-sections we propose extensions related to the main parts of this work: *Quality Refined Through Evolution* and *Quality Defined Through Evolution*.

6.3.1 'Quality Refined Through Evolution' Extensions

Extending the Current Approach to Other Version-based Detection Strategies

In [Mar02] is defined a larger set of detection strategies to identify other design flaws. *DataClass* and *GodClass* were chosen in this work only as examples of the ways in which the version-based detection can be improved by using the time information. We believe that it would be meaningful to enrich with time information other version-based detection strategies. For example the instability in the case of 'LackOfState' [Mar02, p.151] suspects would show the maintenance problems related to addition or modification of the states from the FSM they implement.

Using the 'Class Blueprint' for Refining the Evolution

The stability could be measured for each of the layers of the 'Class Blueprint' [Lan01b]. For example the GodClass suspects whose *interface layers* grow are dangerous as they tend to accumulate even more the control of the system. On the other hand, the maintainability of GodClass suspects whose *implementation layers* grow become harder and harder.

Size vs. Complexity Relation

The variation of the relation between the size of an class and its complexity can be a measure of its maintainability. Let R_c denote the growth rate for the complexity of a class and R_s denote the growth rate for the size of a class. In an ideal situation $R_c \leq R_s$. The higher the R_c beside R_s the more maintainability problems the class will have.

6.3.2 'Quality Defined Through Evolution' Extensions

Combining 'Single-Version Analysis' with 'Multiple-Versions Analysis'

We can use the classic detection strategies and the time-based ones as two independent ways for identifying the same flaw. For example we have definitions of both classic and time-based detection strategies for identifying the ShotgunSurgery suspects. In a similar way we could improve the detection of 'Interface Segregation Principle' by using the version-based detection strategy for its identification [Mar02, p.150] and 'Divergent Change' detection strategy (Section 4.2.3).

Bibliography

[91291] ISO/IEC 9126. Information technology - software product evaluation - quality characteristics and guidelines for their use, 1991. International Standard.

[Ben01] Bennett, K. and Rajlich, V. Software Maintenance and Evolution: A Roadmap. *In Proceedings of the conference on The future of Software engineering*, pages 73–87, 2001.

[BK95] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Symposium on Software Reusability*, pages 259–262, 1995.

[BR88] V. Basili and D. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Softw. Engineering*, 14(6):758–773, Jun, 1988.

[Bro87] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, Vol. 20:10–19, 1987.

[CDS86] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, Inc, 1986.

[Chi94] Chidamber, S.R. and Kemerer, C.F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander 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)*, 2000.

[DT03] Iulian Dragos and Adrian Trifu. Strategy based elimination of design flaws in object-oriented systems. *ECOOP Workshop on Reengineering*, 2003.

[Fow99] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley. Object Technology Series, 1999.

[GDM03] T. Girba, S. Ducasse, and Lanza M. Measuring the evolution of changes in object-oriented systems. *To be submitted at ICSE 2004*, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, 1st edition.* Addison-Wesley Pub Co, 1995.

[Gir03] Tudor Girba. Analyzing the history of object-oriented systems to understand changes, 2003. Internal Report.

[HS95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice Hall, 1995.

[Lan01a] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, page to be published, 2001.

[Lan01b] Lanza, Michele and Ducasse, Stephane. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. *In OOPSLA 2001 proceedings*, 2001.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[Li,93] Li, W. and Henry, S. Object-Oriented Metrics that Predict Maintainability. *The Journal of Systems and Software*, 23:111–122, November 1993.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics. A Practical Guide.* Prentice Hall Object Oriented Series, 1994.

[Mar] Robert Martin. Design principles and design patterns. http://www.objectmentor.com.

[Mar99] Radu Marinescu. A Multi-Layered System of Metrics for the Measurement of Reuse by Inheritance. *Proceedings of the TOOLS-Asia 31, Nanjing, China*, page 146, 1999.

[Mar01] Radu Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition*, pages 173–182, 2001.

[Mar02] Radu Marinescu. *Measurement And Quality In Object-Oriented Design.* PhD thesis, University "Politehnica" from Timişoara, December 2002.

[McC] Hays McCormick. Antipatterns - a brief tutorial. http://www.antipatterns.com/briefing/index.htm.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, December:308–320, 1976.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall PTR, 1997.

[Mih03] Petru Mihancea. Optimizarea detecţiei automate a carenţelor de proiectare in sistemele software orientate pe obiecte. Master's thesis, "Politehnica" University from Timişoara, June 2003.

[Par94] David Parnas. Software aging. *Proceedings of the16th International Conference on Software Engineering*, pages 279–287, 1994.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics.* Addison-Wesley Pub Co, 1996.

[Som01] Ian Sommerville. *Software Engineering, Sixth Edition.* Addison-Wesley Pub Co, 2001.

[Szy98] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, 1998.

[vGB01] Jilles van Gurp and Jan Bosch. Design erosion: Problems & causes. *Journal of Systems & Software*, 61(2):105–119, 2001.

Appendix A

Implementation concerns

The purpose of this chapter is to present the low level problems which we encountered during the implementation and while making the experiments. We start the current part by presenting the specific concerns related to *Smalltalk* language, continue with a presentation of the concerns related to *Moose* metamodel, followed by the extensions added to the classic detection strategy mechanism in order to integrate the time and we end with a short presentation of the 'Entities that change together' algorithm.

A.1 Smalltalk Programming Language Concerns

In this section we will describe the specific problems which arose during our work with Smalltalk. The section is divided in two parts: the first part describes how did the particularities of analyzing Smalltalk systems influence the computation of some of the metrics we used; the second part shows the influences on the computation of 'Entities That Change Together' (Section 4.2).

A.1.1 Metrics Computation Concerns

Some metrics which have already been defined have a totally different implementation in Smalltalk compared with Java or C++ where they were firstly targeted. The particularities of Smalltalk language which affect the current implementation of the metrics along with the specific metrics they influence are:

Dynamic Typing - this feature prevents us from knowing accurately the target method of a call. We can only use a probabilistic measure for called methods and this strongly affects the 'Access To Foreign Data' (ATFD) metric. For the definition of this metric and the implementation details please refer to Section B.1.5.

Polymorphism - the Smalltalk methods are usually highly polymorphic (there

are many methods belonging to different classes having the same names and signatures). This fact, in addition to dynamic typing, increases more the number of candidates for a target call, and affects further the definition of the 'Access To Foreign Data' (ATFD) metric (Section B.1.5).

Access specifiers - the *public* and *private* methods are only a matter of convention in Smalltalk. The methods from the 'public' protocol constitute the public interface of a class. The methods from the 'private' protocol should not be called from outside of the class. However there are another methods which are not 'public' but can be called from outside of the class. (e.g. accessor methods, initialization methods) - this affects the 'Weight Of a Class' (WOC) metric. For the definition of this metric and the implementation details please refer to Section B.1.3.

Accessor methods - all data in Smalltalk is *private*, so accessor methods are required even for subclasses that access data from their superclass. Furthermore there are additional accessor methods for working with collections (e.g. 'aCollection do: aBlock') which are not classic accessors and are not counted in the 'Number Of Accessor Methods' (NOAM) metric (Section B.1.2). This affects the 'Weight Of a Class' (WOC) metric (Section B.1.3) also which tends to be much higher.

Method size - the reuse unit in Smalltalk are methods which tend to be very small. In Smalltalk a class has a larger number of smaller methods than in other common object oriented programming languages (e.g. C++). The big number of methods affects 'Weight Of a Class' (WOC) metric (Section B.1.3) and their size affects McCabe's cyclomatic number and 'Weight Methods Count (WMC) metric (Section B.1.3) as a method has usually a low cyclomatic complexity.

The big number of methods together with the relative small number of attributes (e.g. especially for meta-classes) from a class influence the computation of TCC (Tight Class Coupling) metric which tends to be very small. This affected the *GodClass* detection strategy definition where this metric is supposed to reveal how disparate (unlinked) the methods of a big class are. In order to overcome this problem, we restricted the computation of TCC metric only for classes with a certain number of attributes.

Conditionals and Loops - the conditionals in Smalltalk are implemented as messages sent to class *Boolean*. In a similar fashion, the loops are implemented as messages of class *BlockClosure* and *Number*. In this way, the counting of *conditionals* and *loops* (e.g. for cyclomatic complexity computation - Section B.1.3) is more difficult and inaccurate.

A.1.2 Entities that Change Together Problems

"Single Root" Class Hierarchy - all classes from Smalltalk class hierarchy, except one (i.e. Object), must have a superclass. This increase the value

of metrics such as: DIT (Depth of Inheritance Hierarchy) and raises new problems when computing "Parallel Inheritance Hierarchies" (i.e. the class 'Object' change its number of children many times).

The class vs. meta-class hierarchies - when we extend a class A with a class B the class A_class is automatically extended by class B_class. Thus the meta-class hierarchy is parallel with the class hierarchy resulting that the number of entities from a parallel inheritance hierarchy group is bigger.

A.2 MOOSE concerns

We interact with MOOSE [DLT00] through 'history meta-model' (Hismo) [Gir03]. As shown in Figure A.1 the implementation of 'Time Based Detection Strategies' is based on Hismo and Moose. Hismo adds the time to the entities of a classic meta-model (in this case the classic meta-model used is Moose). Thus we interact with Hismo for time-based entities (e.g. class histories) and with Moose for version based-entities (e.g. classes).

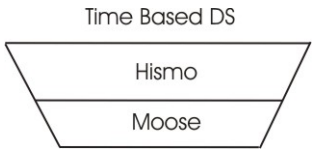

Figure A.1: The infrastructure for time-based detection strategies

While trying to find DataClasses we usually get only a few. Taking a closer look to the metrics as they are defined in MOOSE for Smalltalk systems, we notice that there are some particularities among which is the computation of 'Number Of Pure Accessors' metric.

Number Of Pure Accessors - as defined in Moose (i.e. accessors methods which have only one line of code) very few methods are pure accessors - some do a bit of initialization (e.g. lazy initialization), etc. We provided a new implementation which doesn't consider any more the number of lines of the accessor methods.

Stability vs. Instability in Moose Models for Smalltalk While measuring stability we encountered a lot of noise because many classes have a very small size due to the fact that all meta-classes for all classes of a analyzed Smalltalk system are imported in MOOSE even if these meta-classes don't have any methods or attributes (or have very few). These classes are obviously very stable (there is nothing to be changed). To overcome this problem we measure the stability of a certain class only in comparison with the stability of non-empty

classes.

A.3 Additional Time-Based Detection Strategies Operators

During the implementation of time based detection strategies we were faced with the problem of adapting the two-dimensional set of input data of the time-based entities to the classic, unidimensional input set. This will lead to the possibility to combine more detection strategies not necessarily of the same type (e.g. combine time-based with classic detection strategies - PulsarGodClass). This is a possibility to implement the shape of change.

The predefined adaptation operators presented in the following will include the semantics of the current *And operators* - the final result is computed as the intersection of the partial results obtained from the left expression and right expression.

Some proposed filtering operators are:

- HistoryToClassCollection transforms a "ClassHistoryGroup" into the collection of classes belonging to a specific version. (for example this way we can get all "Pulsar Classes" which were DataClass in a specified version of the system.) In Figure A.2 is presented a particular case of this operator (HistoryToLastVersionOperator) which transforms a collection of histories into a collection of classes belonging to the last version of the system. We can observe from this figure that class A doesn't belong to the result set as its history ends in version n-1. An example showing how easy and naturally becomes to build composed detection strategies - we took as example the *PulsarGodClass* detection strategy - is presented in Figure A.3.

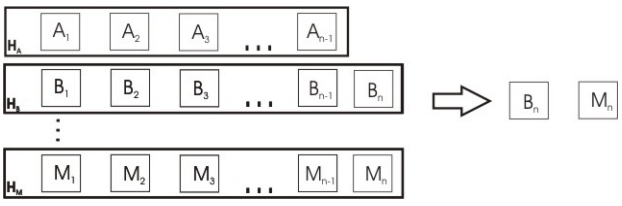

Figure A.2: History To Last Version Adapter

- HistoryToBinaryMatrix transforms a "ClassHistoryGroup" into a matrix with binary elements - each element of the resulting matrix corresponds to a version of a class from the input set. The resulting matrix's

elements have value 1 if the corresponding version of the class has a certain property and 0 otherwise.

`Example` This can be used to compute easily the TimedDataClass suspects. The input collection will be sliced into versions. On each version will be applied the DataClass detection strategy and the results will be collected into a matrix whose lines would represent the entities and columns will be 1 if the specified class was a DataClass and 0 otherwise. In Figure A.5 we present a particular case where class A is DataClass in versions 1,2,4; class B is DataClass in versions 3,4 and class C is not a DataClass in any version. In this figure we underlined the classes with DataClass property. The code for building the TimedDataClassDS is given in Figure A.4.

```
pulsarDS := self getDetectionStrategyNamed: #buildPulsarDS.
godClassDS := self getDetectionStrategyNamed: #buildGodClassDS.
pulsarGodClassExpression := DSHistoryToLastVersionAdapter new
                              left: pulsarDS expression
                              right: godClassDS expression.

pulsarGodClassDS := DetectionStrategy
                      new: 'Pulsar God Class'
                      withExpression: pulsarGodClassExpression
                      withInputBlock: [
                        VanSystemManager uniqueInstance defaultSystem
                             classHistories selectNonLibraryClassHistoryGroup
                      ].
```

Figure A.3: HistoryToLastVersionAdapter usage for PulsarGodClass

```
dataClassDS := self getDetectionStrategyNamed: #buildDataClassDS.

"freqCountBlock applied to a collection counts the relative
number of non-null values"
higherFrequencyThan := (DSHigherThanOperator withName: 'FreqCount HigherThan')
                         initialize: freqCountBlock
                         with: 0.1.
timedDataClassExpression := DSHistoryToBinaryMatrixAdapter new
                              left: dataClassDS expression
                              right: higherFrequencyThan.
timedDataClassDS := DetectionStrategy
                      new: 'Timed Data Class New'
                      withExpression: timedDataClassExpression
                      withInputBlock: [
                        VanSystemManager uniqueInstance defaultSystem
                              classHistories selectNonLibraryClassHistoryGroup
                      ].
```

Figure A.4: HistoryToBinaryMatrixAdapter usage for TimedDataClass

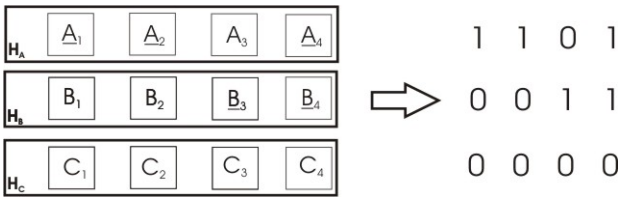

Figure A.5: History To Binary Matrix Adapter

A.4 Detecting the Entities that Change Together

For the computation of 'Entities That Change Together' we iteratively build a matrix whose lines are associated to groups of entities (i.e. classes or methods) and whose columns are associated to the versions of the analyzed system. An element (i,j) of the matrix has the value of 1 if the entities corresponding to `Group i` change together in `Version j`. The algorithm for building the matrix is given in the following as a succession of steps. Beside the group of histories this algorithm has as additional input two values: 'Minimum number of changing versions' (MNV), and 'Minimum number of entities that change together' (MNE). The partial matrices built are named M_i with (i$\geq$0).

1. Compute Initial Matrix - in this step we build the initial matrix. Each line of this matrix corresponds to a group made of one entity. On each column j we have 1 if the entity changes a certain property between versions j-1 and j; and 0 otherwise. We will name the resulting matrix M_0.

2. Compute A New Matrix - this step has as result the computation of M_n (n>0) from matrix M_{n-1}. For each of the pair of lines (i,j) belonging to matrix M_{n-1} will correspond a line (k) in matrix M_n. The group of entities corresponding to the line (k) is the reunion of the groups of entities of corresponding to lines i and j from matrix M_{n-1}. Each column (l) of line (k) of M_n has as value 1 only if elements (i,l) and (j,l) of matrix M_{n-1} are 1 in the same time.

3. Compute Current Matrix - each line of the M_n matrix which has fewer than MNV values of 1 is eliminated.

This algorithm iterates through the steps 2 and 3 for MNE-1 times. MNE represents the minimum wanted size of the groups of entities obtained as results.

Appendix B

Metrics Description

Here we will describe the software metrics used in this work. Some of the metrics will be followed by a discussion about our experience gained while working with them. Even though a few metrics were already presented in Chapter 2 they will be also presented in this part for the sake of uniformity (i.e. in this way we make the current appendix a catalog of metrics which appeared throughout the diploma).

B.1 Version-Based Metrics

B.1.1 Project Metrics

These 'project metrics' are used for measuring the size of the individual versions of a system. In this work we used them for measuring the size of the first and the last version of the analyzed case-studies. (Section 5.2.1)

Number of Classes (NOC)

Definition: NOC represents the total number of classes which directly interact with the analyzed system.

Observation: This metric includes both the classes belonging to the system and the classes used in a way by the former ones and belonging to the library. In this way we can have an overview on the total size of the project. Please see also the definition of 'Number of Non-Library Classes'.

Number of Non-Library Classes (NONLC)

Definition: NONLC represents the total number of classes developed within the analyzed system.

OBSERVATION: In contrast to NOC, this metric doesn't include the classes belonging to the library. In this way the value of this metric can vary a lot beside the value of NOC (i.e. if we have few and big classes the difference between the two increases). Please see also the definition of 'Number of Classes'.

B.1.2 Size Metrics

We applied these metrics on classes and methods to asses their size.

Number of Instance Methods in a Class (NOM)

Definition: The total number of *instance methods in a class* counts all the public, protected, and private methods defined for a class' instances. [LK94]

OBSERVATION: In Smalltalk the equivalent of 'static' methods from C++ are the methods belonging to the class side of a class. These methods are instance methods of the meta-class classes.

Number of Lines of Code (LOC)

Definition: A line of code is any line of program text that is not a comment or a blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. [CDS86]

Number of Public Attributes (NOPA)

Definition: NOPA is defined as the number of non-inherited attributes that belong to the interface of a class.

OBSERVATION: Even if in Smalltalk all attributes are private our implementation can analyze systems written in other languages as Moose [DLT00] (our underlying meta-model) is language independent.

Number of Accessor Methods (NOAM)

Definition: NOAM is defined as the number of the non-inherited accessor-methods declared in the interface of a class. [Mar01]

OBSERVATION: We encountered some difficulties when we computed this metric both because of the particularities of Smalltalk and of Moose - these problems are described in Appendix A.

B.1.3 Complexity Metrics

These metrics were used throughout this work to measure the complexity of classes and methods.

Definition: Cyclomatic Complexity

Problems with the 'Cyclomatic Complexity': McCabe's Cyclomatic Complexity is meant to measure the complexity at the procedure level. Here are two aspects which are worth mentioning:

1. This metric does not measure the complexity generated by the coupling of a class as it is oriented mainly on the procedural code.
2. In Smalltalk the computation is less accurate as we don't have keywords for conditionals but only messages sent especially to `Boolean`, `BlockClosure` and `Number` classes. So, we can find complex methods which have cyclomatic complexity very small. (Section A.1). Furthermore the Smalltalk's extensive use of collections clutters even more the computation of cyclomatic complexity (e.g. in Smalltalk the selection of elements from a collection introduces no complexity as there is no classic 'loop' required for this).

Weighted Methods per Class (WMC)

Definition: Consider a class C_1, with methods $M_1, \ldots M_n$ that are defined in the class. Let $c_1, \ldots c_n$ be the complexity of methods. Then:

$$WMC = \sum_{i=1}^{n} c_i$$

[Chi94]

The complexity is not defined in the above definition. However the most common measure for the above complexity is the *cyclomatic complexity*. (in this work we used the 'cyclomatic complexity' as a particular measure for the complexity too)

Weight Of A Class (WOC)

Definition: WOC is the number of non-accessor methods in the interface of the class divided by the total number of interface members. [Mar01]

We implement this metric for Smalltalk by dividing the number of methods that are neither accessors nor private to the total number of non private methods. A Smalltalk method as is represented in Moose (the underlying meta-model [DLT00]) is not private if it doesn't belong to 'private' protocol.

B.1.4 Inheritance Metrics

Depth of Inheritance Tree (DIT)

Definition: Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree. [Chi94]

B.1.5 Coupling Metrics

Coupling metrics measure the degree in which two entities are coupled.

Access To Foreign Data (ATFD)

Definition: ATFD represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods. The higher the ATFD value for a class, the higher the probability that the class is or is about to become a god-class. [Mar02]

We compute ATFD for a class C by dividing the number of calls to methods with a certain name by the total number of methods from the analyzed system with that name which don't belong to the parent classes of C.

Changing Methods (CM)

Definition: CM is defined as the number of distinct methods in the system that would be potentially a affected by changes operated in the measured class. [Mar02]

Changing Classes (CC)

Definition: The CC metric is defined as the number of client-classes where the changes must be operated as the result of a change in the server-class.

B.1.6 Cohesion Metrics

Tight Class Cohesion (TCC)

Definition: TCC is defined as the relative number of directly connected methods. Two methods are directly connected if they access a common instance variable of the class. [BK95]

B.2 Evolutionary Metrics

In this section are presented the metrics which are related to the system's evolution. The notion of a 'history' [Gir03] which appears in the definitions given below was introduced in Section 3.2.2.

B.2.1 Project Metrics

We used these 'project metrics' for measuring the overall size of our case-studies.

Number of Class Histories (NOCH)

Definition: NOCH represents the total number of class histories which directly interact with the analyzed system.

OBSERVATION: Here are counted both the histories belonging of classes from within the analyzed system and the histories corresponding to library classes which interact with the system.

Number of Non-Library Class Histories (NONLCH)

Definition: NONLCH represents the total number of classes developed within the analyzed system.

OBSERVATION: This metric counts only the histories belonging of classes from within the analyzed system.

Evolution of Number of Classes (ENC)

Definition: The ENC shows the dynamics in the measured history in terms of added or subtracted classes. (adapted after [Gir03])

$$ENC = \sum_{i=2}^{n} \mid NOC_i - NOC_{i-1} \mid$$

Latest Evolution of Number of Classes (LENC)

Definition: This measurement is a weighted version of ENC, as it weights the changes with the distance from the last version of the history. The closer to the last version a change is, the more it counts in the result of this measurement. (adapted after [Gir03])

$$LENC = \sum_{i=2}^{n} \mid NOC_i - NOC_{i-1} \mid e^{i-n}$$

Evolution of Number of Non-Library Classes (ENNLC)

Definition: The ENNLC shows the dynamics in the measured history in terms of added or subtracted classes which are non-library. (adapted after [Gir03])

$$ENNLC = \sum_{i=2}^{n} \mid NONLC_i - NONLC_{i-1} \mid$$

Latest Evolution of Number of Non-Library Classes (LENNLC)

Definition: This measurement is a weighted version of ENNLC, as it weights the changes with the distance from the last version of the history. The closer to

the last version a change is, the more it counts in the result of this measurement. (adapted after [Gir03])

$$LENNLC = \sum_{i=2}^{n} \mid NONLC_i - NONLC_{i-1} \mid e^{i-n}$$

Average Number of Classes (ANOC)

Definition: ANOC is defined as the average number of classes during the analyzed life of the system.

$$ANOC = \frac{\sum_{i=1}^{n} NOC_i}{n}$$

B.2.2 Stability Metrics

These metrics are used for assessing the stability or instability of histories of individual entities.

Lines Of Code Stability/Instability

Definition: LOC stability represents the fraction of the life-time of a class in which the number of lines of code did not change.

$$LOCstability = \frac{\sum_{i=1}^{n} S_i}{n}$$

$$where\ S_i = \begin{cases} 1 & \text{if } LOC_i = LOC_{i-1} \\ 0 & \text{otherwise} \end{cases}$$

Observation: In a similar way we defined 'LOC instability' as the opposite of 'LOC stability'.

Number Of Methods Stability/Instability

Definition: NOM stability represents the fraction of the life-time of a class in which the number of its methods did not change.

$$NOMstability = \frac{\sum_{i=1}^{n} S_i}{n}$$

$$where\ S_i = \begin{cases} 1 & \text{if } NOM_i = NOM_{i-1} \\ 0 & \text{otherwise} \end{cases}$$

Observation: We define 'NOM instability' as the opposite of 'NOM stability'.

A Comparison Between LOC and NOM Stability A class is stable from the point of view of NOM metric when no methods are added to or taken from that class. In the same manner, a class is LOC stable if the number of lines of code from it remains constant. The difference between these two metrics is based on the different purposes for adding methods versus lines of code to a class. The modification of the number of methods is caused by functionality addition or subtraction and internal refactorings. The modification of the number of lines of code is caused by functionality addition or subtraction, internal refactorings, patches and bug fixes. Thus we measure the *NOM stability* as a mean to find the modifications in the interface of a class (we ignore methods visibility qualifiers) and *LOC stability* to find all modifications from a class.

Appendix C

Use-Case Description

Our tool is a part of a larger system called Van (Versions Analyzer) developed at Software Composition Group from Bern.

Due to the particularities of 'entities that change together' beside the other detection strategies our tool has two modules: 'Detection Strategies Manager' and 'Entities That Change Together Manager'. The usage of each of these modules will be described in a separate section with the help of **Use Cases**. We chose *Use Cases* for documenting our tool in order to facilitate a quick start of using it. The below use-cases cover all functionality of the program.

C.1 Preliminaries

C.1.1 Actors

We identified two actors which can interact with *V-Soda*.

- Expert - represents the person which has the knowledge to define a new detection-strategy skeleton.
- Developer - is the daily user of the tool. He has no special training to express the design flaws in terms of detection strategies and no Smalltalk knowledge is required - he is merely a user of a library of detection strategies.

C.1.2 Preconditions

The usage of this tool requires that the versions under analysis are loaded into Hismo (Section 3.2.2). The version based meta-models should contain the value of the metrics required by the particular detection strategies used.

C.2 Use-Cases for Detection Strategies Manager

This module corresponds to the classic detection strategies, the time based detection strategies used in 'Quality Refined Through Evolution' part and to 'Speculative Generality' strategy defined in the 'Quality Defined Through Evolution' part.

Use Case 1: Expert Adds A New Detection Strategy

1. Expert clicks on New DS menu
2. System opens the 'Name Request' dialog
3. Expert introduces the name of the detection strategy
4. System opens the 'Detection Strategy Browser'
5. Expert edits the new detection strategy
6. If Expert presses the 'Cancel' button then the system abandons the editing of the new detection strategy without modifying the current library of detection strategies.
7. If Expert presses the 'Ok' button then the current detection strategy is accepted
8. If Expert wants to save the new detection strategy
   (a) Expert presses the 'Save' menu item
   (b) System saves the selected detection strategy

Error Case: If during the Step 7 the text which builds the detection strategy is not a valid Smalltalk method then the Expert is asked to perform corrections on the code.

Use Case 2: Expert Changes A Detection Strategy

1. Expert selects a detection strategy from the list shown in the left panel
2. Expert presses the 'Browse' button.
3. System opens the 'Detection Strategy Browser' on the selected detection strategy
4. Expert changes the detection strategy
5. If Expert presses the 'Cancel' button then the system abandons the edit of the detection strategy without modifying it.
6. If Expert presses the 'Ok' then the detection strategy is accepted

7. If Expert wants to save the modified detection strategy
   (a) Expert presses the 'Save' menu item
   (b) System saves the detection strategy

Error Case: If during the Step 6 the edited text is not a valid Smalltalk method then Expert is asked to correct the input.

Use Case 3: Developer Loads An Analysis Policy

1. Developer clicks on 'Open New Policy' menu
2. System asks the name of the policy to be loaded
3. Developer inputs the name of the policy
4. System loads the policy and displays the detection strategies from it.

Use Case 4: Developer Runs A Detection Strategy

1. Developer selects the detection strategy to be applied
2. If the detection strategy chosen is version based
   (a) Developer selects from the 'Current MSEModel' combo-box the version on which the current detection strategy is applied
3. Developer presses the 'Run' button
4. System runs the detection strategy
5. System displays the results in a newly opened window

Use Case 5: Developer Changes The Thresholds

1. Developer selects the detection strategy
2. System displays the expression tree of the currently selected detection strategy
3. Developer selects the expression leaf whose thresholds are to be changed
4. System displays the current value of the threshold in the 'Threshold' edit field
5. Developer modifies the value of the threshold
6. System updates the value of the threshold

Use Case 6: Developer Runs A Detection Strategy Subexpression

1. Developer selects the detection strategy
2. System displays the expression tree of the currently selected detection strategy
3. Developer selects the desired sub-expression
4. Developer presses the 'Run On Expression Button'
5. System runs only the selected expression and displays the results

Use Case 7: Developer Analyze A System

1. ≪uses≫ Developer Loads Analysis Policy
2. ≪uses≫ Developer Runs A Detection Strategy
3. If Developer wants to change the thresholds
   (a) ≪uses≫ Developer Changes The Thresholds
4. if Developer wants to inspect the sub-expressions
   (a) ≪uses≫ Developer Runs A Detection Strategy Subexpression
5. If Developer wants to continue then goto Step 2

C.3 Use-Cases for Entities That Change Together Manager

This UI is responsible for managing the currently defined 'Entities That Change Together' operators. At the moment of writing this work we have only one actor - the developer.

Use Case 1: Developer Runs An Operator

1. Developer selects the desired operator from the left panel
2. Developer presses the 'Run' button
3. System runs the detection strategy
4. System displays the results in the right panel

Use Case 2: Developer Changes The Input Conditions

1. Developer selects the operator witch's inputs are to be changed
2. System displays the current values of the inputs in the edit boxes
3. Developer introduces the new inputs and leaves the focus
4. System updates the new input values

Use Case 3: Developer Deletes A Result

1. Developer selects a result
2. Developer press the 'Delete Button'
3. System deletes the result from the list

Use Case 3: Developer Browses A Result

1. Developer selects a result
2. Developer presses the 'Browse Button'
3. System opens a browser on the selected result

Use Case 4: Developer Quickly Inspects A Result

1. Developer selects a result and expands its subtree
2. System displays the members of the specific group
3. System displays below the version names in which the change occurred

Use Case 5: The Developer Analyze A System

1. ≪uses≫ Developer Runs An Operator
2. If Developer wants to change the input conditions
   (a) ≪uses≫ Developer Changes The Input Conditions
3. If Developer wants to delete the results
   (a) ≪uses≫ Developer Deletes A Result
4. If Developer wants to save the results
   (a) Developer press the 'Save Results' button
   (b) System saves the results
5. if the developer wants to browse a result
   (a) ≪uses≫ Developer Browses A Result
6. If Developer wants to continue then goto Step 1