Conformity Strategies
Measures Of Software Design Rules

By

Mircea Filip Lungu

Diploma Thesis
Faculty of Automatics and Computer Science
"Politehnica" University of Timisoara

Advisors
Dr. Ing. Radu Marinescu
Dipl. Ing. Tudor Gîrba

Timisoara
September 20, 2004

Abstract

Reengineering is a subfield of software engineering which is concerned with maintaining and improving existing software systems. Reengineering is also a process, the process by which such systems get to be understood, improved and extended. Part of this process is another process, the so called reverse engineering.

In reverse engineering man tries to understand the existing software system. There are different approaches to this task. Several of these approaches are based on metrics. One such approach is the detection strategies, a mechanism which makes use of compositions of metrics. The detection strategies offer a means for detecting design flaws in software artifacts by filtering a given set according to its property of respecting a certain design rule.

The detection strategies have proved to be useful in detecting flaws in software systems, and this is an important step in the process of reverse engineering. However their mechanism of filtering does not make any difference between the different degrees of conformity to the rule by which the filtering is made. To address this, we introduce in this work the conformity strategies, a mechanism who's main purpose is to compute the degree of conformity of the software artifacts to specific design rules.

As an application of the conformity expressions we develop a visualization technique called the Magnet View. It visually presents information about the software artifacts by letting them interact with their properties after laws derived from the equivalent physical laws of magnetism.

Acknowledgements

First of all I want to thank my parents for being two of the most wonderful people I know. I am sure that without their love and dedication, today I couldn't be able to finish a big work like this. Moreover I want to thank them for their trust they put in me when seing that other friends finish university while I still have to work all the summer.

Many thanks go to my advisors Radu Marinescu and Tudor Gîrba. I will remember the discussions we had, the ones regarding the diploma, and the ones regarding anything else, by no means less important. Especially I admire Radu's capacity of encouraging people and Doru's endless enthusiasm. May you never change in these respects.

I want to thank professor Stéphane Ducasse for making possible my collaboration with the University of Bern and for the interesting discussions and advice he gave me.

I also want to express my gratitude towards Oana and Doru for the time spent in their home while living in Bern. You are two of the most wonderful hosts I ever had. May your house be always a place where guests feel so good.

The last but not the least I must thank Adi for reviewing some chapters in this work and for being a great friend, Anca for helping with the subtleties of English and Ada for reminding me that, from time to time, one should use breaks (even if I couldn't always follow the advice).

Timisoara,
September, 2004

Structure Of The Document

The first logical part of this document will present the context of our work, going from a general level to a specific one, chapter by chapter.

Chapter 1 presents reverse engineering as being an important part of the reengineering process. We will see here why there is need for reengineering, the historical evolution of reverse engineering and see some of the approaches to reverse engineering.

Chapter 2 dives into one of the approaches to reverse engineering presented in the previous chapter, namely the software metrics. We see why and who uses metrics and understand some of the basic principles of measurement theory. Eventually we learn about the detection strategies, one approach to using metrics in the quality assessment process. We understand the purpose of the detection strategies and their working mechanism.

With the detection strategies the context is fully defined and we can proceed to presenting the actual work we have done.

Chapter 3 begins by enumerating the drawbacks of the detection strategies. To address these drawbacks, it eventually introduces the conformity strategies, the mechanism we have developed in this thesis. We will see the way the operators that compose the strategies are defined and understand what is the difference between the detection strategies and conformity strategies.

Chapter 4 is meant to be a study of the conformity strategies at work and also a comparison between the conformity and detection strategies. In the end the reader should understand why the conformity startegies are more effective than the detection strategies.

Chapter 5 presents Magnet View, the tool we have developed as a possible application of the conformity strategies. We understand its concept, its functioning and see it at work analyzing some real systems.

Chapter 6 tries as impartially as it can to present our contributions and draw some conclusions. It also sheds light on some possible future developments of the concepts presented in this work.

The last part of this work contains some supplementary material arranged in two chapters.

Appendix A is meant to present some of the implementation details of the conformity strategies and of the magnet view. It contains explanations, source code listing and UML diagrams, still I will buy a beer to the one who is able to read it from start to end.

Appendix B lists code that exemplifies the capabilities of one visualization implemented in the Magnet View, namely *Methods In Need Of Refactoring*.

Contents

Chapter 1

Reverse Engineering

> "Things are always the best in their beginning"
>
> *Blaise Pascal*

1.1 Software Aging

In an article from 1986 Frederick Brooks was stating that because software development is such a conceptually tough craft it was unlikely that in ten years an improvement would be made that could increase the productivity with an order of magnitude [**?**]. If any of the many improvements that appeared since then had made the order of magnitude jump is hard to say because the opinions are split: some say yes, some, including Brooks, say no. What is certain is that the idea behind the courageous prediction is valid - software is difficult because of its essential complexity. [1].

Indeed, the software systems are some of the most complex systems built by man and, therefore, it isn't a surprise anymore that many software projects end up having the architectural look of a "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle." as Brian Foote, so bitterly, puts it [**?**].

Parnas believes that one of the main reasons that lead to the decay of a software system is ignorant change. Changes done by somebody else than the original developer almost always make the structure of the program to

[1] The essential complexity is the mental crafting of the conceptual construct. There is also an incidental complexity represented by the implementation part, but this, as Brooks says, is not the main problem

decay because the newcomer will be unlikely to fully understand the concept the original developer had in mind when designing that system:

> *Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact, they will invalidate the original concept. [...] After many such changes, the original designers no longer understand the product. Those who made the changes, never did. In other words, nobody understands the modified product.*[**?**]

Besides ignorant change there are other factors that influence the degradation of software:

Time Pressure In an ever growing software market, the companies should develop software quick enough not to let the competition take their share of the market. But this haste puts pressure on the developers and managers determining them to try to achieve the goal by using shortcuts. And using shortcuts in the software development process will show its danger only later in time.

Inconsistent Documentation The code and the documentation are not inherently connected, the documentation's sake is merely at the discretion of the developer. This is the reason why usually documentation and code gets out of synchronization. And a software system which is undocumented or badly documented will very likely decay soon.

Prototype Not Thrown Away One of the best approaches to developing a new kind of software system is "planning to throw one away" or building a prototype [**?**]. In this process important domain knowledge is gained, unforeseen problems raise and possible future developments come to light. But alas, the psychologically difficulty of throwing one's work away and the short term benefits usually determine the changing of the prototype's destination: from prototype it becomes product, from temporary shelter it becomes dwelling. Such a project is improbable to become a project easy to maintain or understand.

By no means did we try to present a complete list of factors that determine the degradation of software structure. The important thing to understand is that in a way or another, at some point in time, a project gets to a

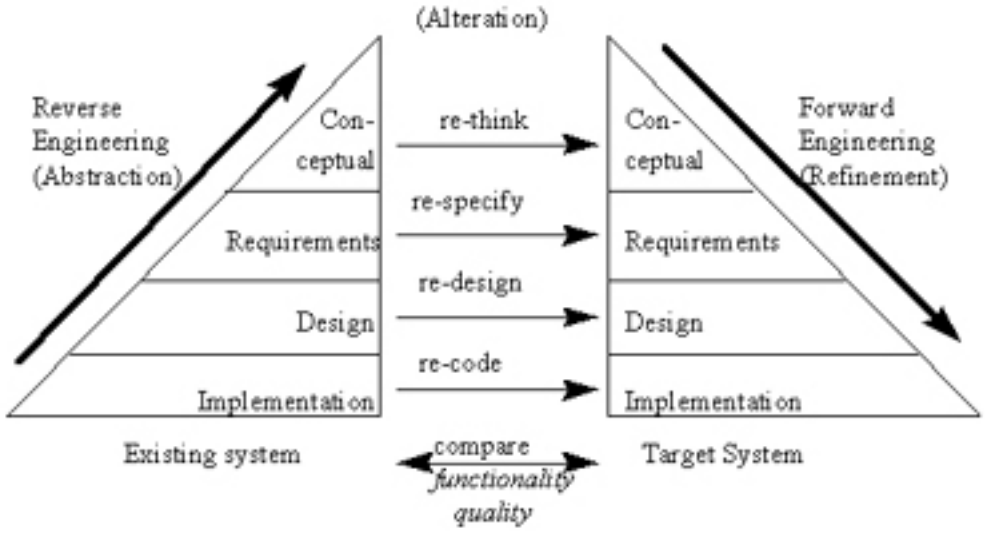

Figure 1.1: The reengineering process

phase when maintenance becomes very difficult and expensive. In this situation, two alternatives appear: rewrite the software or recondition it. Rewrite would be the best alternative, but for the cases when this is not feasible the reconditioning should be done. And in software, this reconditioning is called reengineering.

1.2 Reengineering

The software reengineering is the process of understanding and modifying an existing system. Usually, the goal of the process is to reimplement the existing system for improved quality, functionality or performance. As it can be seen in the figure **??** there are three phases of the process

Reverse Engineering The reverse engineering process is concerned with creating a higher level abstraction of the existing system. No matter which is the desired level of abstraction, there will be several components that interact. Correctly determining the components and their interaction is the focus of this phase. The reverse engineering phase is distinct, it does not look like any of the phases of the traditional forward engineering process.

Alteration After the desired abstraction level has been attained, the system representation at that level must be modified. The alteration must be done according to the current and future needs of the project.

Forward Engineering Forward Engineering is the phase in which, having understood the system, the reengineering team modifies it with the aim of incorporating the design changes introduced in the previous phase. The forward engineering is by no means different than the implementation phase in traditional software development processes. Having a higher level description of the system, the implementation is done accordingly.

Even if not presented in the previous enumeration, an analysis phase, in which the feasibility of the reengineering approach is studied is useful. Surely, the analysis phase should take place after the reverse engineering phase has been finalized, so the reengineering team has a clear picture of what it takes to reengineer the project.

1.3 Reverse Engineering

In a very well-known work of the 90's, and an fundamental one for the software engineering field, Chikofsky and Cross define reverse engineering as "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information"[**?**].

The initial reverse engineering was done on hardware circuits for military or industrial espionage. Even if more domestic in its motivation, the software counterpart has the same aim: to create a more abstract representation from the existing one. This means that there are several levels at which the reverse engineering can be done, as it can be seen from Figure **??**. The implementation can be abstracted to design, the design can be abstracted to requirements, the requirements can be abstracted to a higher level concept. The efforts of today are mainly focused at the first level, towards extracting the design from code.

For the reverse engineering process the research is developing in several directions, some of them being mentioned here with a brief description.

Software Metrics Software metrics offer a way of compressing the information available in code. This is useful because analyzing large systems by reading the code is sometimes impossible. In these cases, by

using metrics, an intuitive image of the software system can be developed. This technique is especially valuable when it is used in corroboration with software visualization. Also, by using metrics, component quality can be assessed and therefore components that should be reengineered can be detected [**?**].

Visualization Because of the brain's inherent capabilities of working with images, using the power of computer graphics to present aspects of software may prove to be an important factor in enhancing the human understanding of computer programs. There are many categories of software visualization and many tools which support software visualization. In this work, we will focus on static, metric based visualization [**?**].

Program Slicing Program slices are noncontiguous parts of code that determine the state of a variable at a certain point in the program. The automatic detection of program slices can be useful in making easier the code comprehension process by letting the software engineer focus only on the statements that influence the outcome of a variable. Program slicing is also very useful in the debugging and testing processes [**?**].

Grouping The term grouping encompasses several techniques of moving from a detailed description of a system to a more abstract description. This process is useful in reducing the perceived complexity of the system and making the information easier to fit in the head of the reengineering team's members [**?**].

Concept Assignment The code in a software system is an abstract representation of real-world concepts. In order to understand the system a human must make a mapping between programming language artifacts and the real-world concepts. This mapping is called the concept assignment problem. Even if there is an *a priori* knowledge about the domain model, the concept assignment problem is a difficult task. Software that would help in this process would be very useful [**?**].

From the previously mentioned techniques, in this work we make use of software metrics and visualization techniques. More than this, the following chapter is dedicated entirely to software metrics.

Chapter 2

Software Metrics

> "'The degree to which you can express something in numbers is the degree to which you really understand it"
>
> *Lord Kelvin*

We use software measurements, or metrics, for various reasons. Some of the most used are: quality assessment, effort prediction, complexity measurement, process evaluation. There are many issues related to software measurement. In this chapter we look at measurement theory, internal and external attributes, measurement processes and in the end see a way of composing metrics, namely the detection strategies.

2.1 History

Companies like Siemens, Hewlett-Packard, Hitachi, Motorola, NASA have one thing in common, they use metrics in their software development processes. They use them at different points in their corporate life and with different purposes [**?**]. Hewlett-Packard used it to evaluate the effectiveness of the software inspections and to improve its production process. NASA implemented a measurement program since 1973. Its main goal was improving the software development process. Hitachi created Hitachi Software Engineering company to support its software development. The goal of HSE was managing costs, schedules and quality. Siemens employed measurements as part of its best-practices program. The program resulted in a focus on quality and increased customer satisfaction.

The importance of metrics results from their capacity of making visible a process that normally is not. When you build a house, anybody can see the progress. Still, when you build software, the progress is hard to grasp. This is why having a metrics program can make the development process more visible and as a consequence, more controllable.

2.2 What Can Be Measured?

The first obligation of a measurement activity is identifying the entities and attributes that should be measured. In software there are three types of entities:

- **Processes** are the activities involved in the creation of a product, in our case, a software product. Process measurements are a way of improving the visibility of the software development process.

- **Resources** represent all the inputs of a process. They can be people, machinery or the result of other processes. In this work we will focus on the measurement of resources and mainly of the measurement of the source code.

- **Products** are the expected results of a process. The specification is the result of the requirements engineering process, the high level design is the result of the analysis and design process.

Many times what we want to measure and what we can actually measure are not the same thing. For example, we would like to measure the complexity of a module. The problem is that complexity is an abstract concept. A concrete attribute that can be measured would be the number of lines of code of that module.

Taking the previous example we can introduce the notions of internal and external entity attributes. The complexity of a module is an external attribute of the module, because it depends on the context in which we look at the module. For example, the author of the module will find it less complex than the newcomer, because of his experience. On the other hand, both the newcomer and the author of the module would agree on the number of lines of code. Therefore, the number of lines of code measure doesn't depend on the context, it is an internal attribute. Generalizing the previous example we can say that:

- **Internal Attributes** are those attributes of a process, product or resource that can be measured purely from a process, product or resource itself. They do not depend on the environment of the attribute.

- **External Attributes** of a process, product or resource are those attributes that can be measured only taking into account the way the process, product or resource relates with its environment.

2.3 Internal Product Attributes

We saw earlier in this chapter that there is a distinction between the internal and external attributes of the measured entities. Usually the internal are the easiest to measure but the external are the most important to be measured. From the internal attributes of software, the most often measured are size and structure.

Size

We will look in this section at some of the metrics related to size which have been proven to be the choice of the software engineers over time.

By far the most widely used metric in software engineering is LOC. LOC is the acronym for Lines of Code, a metric which is undoubtedly a measure for the size of the system. The reason for the high usage of this metric is maybe the simplicity of computing it. It is very easy to count lines of code. The problem gets a bit trickier when faced with the problem: what constitutes a line of code? Should the comment lines be counted? What about the block delimiters which usually stay on their own lines? But the whitespaces? Or maybe the variable definitions? From the previous questions it is obvious that one should clearly define his definition of LOC and use it consistently.

There seems to be another problem with LOC. Some lines of the program are more complex than others. Wouldn't it be fair that these should weight more in the counting? This is a tricky question. There were people which trying to address this problem introduced the S/C (size/complexity) metric [**?**]. It weights the lines according to their complexity. But why is this a tricky question? Because you can't say that LOC is a bad measure. LOC is absolutely right if used to measure the number of sheets of paper the program will be printed on. Actually it has been proved that LOC is good for more than this: it is very good as a size estimating measure. The previous

objection to LOC is similar to saying that that age, as a measure, is not enough because you can not use it to determine the weight of a man.

Surprisingly it has been proved that LOC is almost as good a measure of size as S/C. So LOC is right for the right purposes. And here we come again at the Goal Question Metric model that we presented earlier. You can't say a metric is not good. All you can say is that it might not be good for your purpose.

The problem with LOC is that it might have been a good estimator back in the nineties when almost all the programs were typed. But nowadays when graphical user interfaces and automatically generated code are in high demand, LOC is that useful anymore. A size metric which seems to be more fit for this is Albrecht's Function Points (Albrecht cited in [**?**]). Function points measures the size of a software project in terms of inputs, outputs, files, inquires. Even if they are more subjective than LOC, if used with care, FP can be a much more precise size estimator than LOC. Relevant, we might say, is that a study showed that in 1996 more than 60% of the software contracts in Netherlands have their price depending somehow on a FP estimation [**?**].

For the object oriented software good size estimators are the number of methods of a class (NOM) as a measure of class size and the number of classes (NOC) as a measure of the size of the system.

Structure

Many times the estimation of effort needed to understand a software is important. For this size is not a very good predictor. An extreme example would be the famous obfuscated programs, which, although have few lines, take much time to be understood. What we can learn from them is that naming conventions, formatting conventions, code documentation and code structure matter very much in the process of understanding software. Little effort was dedicated in the metrics field to the naming, formatting and documentation. Structure, on the other hand, has been more privileged.

One of the most widely used structure metrics is McCabe's Cyclomatic Complexity. Cyclomatic Complexity has a theoretical foundation based on graph theory. McCabe considers that a good measure for the complexity of a block of code is the cyclomatic complexity of the associated flowgraph. For a program with a flowgraph F the cyclomatic number can be computed as

$$V(F) = e - n + 2$$

where F has e arcs and n nodes. The value of $V(F)$ represents the number of linearly independent paths existing in the graph.

McCabe says that a module having a cyclomatic number higher than 10 could be problematic. There were other studies which showed a relation between the cyclomatic number in a module and the number of updates needed for the module. In the Channel Tunnel the security reasons make any module with a cyclomatic number higher than 20 to be rejected. Therefore, even if it is not perfectly intuitive, the cyclomatic number is widely accepted as a good measure of structural complexity.

There are other reasons why one would want to measure the structural complexity of the code besides understanding it. One would be predicting the difficulty of testing the code. Here we talk especially about white-box-testing. There are different approaches to white-box-testing and we will discuss each of them based on the program flowgraph concept. First would be executing each program statement at least once. This is equivalent to finding a set of paths such that each node lies on a path. Another strategy would be to execute each branch at least once. This is equivalent to finding a set of paths through the graph such that each edge is contained in at least one path. And finally, the most exhaustive would be to execute each path in the program. However this is impossible because in programs containing loops there is an infinity of possible paths. However, there is a problem with white box testing because even if you test each path, which we saw it is not possible, you might not find all the problems. See the discussion in [**?**], page 301, for a more detailed analysis of this aspect.

Because we were talking about the testing strategies, it is interesting to shake a little bit the calm we are dwelling in when thinking about testing. Mike Hennel has observed that a test with 100% statement coverage has an effectiveness ratio of only 40% while branch coverage has even more modest results.

Other Internal Properties

In a study at IBM [**?**], researchers introduced two measures of software quality. They are coupling and cohesion. For none of them a widely accepted measure is available, still they are considered important for the quality of software. Coupling is a measure of the interdependency between two modules. Cohesion is a measure of the degree in which the individual components of a module perform the same task. Data structure captures a class of structural aspects that were not given a too much attention in the research. But this is wrong, because data structure complexity is part of the complexity

of software.

The best example to illustrate the difference and complementarity between data and control structure is the paradigm shift from structural programming towards object oriented programming. The polymorphism tends to eliminate the control structure complexity (i.e. case statements) by introducing data structure complexity. It would be interesting to study if the new complexity is as everybody advertises lower than the replaced one. It would be also interesting to state a law for the conservation of the complexity similar with the law of the conservation of the energy.

2.4 External Product Attributes

As we have seen external attributes of a product are those attributes can be measured only in relation to the product's environment. For example maintainability is an external product attribute because it depends on the people who are involved in the process of evolving the software. This is obvious because if the maintenance phase the same developers are involved like in the development phase maintenance will be much more easier than when the maintainers are not familiar with the project.

Because we have seen that the external attributes depend on user interaction, therefore they are defined by the user, they usually are seen as quality aspects. We say aspects because quality is a manifold measure, or as Fenton put it *"Quality is in the eye of the beholder"*. This means that for the real-time system developer quality might mean timeliness, for a critical system developer it might mean lack of bugs and for the end-user, quality might be synonym with usability or fitness for purpose.

It is this human involvement that accounts for the difficulty of measuring the external product attributes. The situation becomes more interesting when we realize that even though they are the most difficult to measure they are in the highest demand to be measured because they are the ones that have meaning for the managers of a company. Therefore, given the high demand for external product attribute measures and the simplicity of measuring internal product attributes, it is easy to understand the efforts of measuring the external attributes by making use of the internal ones. We will look at some of the most popular external property attributes in the following paragraphs.

Defect Density

There are defects in the software that are detecting during the development process, the known defects, and there are defects which are not found, the latent defects. For the definition of defect density the following formula is used:

$$Defect\ Density = \frac{Number\ Of\ Known\ Defects}{Product\ Size}.$$

There are different problems with defect density which appear because of the multiple ways both the denominator and the numerator of the fraction can be defined. We have already discussed in the section dealing with internal metrics the multiple ways size can be measured. The same stays true for *number of known defects*: we could count only the faults or include also the failures. The failures can be found before or post-release. There are critical and non critical failures. Maybe in some cases there would be a time frame to be used in counting the defects. As it can be seen only after a rigorous definition of the terms in the fraction could we use the defect density measure and even when we have such a clear definition we should be very attentive when comparing different organizations.

Studies show that usually the defect density in the American companies ranges from 4 to 10 per KLOC and the density is lower for the Japanese companies. Software engineers agree that a defect density of 2 defects per KLOC is an extremely good achievement. An interesting measure computed in terms of defects is spoilage, defined as the fraction of total development time used to fix post-release defects.

Usability

A possible definition could be that usability is *the extent to which the product is convenient and practical to use*. Or in our computer-science jargon we would say that a product is usable if it is *user-friendly*. We still need a way to measure the usability and for this we try to decompose the external attribute in internal, measurable ones. Therefore we could say, as [**?**], that usability is a mixture of

- consistent interfaces
- good manuals
- effective use of menus and graphics
- number of help screens

Still we didn't progress much, because we can count the number of help screens and not be able to express the relationship between the number of screens and usability. More than this, good manuals factor, is again an external factor because it depends on the person whom we ask if he considers a manual as being good.

2.5 Measurement Theory

The Representational Theory Of Measurement

We use measurements in our everyday life. We measure things by comparing them and establishing relations between them. This kind of relations are called empirical relations. Because the empirical measurement is a subjective experience, it would be very useful if we could translate this knowledge in a widely accepted and rigourously formalized way. This is the concern of the representational theory of measurement.

For the relations to be preserved, a mapping from the real world to the formal, mathematical world is needed. This mapping is done in the process of measurement.

Definition 1 (Measurement) *A measurement is a mapping from the real world to the formalized mathematical world.*

Definition 2 (Measure) *A measure is a numerical value or a symbol that is assigned to an entity's attribute in the process of measurement.*

Because a measurement is a mapping, in order to be completely defined, it requires the following components

- **Domain** The domain in our case is the real world
- **Range** The range is the mathematical world
- **Rules** The rules make explicit the way of performing the mapping

More than this, a mapping should satisfy the representational condition. This means that a mapping must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations [**?**].

Scale	Transformations
Nominal Scale	1 to 1 mapping
Ordinal Scale	Monotonic mapping
Interval Scale	$M' = aM + b$ (a¿0)
Ratio Scale	$M' = aM(a > 0)$
Absolute Scale	$M' = M$

Table 2.1: Measurement scales in decreasing order of their sophistication

Measurement Scales

The advantage of mapping relations into an formal system is the power the formal system gives us of manipulating them. However, different measurements can be manipulated in different ways: we can say that the number of pages of this work is *twice as much* as the number of pages of another one but we can't say that the 30 degrees is *twice as much* as 15 degrees when working on the Celsius scale. For the study of the possible operations on measures there is a need of introducing measurement scales. There are five types of scales and we shall look briefly at each one

1. Nominal scale - it is a scale in which the measures are assigned arbitrarily. Any transformations can be applied.

2. Ordinal Scale - in such a scale the mappings to the formal system should preserve the empirical relations. Therefore any affine transformation could be used on such a scale.

3. Interval Scale - in such a scale all the properties of the ordinal scale are kept; moreover the difference between two values on the scale has meaning. Admissible transformations are linear transformations.

4. Ratio Scale - all the properties of the interval scale hold for the ratio scale; moreover there is a point named 0 which signifies the total lack of the property and therefore ratio's have meaning on such a scale. The possible transformations are proportional modifications of the input (i.e. $M' = aM, (a > 0)$)

5. Absolute Scale - there is only one transformation on such a scale and it is the identity transformation. Usually the mapping is done by counting the number of components of an attribute of the entity.

We talk about the sophistication of a scale as being direct proportional with the number of transformations that are possible to be applied on that scale. The more sophisticated the scale, the more transformations we can add to it. We see here a clear example of how rules that seem to restrict the freedom of such a scale offer freedom in the ways the measure on the scale can be interpreted.

Therefore, knowing on which scale we are working is of paramount importance: only this way we know which are the possible ways of manipulating the formal system we are working with. We will talk again about this subject in the chapter where we describe our contribution and study the scale our conformity strategies mechanism is using.

In software, measurements are also called metrics, even if this is not correct. A metric is a mathematical function which has several properties that the measurement does not hold. However, from historical reasons the term metric is widely used.

2.6 The Measurement Processes

Gilb's principle states that "Targets without clear goals will not achieve their goals clearly". Based on this principle, Fenton observes that a metrics program takes time and effort, therefore it is important to have clear goals in mind when starting it. This is the basis of the Goal-Question-Metric approach to measurement.

The Goal Question Metric was introduced by Basili (Basili cited in [**?**]. It proposes three steps for finding the correct metrics that should be collected during a program.

1. Establish the goals of your maintenance or development project.
2. Derive, for each goal, questions that allow you to verify its accomplishment.
3. Find what should be measured in order to quantify the answer to the questions.

The GQM model seems too simple to deserve to be called a model. Still, its usefulness becomes clear when realizing that many metrics programs start not with a goal in mind, but with measuring what is easy to measure and end up with bunch of unrelated and nonconclusive measurements. This situation was addressed also by Kybourg: "If you have no viable theory in which X enters, you have very little motivation to generate a measure of X".

And as a final argument for the GQM it is worth mentioning that AT&T used it successfully in assessing the utility of code inspections [**?**]

Once the metrics that will be collected are in place, it is the time to collect the data. Data collection should be done in such a way that it minimizes data corruption. For this there is need for an orthogonal system of categories in which to classify data [**?**].

Each software process cares most about quality and maintainability. For this there are some data that should be always collected like faults, failures and changes. For each the process of collection should be non-obtrusive.

Once the data is successfully collected, the analysis phase begins. Statistical methods are used in this phase of the process. One should be aware of the pitfalls that exist during this phase. One important issue now, is to realize which methods can be applied and which can not. For example using average on a set of data which is not normally distributed would not make sense.

2.7 Detection Strategies

The problem with metrics is that sometimes they are too fine grained. The result of a single metric cannot help us much in assessing the quality of the software[1]. Nobody can tell the quality of the software from its number of lines of code, classes or cyclomatic complexity.

As a step forward in working with metrics Marinescu introduces the detection strategy concept [**?**]. This section is a brief introduction to detection strategies. This is an important concept for our work because in the subsequent chapter we will present an extension of the detection strategies.

2.7.1 Defining Detection Strategies

The detection strategies were introduced as a mechanism for composing metrics and therefore allowing the engineers to work on a more abstract level, which is conceptually much closer to the real intentions when using metrics. The mechanism defined for this purpose is called detection strategy:

Definition 3 (Detection Strategy) *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.*

[1] As we have shown in a previous chapter, there are many aspects of quality and we wouldn't be wrong even if we extrapolate and talk about *assessing an external product attribute* instead of *assessing the quality of software*

The use of metrics in the detection strategies is based on the mechanisms of filtering and composition. In the following sections we will describe the two mechanisms in more detail.

Filtering Operators

A data filter is a mechanism through which a subset of data is retained from an initial set of measurement results, based on the particular focus of the measurement. The main reason for reducing data is to present to the user only the data that is useful to his purpose. If we consider the initial set to be sorted, filtering would result in a contiguous subset of the initial set which can be defined using an upper and a lower limit.

The data filters can be of two types

- **Absolute Filters** are those filters that can be parameterized with a numerical value representing a threshold. These filters are used to express very sharply defined design heuristics. Filters of this kind would be the ones called *HigherThan* and *LowerThan*
- **Relative Filters** are those filters that are parameterized with the number of entities we are interested in rather than the values of the properties for the entities. These filters are used for quantifying a "fuzzy" design rule like. Filters of this kind would be *TopValues* and *BottomValues*

Composition Operators

As we already saw, the aim of a detection strategy is to be provide the possibility of quantifying design rules. Therefore in addition to the filtering mechanism which implements the quantification of parts of the rules we need a way of putting together the parts and this is the composition mechanism. The composition mechanism is also based on a set of operators. The operators are *And Or* and *Not*. The operators can be regarded from two points of view

- **The Logical Point Of View**. From the logical point of view the operators can be seen as the way of translating the informal design rule in the formal one. They are the connection between the different assertions contained inside a rule.
- **The Set Theory Point Of View**. From the set theory point of view, the operators are the way of composing the results of the various data sets, sets that were obtained as result of the filtering process.

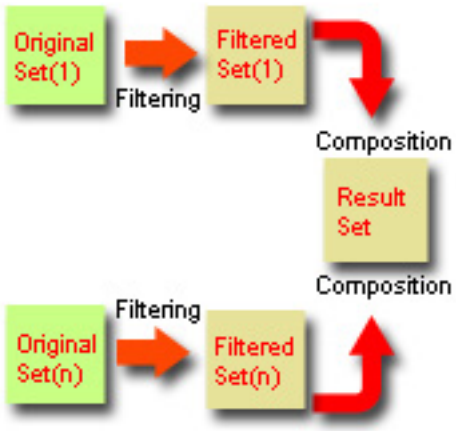

Figure 2.1: The way filtering and composition interrelate

2.7.2 Examples

Two of the detection strategies introduced in [**?**] are *FeatureEnvy* and *ShotgunSurgery*. We will show how the filtering and composition mechanisms introduced earlier work together to detect classes presenting the corresponding design flaws.

Feature Envy

The idea behind object oriented programming is to keep related data and behavior in the same class. When this is not happening methods can be found which work more with data from other classes then from their's [2]. Therefore the basic principle of object oriented programming is not respected. A solution would be to move the method to the class who's data it "envies". The expression of the strategy is as follows:

> FeatureEnvy := ((AID, HigherThan(4)) and (AID, TopValues(10%)) and (ALD, LowerThan(3)) and (NIC, LowerThan(3))

There are three metrics involved in defining the strategy and they are presented below

[2]The data can be accessed directly or using accessor methods

- **AID** is the number of data members accessed by a method either directly or through an accessor method
- **ALD** Access to Local Data, the number of local data members accessed by the method.
- **NIC** Number of Import Classes, the number of classes from which the method uses data.

Shotgun Surgery

There are situations when a change in a class leads to many small changes in many other classes. This situation is named shotgun surgery in "Refactoring: Improving the Design of Existing Code" [**?**]. The flaw is obvious because maintaining a system in which you have to keep track of multiple places where you have to make changes when something changes is hard. We have here a problem of coupling: the system is too strongly coupled. One of the possible reasons would be failing to respect "Demeter's law". The expression for detecting classes which are suspect of inducing shotgun surgery in the system is

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

There are two metrics involved in the expression

- **CM** Changing Methods, the number of distinct methods in the system that would be potentially affected by changes operated in the measured class.
- **CC** Changing Classes, the number of client-classes where the changes must be operated as the result of a change in the analyzed class.

Chapter 3

Conformity Strategies

> "Measurement owes its existence to Earth; Estimation of quantity to Measurement; Calculation to Estimation of quantity; Balancing of chances to Calculation; and Victory to Balancing of chances"
>
> *Sun Tzu*

In this chapter we discuss the drawbacks of the detection strategies and, in an attempt to address these drawbacks, we introduce an extension for them: the conformity strategies.

3.1 Detection Strategy Drawbacks

In the previous chapter we have seen how the detection strategies work. We have seen that they offer a mechanism for working with a higher level of abstraction than the mere metrics and express in a quantifiable manner some design rules.

If we had to graphically depict the conceptual working of the detection strategies we could use Figure **??**

3.1.1 Lack Of Ranking

The rules that the detection strategies quantify are "rules of thumb" or statistical rules. One characteristic of such rules is that different entities will conform to the rules in different degrees. And from here we can see

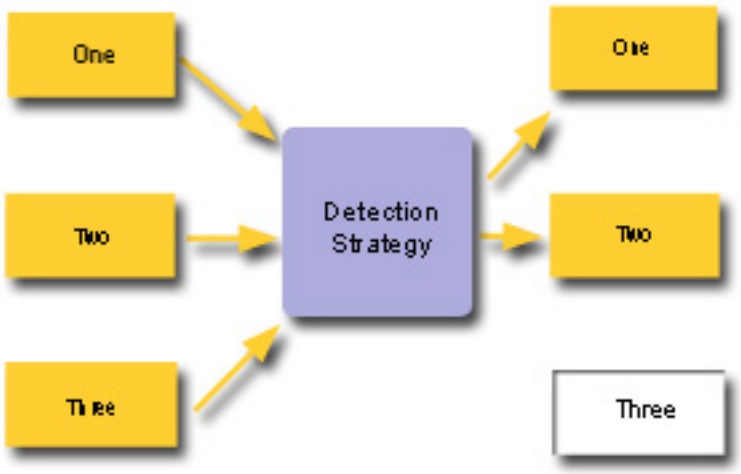

Figure 3.1: *Detection Strategy Concept.* The detection strategy filters the set of input entities. The figure shows that entities *One* and *Two* conform to the rule (they are presented as colored) while entity *Three* does not conform to the rule (it is colored in white)

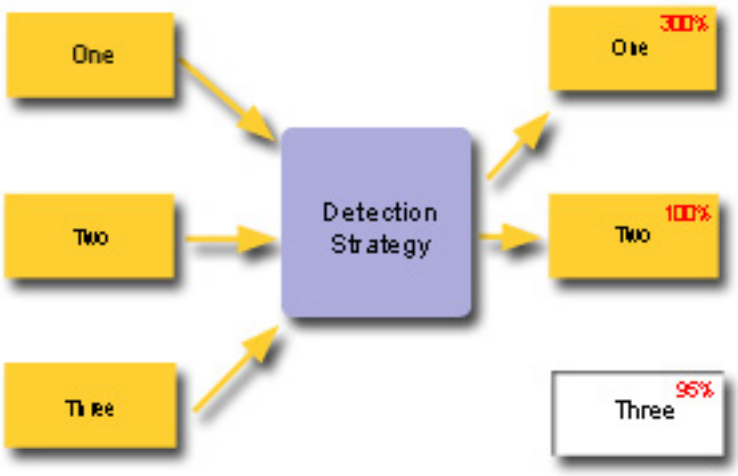

Figure 3.2: *Conformity Strategy Concept.* Conceptually the filtering mechanism remains in place but the entities are annotated with the *degree of conformity to the rule* of the detection strategy.

one facility that is not provided by the detection strategies, but would be useful in practice, the ranking of the filtered set according to the degree of conformity to the quantified rule.

Let's take for example the rule *"A method should not have the cyclomatic complexity number (TCC) higher than 20"*[1] and quantify it as the *Structurally Complex Method* detection strategy. Creating a detection strategy for this rule and filtering a given set of entities could result in a filtered set containing a method with TCC = 20 and one with TCC = 40. Even if none of them is conforming to the rule the second is more likely to hide a design problem than the first.

3.1.2 The False Negatives Issue

The detection strategy is more than a filter because it attaches a meaning to the entities that it detects. This being said, in the very definition of the strategy, there is an assertion included about the entities that are going to

[1]The rule is applied to the software written for the Channel Tunnel. See [**?**]

be filtered. For the rule defined in the previous section we suppose that the entities (i.e. methods) that will be filtered are structurally very complex. If there were structurally complex methods that were not detected we would say about them that they are *false negatives*. They are detected as negative relating to the rule but this is false.

Because of the rigidity of the threshold mechanism involved in defining the expressions for the detection mechanism, the existence of false negatives is inevitable. Let's take again the rule we defined relating to the structural complexity of the methods. We could apply the rule on the entities (i.e. methods) of a system to detect the structurally very complex methods of the system. Suppose we have a number of methods with TCC = 20. Applying the *Structurally Complex Method* detection strategy on the system won't detect our methods because they don't have $TCC > 20$ (!). Even if our methods are very complex they won't be detected because of the rigidity of the filtering mechanism.

The filtering mechanism on the other way is the strong point of the mechanism. By making use of it we can reduce the initial set of entities to one that can be more easily handled and is relevant to the problem. Therefore we would like to address the problem of the false negatives in some way but don't want to interfere with the already proven useful filtering mechanism. The solution we found will be detailed in the following chapter and is represented conceptually in Figure **??**. As it can be seen the solution is to rate each entity according to *the degree of conformity to the detection strategy's rule*. There are many possible rules for rating the entities and we will look in the next chapter for some of the possibilities.

3.2 Conformity Operators

Definition 4 (Conformity Operators) *The conformity operators are operators that, applied on an entity, give as result a rating representing the degree of conformance of an entity to the simple rule expressed by the operator.*

The HigherThan and LowerThan were easily implemented as filters. If the property that was taken into account was higher or lower than the threshold, as the filtering operator suggested, the entity was filtered or not. It is our concern in the new implementation not only to filter the entities but to also give ratings to them.

Let's take for example the following operator $NOM > 30$. We decide to return the result as the degree in which the analyzed entity conforms to

the rule in percents. One possible implementation would be to assign the ratings linearly, having therefore

$$HigherThan\ 30\ rating = \frac{Actual\ value\ of\ NOM\ for\ entity}{30}.$$

This means that for an entity with NOM = 30 we would get a result of 100% while for a actual value of NOM of 60 we would get a result of 200%. From here the filtering mechanism can be very simply implemented by offering as a result only those entities with a rating higher than 100% like in the following code extracted from the class Filter

```
SCG.Van defineClass: #Filter
        superclass: #{SCG.Moose.MSEAbstractTool}
        instanceVariableNames: 'expression'

Filter>>computeFor: aGroupOfEntities
        ^aGroupOfEntities
            select: [:each|
                (expression computeFor: each) >100
                ].
```

Surely, there can be alternative ways of computing the rating for an entity depending on the characteristics of the conformity operator and also depending on the measured property but, nevertheless, the filtering remains the same.

We will focus now on the implementation of the operators in our implementation giving alternative rating methods where they make sense. The operators are HigherThan, LowerThan, Defuzzify.

Axiom For The Preservation Of The Behavior

The new conformity operators should respect the following axioms in order to preserve the behavior of the detection strategies:

Definition 5 (Axiom For The Preservation Of The Behaviour) *The rating should be greater than equal 100% for all entities that would have been filtered by the old filtering operators.*

3.2.1 HigherThan

Rating

The rating for HigherThan will be done according to the formula

$$Rating(HigherThan) = \frac{Actual\ value\ of\ metric\ for\ entity}{Specified\ treshold\ for\ metric}.$$

Regarding this formula we can make a few observations:

- As we recall from the chapter on measurements, this formula does make sense only if the measurement is made on a ratio scale. Fortunately, many of the metric definitions available today are on a ratio scale
- Even if we are working on a ratio scale, for a threshold of 0 the formula does not make sense. In this case we will consider the output 100% whatever would be the actual value of the metric.
- The conformity rating for the HigherThan operator is a ratio, an adimensional number. It does not have the meaning of a metric anymore. It merely suggests a degree of conformity or an alarm level.

If we looked at the operator as a function we could represent it graphically in the following way

Implementation

The code for computing the rating for one entity is very simple. It must be mentioned however that for the complete understanding of the place in the framework of the presented code the reader should consult appendix A.

```
CFHigherThanOperator>>computeFuziness: aValue
        threshold = 0 ifTrue: [^100].
        ^(aValue / threshold * 100) asInteger.
```

3.2.2 LowerThan

Rating

The rating for this operator is done according to the formula

$$Rating(LowerThan) = \frac{Specified\ treshold\ for\ metric}{Actual\ value\ of\ metric\ for\ entity}.$$

Regarding this formula we can make a few observations:

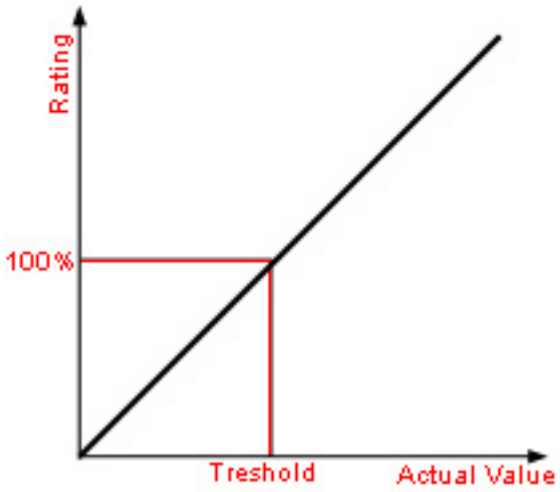

Figure 3.3: *HigherThan* operator, represented graphically as a function

- As we recall from the chapter on measurements, this formula does make sense only if the measurement is made on a ratio scale.
- Even if we are working on a ratio scale, for an actual value of metric of 0 the formula does not make sense. In this case we will consider the output 100% whatever would be value of the threshold.
- The conformity rating for the LowerThan operator is a ratio, an adimensional number. It does not have the meaning of a metric anymore. It merely suggests a degree of conformity or an alarm level.

If we looked at the operator as a function we could represent it graphically in the following way

Implementation

The code for computing the rating for one entity is very simple. It must be mentioned however that for the complete understanding of the place in the framework of the presented code the reader should consult appendix A.

```
CFLowerThanOperator>>computeFuziness: aValue
        aValue = 0 ifTrue: [^0].
        ^(threshold / aValue * 100) asInteger
```

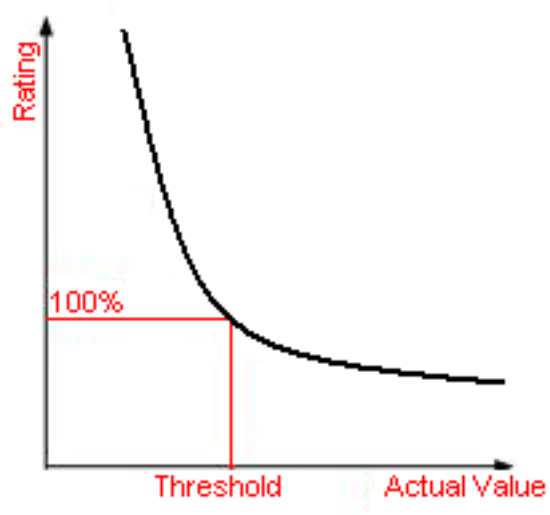

Figure 3.4: *LowerThan* operator, represented graphically as a function

3.2.3 Defuzzify

Rating

The rating for this operator is done according to the formula

$$Rating(Defuzzify) = \begin{cases} 0 & Actual\ Value <= Threshold \\ 100 & Actual\ Value > Threshold \end{cases}$$

If we looked at the operator as a function we could represent it graphically in the following way

Implementation

The code for computing the rating for one entity is very simple. It must be mentioned however that for the complete understanding of the place in the framework of the presented code the reader should consult appendix A.

```
CFDefuzzyfyOperator>>computeFor: anEntity
    (self children first computeFor: anEntity) >= 100
        ifTrue: [^100]
        ifFalse: [^0]
```

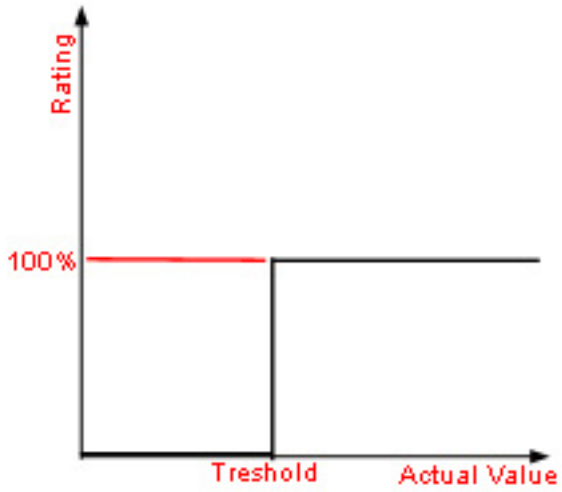

Figure 3.5: *Defuzzyfy* operator, represented graphically as a function

3.3 Composition Operators

The composition operators have the role of

In the old detection strategies the composition operators had the role of set operators. "And" was defined exactly as the logical and, and "or" the same way. In our mechanism we took a different approach.

The expression of the detection strategy is computed separately for each entity, the entity is rated according to the expression. Applying the strategy on a set of entities will yield another set with each entity having associated its rating. Only then the filtering takes place according to the rating of the entities. The difference between the two mechanisms is that while the old detection strategies did many filtering operations in the computation of an expression, the conformity strategies do only a filtering step: the final step before presenting the result.

However, we must assure that the implementation of the composition operators and the overall implementation of the conformity strategies based on them, preserves the behavior of the old detection strategies. We will prove this after introducing the composition operators.

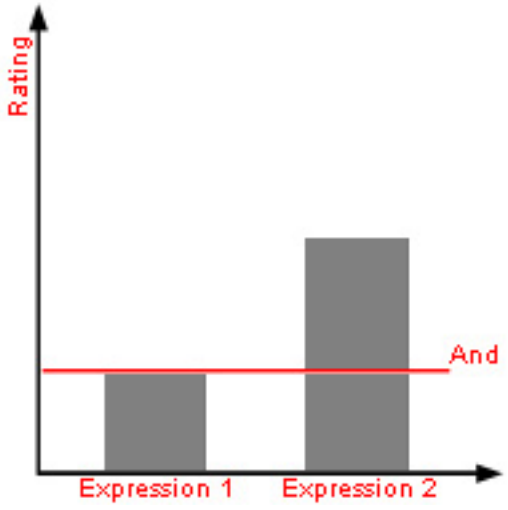

Figure 3.6: The *And* operator

3.3.1 And

Rating

The composition operators are binary operators. As input they have two conformity expressions. The rating of the composite expression will be given in the case of "And" by the minimum of the ratings of the two expressions. This approach is a reminder of the way the "And" operation is done in fuzzy logic. The only difference is that the result in fuzzy logic the result is restricted to the [0..1] range while in our rating scheme the values can be bigger than 100%.

For an expression ExpAnd = Exp1 *And* Exp2 we could compute the result using the formula

$$Rating(ExpAnd) = min(Rating(Exp1), Rating(Exp2)).$$

Regarding this formula we can make a few observations:

- The chosen implementation of the rating for "And" preserves the results the old detection strategies had. Suppose we have two expressions which respect the *Axiom For Preservation Of The Behavior* (Section **??**). The old detection strategies would check if the entities respect the rule of each expression and all those who respect will be filtered.

Therefore we would have as a result two sets. The entities that exist in both sets will form the result set after being filtered again by the And operator. But the property of the entities in the resulting set is that they respect both composed rules. Therefore when applying the conformity expressions on each of those entities they, and only they, will have the rating for each expression higher than 100%. Therefore after applying the rating for the "And" operator as defined earlier they, and only they, will have a resulting rating higher than 100%. This means that if we apply the final filter we presented in the first section of this chapter, they and only they, will form the resultant set. Therefore, we see that the behavior will be preserved.

- Somebody might raise the objection that we are adding apples with oranges here because we assign a number to a composition of two metric expressions, metrics that might not be related one with another (e.g. NOM>20 and NOC>4). The important observation is here that the conformity rating is not a metric anymore. We have seen when talking about conformity operators, that the meaning of the rating for the operators is merely that of an alarm.

Implementation

The code for the effective computing the rating for one entity is spread across two levels of the class hierarchy under CFCompositionOperator.

```
CFCompositionOperator>>computeFor: anEntity

        ^self
            fuzzyOperationOn: (self left computeFor: anEntity)
            and: (self right  computeFor: anEntity).

CFAndOperator>>fuzzyOperationOn: firstPropertyValue and:
secondPropertyValue

        ^firstPropertyValue min: secondPropertyValue.
```

3.3.2 Or

Rating

The rating of the composite expression will be given in the case of "Or" by the maximum of the ratings of the two composing expressions. This

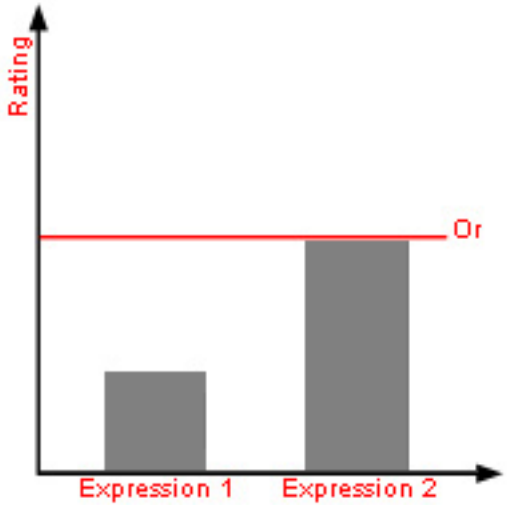

Figure 3.7: The *And* operator

approach is a reminder of the way the "Or" operation is working in fuzzy logic. The only difference is that the result in fuzzy logic the result is restricted to the [0..1] range while in our rating scheme the values can be bigger than 100%.

For an expression ExpOr = Exp1 *Or* Exp2 we could compute the result using the formula

$$Rating(ExpOr) = max(Rating(Exp1), Rating(Exp2)).$$

Regarding this formula we can make a few observations:

- The chosen implementation of the rating for "Or" preserves the results the old detection strategies had. Suppose we have two expressions who respect the *Axiom For Preservation Of The Behavior* (Section **??**). The old detection strategies would check if the entities respect the rule of each expression and all those who respect will be filtered. Therefore we would have as a result two sets. The entities that exist in at least one set will form the result set after being filtered again by the "Or" operator. But the property of the entities in the resulting set is that they respect at least one of the rules joined with "Or". Therefore when applying the conformity expressions on each of those entities they, and only they, will have the rating for at least one expression

higher than 100%. Therefore after applying the rating for the "Or" operator as defined earlier, they, and only they, will have a resulting rating higher than 100%. This means that if we apply the final filter we presented in the first section of this chapter, they and only they, will form the resultant set. Therefore, we see that the resulting set behavior will be preserved in the case of "Or".

Implementation

The code for the effective computing of the rating for one entity is spread across two levels of the class hierarchy under CFCompositionOperator.

```
CFCompositionOperator>>computeFor: anEntity

        ^self
            fuzzyOperationOn: (self left computeFor: anEntity)
            and: (self right  computeFor: anEntity).

CFOrOperator>>fuzzyOperationOn: firstPropertyValue and:
secondPropertyValue
        ^firstPropertyValue max: secondPropertyValue.
```

3.4 Conformity Strategies

We have seen that the main goal of the detection strategies is to provide the engineers with a filtering mechanism based on the composition of metrics. In this process an expression is built which expresses a design rule. The software artifacts are filtered according to this expression.

The goal of the conformity strategies is more than filtering, is computation of the degree of conformity to the expressed rule. This degree of conformity is given by a grade assigned to the subject entity[2]. In the process of assigning the grade we also use a similar expression as the one for the detection strategies but with different operators, operators that were described in the previous sections.

Definition 6 (Conformity Expression) *A conformity expression is a recursive function composed of conformity operators applied on software artifacts*

[2]While it was the main goal of the detection strategies, filtering is just one possible application of the conformity strategies

and composition operators applied on the results of conformity or on other composition operators.

The conformity strategy is not more than an expression which has a semantical meaning. We see this from the definition

Definition 7 (Conformity Strategy) *A conformity strategy is a function used as a means of automatically computing the degree of conformity of some software artifacts to a design rule.*

As it can be seen from the two previous definitions, there is a subtle difference between conformity expressions and conformity strategies. A conformity expression does not have to represent a design rule. Therefore a strategy always has an associated expression but a strategy is not necessarily built from a given expression. This is true because there could be expressions which do not have any semantic, which do not represent any meaningful rule.

3.5 Chapter Summary

In this chapter we saw the context in which we implemented the conformity strategies, we presented some of the implementation details and we understood how the different operators interact in order to create the conformity expressions.

We have been theorizing about the conformity strategies in this chapter. The theory is over and we move to the applicative part. In the following chapter we take a software system and apply conformity strategies on it in the context of a case study. The subsequent chapter will present an application of the conformity strategies.

Chapter 4

Case Study

> "Geometry is the art of finding perfect solutions to problems with the help of imperfect figures"
>
> *Anonymous*

Because engineering is not geometry, we never have perfect solutions. Neither detection strategies nor conformity strategies are flawless. However, in this chapter, we will show on several real case studies why conformity strategies are a better solution than the detection strategies. For this we plan to apply the two mechanisms on several smalltalk systems and compare the results.

4.1 The Experimental Setting

We will analyze the following systems: CodeCrawler, AdventureBase and Advance (Table **??**. For the analysis of CodeCrawler we had the permission of the author while the other systems are in the public domain so we didn't have to ask for any permission. For each o the systems we will provide a separate subsection in which we discuss the problems related to it.

The goal of the analysis is to detect God Classes. God Class is a "bad smell" defined by Fowler in his *Refactorings* book [**?**]. A God Class is a class which concentrates much of the systems intelligence. This is a sign of action oriented programming which is opposed to object oriented programming. As Riel advises, the system intelligence should be distributed horizontally so "the top-level classes in a design should share the work uniformly"[**?**].

System	Number Of Classes	Number Of Model Classes
Advance	2037	264
Adventure Base	308	108
Code Crawler	457	115

Table 4.1: The systems used in our study

For the detection of God Classes we use a simplified expression of the strategy defined by Ratiu [**?**]

$$God\,Class = (ATFD > 40) AND ((WMC > 75) OR ((TCC > 0.20) AND (NOA > 20))).$$

The metrics involved in the expression have the following meanings

- **ATFD** (Access To Foreign Data) 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.

- **WMC** (Weighted Method Count) is the sum of the statical complexity of all methods in a class. If this complexity is considered unitary, WMC measures in fact the number of methods (NOM). Usually WMC is computed as the sum of the cyclomatic complexities of all methods in a class.

- **TCC** (Tight Class Cohesion) 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.

- **NOA** (Number Of Attributes) The number of attributes of the class. As an interesting issue, in smalltalk all the attributes are private.

The expression represents the formal representation of the following rule: *"If the class has a high degree of access to foreign data and has a high internal complexity or if it has a high degree of access to foreign data and a low cohesion"* than the class is a God Class. The two variants of the rule can be traced back to the two flavors of God Class defined in [**?**].

Class Name	CS Rating	DS Result
Class A	185	detected
Class B	100	detected
Class C	40	
...	< 40	

Table 4.2: Sample Results Table

4.2 The Experiment

We apply the detection strategy and the conformity strategy based on the previously introduced God Class expression on each of the systems mentioned at the beginning of this section. The results are separately discussed for each system and presented as tables.

For the layout of the tables we can look at Table **??**. For each row we have a class and the results obtained by using the two mechanisms. On the column corresponding to the conformity strategy there is a rating representing the degree ion which the class conforms to the God Class strategy. On the column corresponding to the detection strategy we have the string *detected* if the entity was detected as being suspect of having the design flaw and nothing otherwise.

We can see that in the last row from table **??** there is no class name and an inequality instead of the rating. This means that the entities which had a rating expressed by the inequality were not presented because we considered them to be irrelevant.

4.2.1 CodeCrawler

The result of applying the strategies on Code Crawler is presented in Table **??**. We can see that all the classes that were detected by the detection strategies as suspects, were assigned a higher than 100 rating by the conformity strategies. This should not be a surprise, because in the previous chapter we already proved the equivalence between the results of the detection and conformity strategies.

What is worth mentioning is that after analyzing the code we come to the conclusion that CodeCrawler really is a class which concentrates a lot of behavior, controls some sequential events and can be considered to be a God Class. CCNodeFigureModel and CCDrawing are big classes but they are more cohesive and represent some clearly defined concepts. They can be considered suspects of being God Classes but they are not so acute as

Class Name	CS Rating	DS Result
CodeCrawler	185	detected
CCNodeFigureModel	118	detected
CCDrawing	117	detected
CCItemFigureModel	62	
CCEmbeddedSpringLayout	60	
CCGraph	52	
CCViewSpecNodeTypeSubcanvas	48	
CCNode	46	
CCSingleClassBlueprintLayout	45	
CCDrawingProxy	45	
CCViewBuilder	41	
...	< 40	

Table 4.3: God Class detection in CodeCrawler

the first mentioned. This is information we derived after looking at the code. The same information could have been deduced from the result of the conformity strategy but couldn't be deduced from the detection strategy as Table **??** suggests.

4.2.2 AdventureBase

AdventureBase is a framework for the development of role-playing games in Smalltalk of medium size, having more than 5Klines of code and 154 classes (308 if we also count the metaclasses). After applying the strategies on the system we got the results listed in Table **??**.

The most interesting class for our discussion is the AdventureContainer-Inspector class. We see that it *almost* conforms to the rule. If we used only the detection strategies we would have discarded this class as not being suspect. However, its high rating makes it a suspect and we should better do a manual inspection to determine if it is flawed or not.

Analyzing the class we see that the class is referred by only three methods in the system but it refers 251 methods. Therefore we see that the class really is some kind of a controller class. The class is also in the top 3 classes in the system with respect to accessing foreign data having ATFD = 55. In fact only 10 classes in the system have an ATFD higher than 10 and it must be reminded again that the system has more than 100 classes. Therefore our class seems to do too much, access too much external data and probably

Class Name	CS Rating	DS Result
AdventureThing	178	detected
AdventureControlPanel	114	detected
AdventureContainerInspector	98	suspect
AdventureActor	84	
AdventureRobot	66	
Machine	30	
AdventurePlayerMonitor	30	
MachineState	26	
AdventurePlayer	26	
Adventure	24	
AdventureLauncher	21	
...	< 20	

Table 4.4: God Class detection in AdventureBase

should distribute its behavior to other classes. The symptoms described suggest the class as a candidate to being a flawed class.

The important thing to notice in this example is how the conformity strategies enabled us to detect a class which would have remained undetected using the old detection strategy mechanism.

4.2.3 Advance

The conclusions we got after we studied the code are reported below. In this case study there is no doubt that the first detected class is a God Class. AD2DiagramPainter is the brain of the system, it is the class which handles the graphical editing and this is why it is so complex. On the other hand it's complexity would be hard to split among other classes and maybe the design would be less intuitive if one did so.

The next detected classes are big and complex classes but, after manual inspection, we concluded that they are cohesive enough to represent unique concepts in the domain so we don't consider them to be dangerous God Classes. Nevertheless, even if not dangerous, the fact that a class in the system is so big, complex and accesses so much external data might be a sign of poor design.

It's important to notice that even if some of the detected classes are false positives, the ranking possibilities offered by the conformity strategies increase the probability that the real flawed classes are detected first. This

Class Name	**CS Rating**	**DS Result**
AD2DiagramPainter	255	detected
AD2DiagramModel	221	detected
AD2ClassModel	209	detected
AD2SubjectBrowser	127	detected
NVEditor	102	detected
AD2ApplicationModel	85	
ICC1Dialog	66	
IccVWSystemOrganisation	64	
AD2ToolbarGenerator	53	
IccHTMLVWDocumentation	50	
...	< 50	

Table 4.5: God Class detection in Advance

is what happened with AD2DiagramPainter in our example: we saw that it had the highest probability of being flawed from all the classes in the system and, in the end, we concluded that the rating was right.

Chapter 5

Magnet View

> "Software is intangible, having no physical shape or size. Software visualization tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behavior"
>
> *T. Ball*

Having validated the thesis, we go a step forward and present one application which uses conformity strategies, the Magnet View. We begin this chapter by presenting the philosophy behind Magnet View and its purpose. After that we go in a little detail about how the tool is integrated in the Moose environment and especially in CodeCrawler a software visualization tool. In the end we will have a look at some applications of Magnet View.

5.1 The Concept

We start by explaining the second term from the name of our tool, *view*. One of the definitions of view in the Webster's dictionary is *to regard in a particular way* and this is exactly what we want to do with Magnet View, to introduce a new way of looking at a software system. In our implementation of the view we used the exiting infrastructure of CodeCrawler, infrastructure that will be discussed in one of the following sections.

The other term from the name, *magnet*, suggests the particularities of the view. If we generalize the way a magnet attracts iron entities, we could say that an *attractor*, the magnet, attracts the *attractees*, the iron entities,

conforming to some physical laws. What we did was to implement a view that lets the software entities, as attractees, interact with several attractors which represent attributes of the attractees by respecting simple laws analogous to the magnetic laws.

The attractees In Magnet View the attractees are software artifacts have corresponding entities in Moose. That is they can be subsystems, classes, methods, attributes or invocations. However it must be mentioned that, although we can create views that work with any of the previous mentioned types of entities, in a view all the entities should be of the same type.

The Attractors The attractors as we have already said should represent quantifiable properties of the attractees. In Moose the attractors could be metrics but could also be conformity strategies because the conformity strategies are compositions of metrics.

The Interaction Law The interaction law states that each entity is positioned in such a way that the attractions of all the attractors that affect it be in equilibrium. Each attractor attracts each attractee with a force proportional with the degree the attractee has the property symbolized by the attractor.

The Magnet View The Magnet View is the view that results by letting the attractors and attractees defined before interact. By observing the resulting positions of the attractees we would be able to make assertions about the entities they represent. By observing the configuration of the attractees we would be able to make assertions about the system.

5.2 Relation With CodeCrawler

For the visualization that we have proposed in this chapter we could have developed a new software tool. We decided not to do this but to use the existing framework provided by CodeCrawler. CodeCrawler is a tool based on Moose that supports reverse engineering by software visualization. Its aim is to present an overview, a *feeling* of the system, a starting point from where the reverse engineering process can continue.

We have decided not to create a tool from scratch because we wanted to test our concept. Therefore we needed a fast implementation, and the fastest

Figure 5.1: *Magnet View Concept.* We can see how the attractors (A1, A2, A3) create forces (F1, F2, F3) that influence the position of the attractee (E). It should be noted that because A1 attracts E weaker (F1 is smaller than F2 and F3) E is closer to A2 and A3

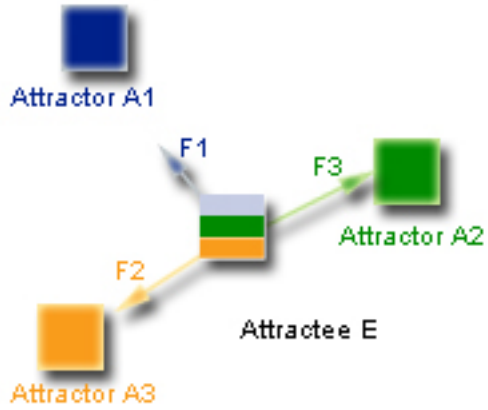

Figure 5.2: *Magnet View as a polymetric view.* We can see that the blue band has a faded blue shade to express the fact that the blue attractor has a weak influence on the attractee

way to get this was integrating our view in CodeCrawler. CodeCrawler offered some facilities like polymetric views, integration with moose and interactivity features that were needed. To implement our extensions we had to subcases some existing classes and overwrite some methods by making use of the extensions mechanism provided by Smalltalk.

The aim of polymetric views is to present as much information as possible in a 2d diagram. A polymetric view maps attributes on the dimensions of the rectangular figures representing the entities, and also can map attributes on color and position of the entities. As it can be seen in Figure **??** we used polymetric views that make use of color an position. The color of each attractee is an aggregate of bands each band having the color of an attractor and the intensity proportional with the force the attractor determines on the attractee. The position is given by the overall interaction of the forces the attractors induce on the attractees.

5.3 Applications

We introduce now a set of views that illustrate possible ways of using the Magnet View tool. These are some of the possible views and by no mean are they an extensive list. Depending on needs and model properties new views can be defined.

In the following sections we will see some ways of using the Magnet View. Each visualization has its section. We split each section in the following subsections:

- *Purpose* represents the high level goal of the visualization
- *Applies On* specifies the entity type addressed by the current view. It could be class, method, package or any other entity type defined in Moose.
- *Rationale* presents the aims we had for defining the view.
- *View Definition* introduces the attractors of the view and the attracted entities.
- *Interpretation* presents what properties we expect the entities from a specified region on the graph to have. Also introduces any other observations regarding the view.
- *Validation* takes a real-life software system and applies Magnet View on it. From the view it derives some conclusions that are then validated or invalidated by analyzing the code of the system or from studying writings about the system.

5.3.1 Important Classes

Purpose

System Understanding

Applies On

Classes

Rationale

A class to which much effort was dedicated should be an important class of the system. A sound measure of effort is size, and two measures of size are NOM and LOC (see section **??**) This is why we correlate these two measures in order to detect the important classes of the system.

View Definition

We use two conformity expressions as attractors. They are,

- **(NOM > 20 OR WLOC > 200)**. The thresholds are for a smalltalk system where the average method length is 8 LOC and average number of methods for a class is 10 [**?**].
- **TRUE**. The default attractor attracts all the entities with a force equal with the force an normal attractor would show for an entity which conforms 100% to its rule.

This view applies on all the classes of the system.

Interpretation

Because of the default attractor the classes with conformity around 100% should be at the half of the distance between the two attractors. The classes closer to the default attractor are classes which do not respect neither one of the two conditions imposed by the Main Actor conformity expression.

Validation

HotDraw is a 2-D graphical framework written in smalltalk. Suppose we never heard about it and we want to understand the system in order to further develop it. A first step could be detecting the main actors of the system. We apply the view on the system and we get the result shown in the Figure **??**

In Figure **??**, the region marked with 1 we have the following classes: Drawing, DrawingEditor_class, Figure, ToolState, Tool_class. Because they are the leftmost they are the most suspect of being MainActors for the system. Now we need to verify if indeed these classes are main actors in the system.

One of the most famous papers on HotDraw is R. Johnson's "Documenting Frameworks Using Patterns" [**?**]. In his paper Johnson proposes a new

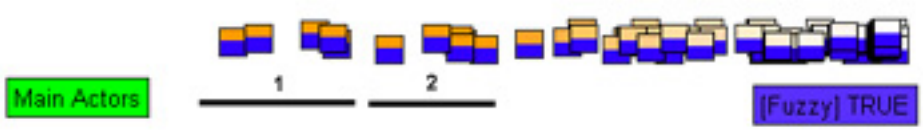

Figure 5.3: Main Actors as they are detected in the HotDraw Use Case

approach to documenting frameworks by presenting a structured list of ways of using them. These ways of using the framework he calls patterns. As a validation of his approach he documents HotDraw using patterns. For the proposed task he uses a set of 10 patterns he arranges in the order of their importance.

We would consider that we succeeded in our approach if the classes we suspect as being main actors are found in Johnson's patterns. And now we take each of our suspects and see if they exist in Johnson's documentation.

- *Drawing* We just quote from Johnson: "Other important classes are Drawing [...]". In pattern 9 Johnson talks about animation and says "Animation is provided in HotDraw by making a subclass of Drawing that defines the step method. This is the main reason that Drawing is subclassed."

- *DrawingEditor_class* The first pattern is dedicated to introducing the DrawingEditor. We quote again: "The DrawingEditor is the model, and is responsible for keeping track of the drawing, the set of tools, the current tool, the menu of operations on the drawing, and many of the operations on drawings."

- *Figure* The second pattern describes the Figure class. Here we find out that "Each kind of drawing element is a subclass of Figure. [...] A figure keeps track of other objects that depend on it [...] Each drawing element in a HotDraw application is a subclass of Figure"

- *ToolState* This class is not found in the documentation. We will investigate why was it detected.

- *Tool_class* Pattern 8 is entirely dedicated to tools. Here we find that "Selecting a tool from the palette lets the user manipulate figures, create new figures, or perform operations upon a figure or the entire

drawing. An important part of designing an editor using HotDraw is to design the set of tools that will be on the palette."

The quotations we gave from mr. Johnson's paper make any further comments of ours superfluous. However, as a conclusion we can say that for this case, our tool had proved to be effective in helping us detect the main actors of the system.

5.3.2 Internal Complexity vs. Interface

Purpose

System Maintenance, System Understanding

Applies On

Classes

Rationale

There are classes in the system which do have a high internal complexity, still they expose a small interface to the other classes. We use this view to emphasize these classes. These classes might be classes which

View Definition

We use three conformity expressions as attractors. They are,

- **WMSG > 250** The thresholds correspond to the used case study. WMSG means number of all message sends in all methods of the class. We chose the threshold to be the median value of the WMC distribution. This is very different from the average which would be much lower due to the high number of classes in the system having a small WMSG count. However the distribution of the values is not normal so statistics say that it makes no sense to use average with such data sets.

- **WMC > 40** Threshold is relative to the studied system and it represents the median of the data set with WMC values. WMC is the sum of all the CYCLO for all the methods of the considered class. It is worth observing that for more than 75% of the methods in our study the CYCLO metric is 1.

- **PubM > 40** Threshold is related to WMC. Knowing that most of the WMC are 1 we would expect that entities in the center of the image to be attracted equally by WMC > 40 and PubM > 40. Any unbalance would mean a disproportion in the expected relation between PubM and WMC and might be inspected.

This view applies to all the classes of the system.

Interpretation

The $WMC > 40$ and $WMSG > 250$ attract the classes with a high internal complexity. We have considered high internal complexity to be sending many messages to other classes and also having a big cyclomatic complexity for their methods. The PubM is a measure for the public interface of a class.

Validation

We have already introduced CodeCrawler, the visualization tool available in Moose. Applying the *InternalComplexity vs. Interface* view on it now and we get the result shown in the figure **??**

In Figure **??**, we see that there are three outlier classes. One is the class marked as 1 and the other is the group of two classes marked as 2 [1]. Looking at the view we can state that class 1 is strange because it sends a lot of messages in its methods still it does not provide any interface to other classes nor does it have a high structural complexity. Therefore we expect this class to have methods which send a lot of messages in a linear fashion. Looking at the code we realize that we were right, the class is CCSplashScreen_class and it has a method splashScreenImage with many lines of generated code, some of them we list here for :

```
... at:1 put:Graphics.ColorValue black;
at:2 put:(Graphics.ColorValue scaledRed: 57 scaledGreen: 56
scaledBlue: 54);
at:3 put:(Graphics.ColorValue scaledRed: 30 scaledGreen: 17
scaledBlue: 26);
at:4 put:(Graphics.ColorValue scaledRed: 79 scaledGreen: 77
scaledBlue: 79);
```

[1] It is interesting to note that in the proximity of groups 1 and 2 there are also other groups of classes. We didn't focus our attention on them because, as it can be seen from the intensity of their colored bands, those classes are weakly attracted so they do not represent extreme cases

Figure 5.4: *Internal Complexity vs. Interface* view applied on CodeCrawler

at:5 put:(Graphics.ColorValue scaledRed: 79 scaledGreen: 79 scaledBlue: 77);
at:6 put:(Graphics.ColorValue scaledRed: 39 scaledGreen: 37 scaledBlue: 37);
at:7 put:(Graphics.ColorValue scaledRed: 44 scaledGreen: 39 scaledBlue: 37);
at:8 put:(Graphics.ColorValue scaledRed: 45 scaledGreen: 35 scaledBlue: 39);
at:9 put:(Graphics.ColorValue scaledRed: 49 scaledGreen: 49 scaledBlue: 49);
...

The group marked 2 represents another kind of outliers. Classes, with high internal complexity, but with a very small interface. We look at the two classes and realize that they are classes implementing layouts, classes which have one single public method called *layout* (e.g. CCSingleClassBlueprint-Layout.layout, CCCorrelationViewLayout.layout). It is interesting to realize that one of the detected classes is the layout class we have implemented for extending CodeCrawler.

5.3.3 Methods In Need Of Refactoring

Purpose

System Maintenance, Restructuring

Applies On

Methods

Rationale

As Fowler suggests, a method shouldn't have a too big size because they will become too hard to understand. In these cases the "extract method" refactoring procedure should be applied.

View Definition

For this view we use three conformity expressions as attractors. They are enumerated below.

- **LOC > 100** attracts the methods with a big number of lines of code. We thought that methods having more than 100 LOC can be considered too big considering especially the 7 LOC per method rule used in smalltalk.
- **NI > 100** attracts the methods with a high number of invocations. Usually on a line we have at most an invocation. Therefore for 100 LOC a TI of 100 is a high value.
- **Normal Methods** is a conformity expression of the following form

$$Normal\ Methods\ =\ LOC < 100\ and\ NI < 100$$

  Its use is to attract the methods with normal values for LOC and NI so the outliers will be more easily seen.

Interpretation

There are four zones on this view. The first is the normal methods, which are grouped around the virtual axis perpendicular on the "Normal Methods" attractor in the lower half of the image.

The outliers with many lines of code but few invocations should be found on the axis that goes from "Normal Methods" attractor to the LOC > 100. The outliers with many invocations but few lines of code are to be found on axis between the NI > 100 and "Normal Methods" attractors. These two classes of outliers are usually special kinds of methods, like smalltalk window specifications or image data in the smalltalk language.

The outliers we are looking for corresponding to the methods that should be refactored, are to be found around the top of the imaginary perpendicular axis on the "Normal Method" attractor. This is so because the previously mentioned methods should be attracted by LOC and NI with comparable forces while in the same time be very faint attracted by the "Normal Methods" attractor.

Validation

This experiment will be done with an adventure game framework for smalltalk, called *AdventureBase*. Adventure base contains more than 100 classes corresponding to game logic and user interface. We want to see if we can detect methods in need of refactoring in this package.

We see that in the region where we expected to have methods in need for refactoring, we, indeed, have two obvious outliers. Looking at them we found

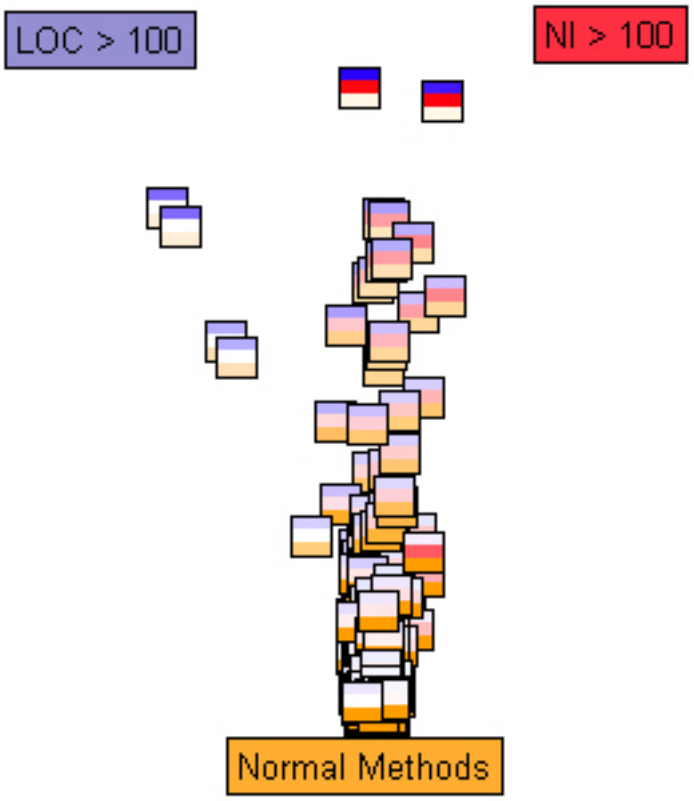

Figure 5.5: *Methods In Need Of Refactoring* view applied on AdventureBase

Machine class.lightSwitch and AdventureActor.addNonsenseHandlerTo: successBlock: successState methods. The two methods, having 90 and respectively 127 LOC, were too long to be listed here so we provided an appendix containing the first, shorter one. They also didn't have comments [2] so they were hard to understand. By applying the "extract method" we could bring the first mentioned method to a decent size of less than 30 non comment LOC and we believe, increase its comprehensibility.

It can also be seen from Figure **??** that we have four outliers with big LOC and small NI. All of them represent window specifications so they do not need refactoring.

Surely, we treated here only the extreme outliers, if we were to treat the not so obvious outliers, we could continue our search for methods in need of refactoring with other methods not so extremely positioned. However, we consider that we have already proved the usefulness of the view.

5.4 Integration In Moose

The ultimate goal of Magnet View is to provide a mean to analyze a system by visualizing the relationships between entities of the system and the properties of the entities. But how do we load a system in Magnet View? How do we select the entities we want to visualize? How do we select the attractors? What do the conformity expressions have to do with the visualization? These are the questions we plan to answer in this section.

The Magnet View is integrated in Moose. In Moose you can load a system and browse its entities in a browser that categorizes them. The entities are grouped in categories according to their entity type[3]. Each category has a menu associated with it. In this category menu, we have added an option that starts the Magnet View Runner. In Figure **??** we can see the menu for a group of classes

Therefore after Magnet View Runner is started, it has a reference to the system currently loaded in Moose and also to the group of entities that it is working with. Having a reference to the working set of entities, it can obtain through Moose all the defined properties that are available for that set of entities.

Those properties can be simple metrics or can be conformity expressions

[2] This also inspired us to think about a view which to corroborate the current expressions with information about the number of comment lines.

[3] Subgroups of these initial groups can also be generated by filtering mechanisms

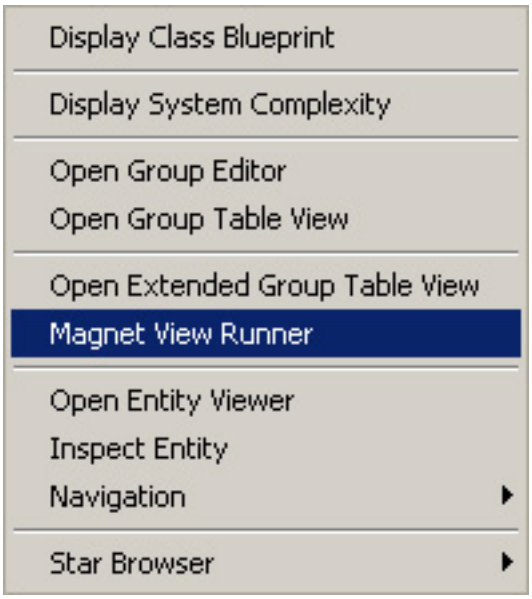

Figure 5.6: The menu attached to a class group from which the Magnet View Runner can be run.

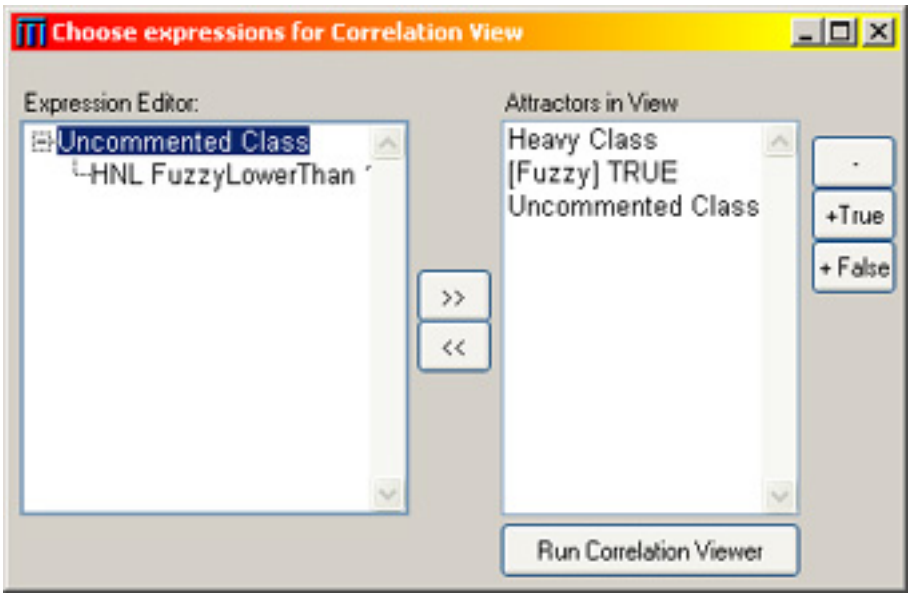

Figure 5.7: MagnetViewRunner the mediator between Moose and MagnetView

that are associated with a certain entity type[4]. The properties can be used as attractors themselves or can be composed using the conformity expressions mechanism with the use of the Magnet View Runner [5]

In Figure **??** we present now the MagnetViewRunner the mediator between Moose and the Magnet View visualization.

In the right side of the Magnet View Runner we have the attractors. We can define an attractor in the *Expression Editor*. The editing can be done by making use of the edited expressions' associated menu.

The operations executed on expressions in the expression editor with the help of the edit expression menu are

1. **Replace expression with another.** The possible expressions that can replace the current are presented in a menu which is dynamically

[4]See Appendix **??** for details on how a conformity expression can be defined as property of a specific entity type

[5]Here we see an important property of the conformity expressions. They can be composed themselves to form even higher-level compositions. The properties that are computed as conformity expressions can be used to form the basis of other conformity expressions in the Magnet View Runner expression editor we will describe in the following pages

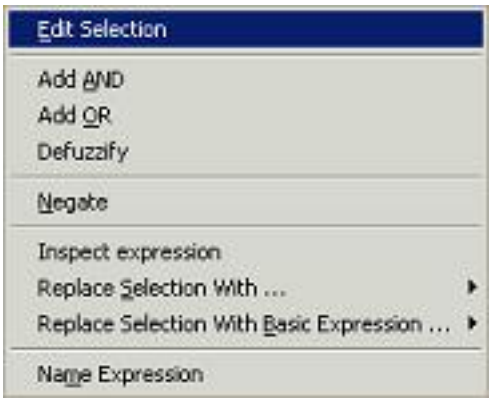

Figure 5.8: The expression menu lets the user edit the expressions in the editor. An expression can be replaced with another, made the child of another or have its parameters modified.

generated as all the combinations between the defined operators (i.e. HigherThan, LowerThan) and the metrics defined for the working set of entities.

2. **Make expression child of another.** This is used in the process of composing an expression. In order to compose two expressions by *And* the user should create the first expression then make the expression the child of an *And* expression and then edit the second child of the *And*

3. **Edit expression's parameters.** This is used because when executing operation 1 (*Replace expression with another*) the new expression has a default threshold. Therefore this default threshold should be adapted to the current situation.

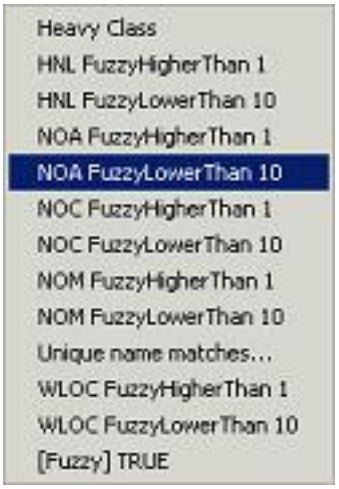

Figure 5.9: The menu that allows the replacement of an expression with another. The expressions in the image apply to classes.

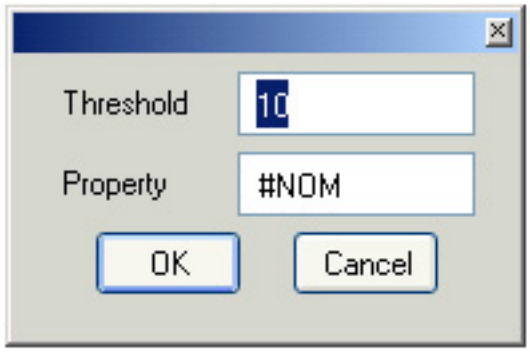

Figure 5.10: The editor for the parameters of the expressions

Chapter 6

Conclusions and Future Work

> "The black belt represents the beginning – the start of a never-ending journey of discipline, work, and the pursuit of an ever-higher standard" says the student.
> "Yes. You are now ready to receive the black belt and begin your work."
>
> *Chinese Parable*

As one of our professors used to say, a successful research work will raise more questions than answers. And as the student in the parable learnt, an achievement is not an end, is just the foundation for what is to come. This chapter is a good place to assess what has been done and show what remains to be done. We start by presenting what is original in the work, continue with some conclusions and end by presenting several possible continuation pathways that are opened by the work.

6.1 Contribution

The original contributions of the author to this paper are enumerated below

The Conformity Strategy Concept The idea of measuring the degree of conformity of software entities to the formal rules expressed by the conformity expressions is the main idea of this work. We have seen it

applied in different contexts through this work and we believe it is an interesting and potential mechanism. Maybe it still is a bit ahead of its time because the software engineering field is still new and many rules of thumb do not exist.

Composition Operators Implementation The implementation of the composition operators (presented in Chapter **??**) in a manner similar to the way the fuzzy logic operators are impl4emented, is a concept we didn't find anywhere else in the software reengineering literature.

The Magnet View Concept The visualization layout in which the software artifacts and several of their properties are positioned as a result of their interaction, interaction happening based on laws analogous with the magnetic ones, is a new concept and proved to be an effective technique of visualizing software systems.

6.2 Future Work

During our work we discovered possible continuation paths for it. Although we didn't have the time to explore them, we mention some of them here. They are grouped in two categories: the ones related to conformity expressions and the ones related to Magnet View.

6.2.1 Conformity Expressions

- **Argumentation Of Result**. Exactly the same way an expert system is expected to be able to make explicit the steps that led to one decision, the conformity strategies could be extended to explain the rating they assign to an entity. This means they should be able of presenting the information about the rating of each operator that composes an expression and not only the final rating. Therefore if we will have an *and* between an operator with a rating of 50 and one with a rating of 80, by making explicit the intermediary decisions, we will be able to know that there was a rating of 80 even if the final one is only 50. The question is, would this bring the user more useful information, or just more information?

- **Study Of The Evolution Of Ratings**. Studying the evolution of the ratings of an entity relative to a design rule we believe could expose relevant insights about it. This was done for the detection strategies [**?**] but with the supplementary information provided by the conformity

strategies the results could be even better. We might even be able to predict the evolution of the entity from the past evolution. An related application would be a real-time alert system integrated in a development environment, which would use the ratings and prediction algorithms to offer feedback as the user is developing the project. This would be great progress, because the development and the quality assessment would not be two separate steps anymore.

- **Alternative Conformity Operators**. We presented only one implementation for the conformity operators but there could be other implementations, also. Let's take HigherThan for example. We could devise an implementation which to use a logarithmic transfer function instead of a linear one. This would make the rating given for values bigger than the threshold not so preeminent.

- **Weight The Conformity Operators**. Because in composed a rule not all the clauses have the same importance, the conformity strategy mechanism would be closer to representing real design rules if it implemented a way of weighting it's component subexpressions. Nevertheless, it remains to be studied if the benefits of this enhancement pay for the increase in complexity.

6.2.2 Magnet View

- **Predefined View Suites**. The definition of views is the most challenging part of the utilization of Magnet View. As we already discussed in the previous section, working with thresholds is a difficult task. Therefore it would be great if we could provide a standard set of views to be used in the process of reverse engineering. The views we have presented are just examples and they lack the cohesion of a dedicated reverse engineering suite.

- **Magnet Maps**. This would be a very cool feature. We would have graphical representations of regions of a view. We could exactly define diagnostics related to each view. So if we have an entity positioned in a certain zone, we would be able to automatically diagnose it. This way we could eliminate the actual human reasoning about the position of the entities, human reasoning which is susceptible to errors.

Appendix A

Implementation Details

A.1 Conformity Expressions

The classes used in the implementation of the conformity expressions in Moose are treated below. We look first at the hierarchy under CFIntrospectionOperator (hierarchy that can be seen in Figure **??**) by taking each class and presenting the important facts about it.

CFIntrospectionOperator

CFIntrospectionOperator is the root of the class hierarchy responsible with implementing the operators. From its implementation we observe the constructor for its subclasses and the initialization method

```
SCG.Van defineClass: #CFIntrospectionOperator superclass:
#{SCG.Van.CFExpression} instanceVariableNames: 'block threshold '
category: 'VanModestCFOperators'

CFIntrospectionOperator class>>withBlock: aBlock
        withThreshold: aNumber
        ^self   new initialize
                setBlock: aBlock threshold: aNumber.

CFIntrospectionOperator>>setBlock: aBlock threshold:  aNumber
        block := aBlock.
        threshold := aNumber.
```

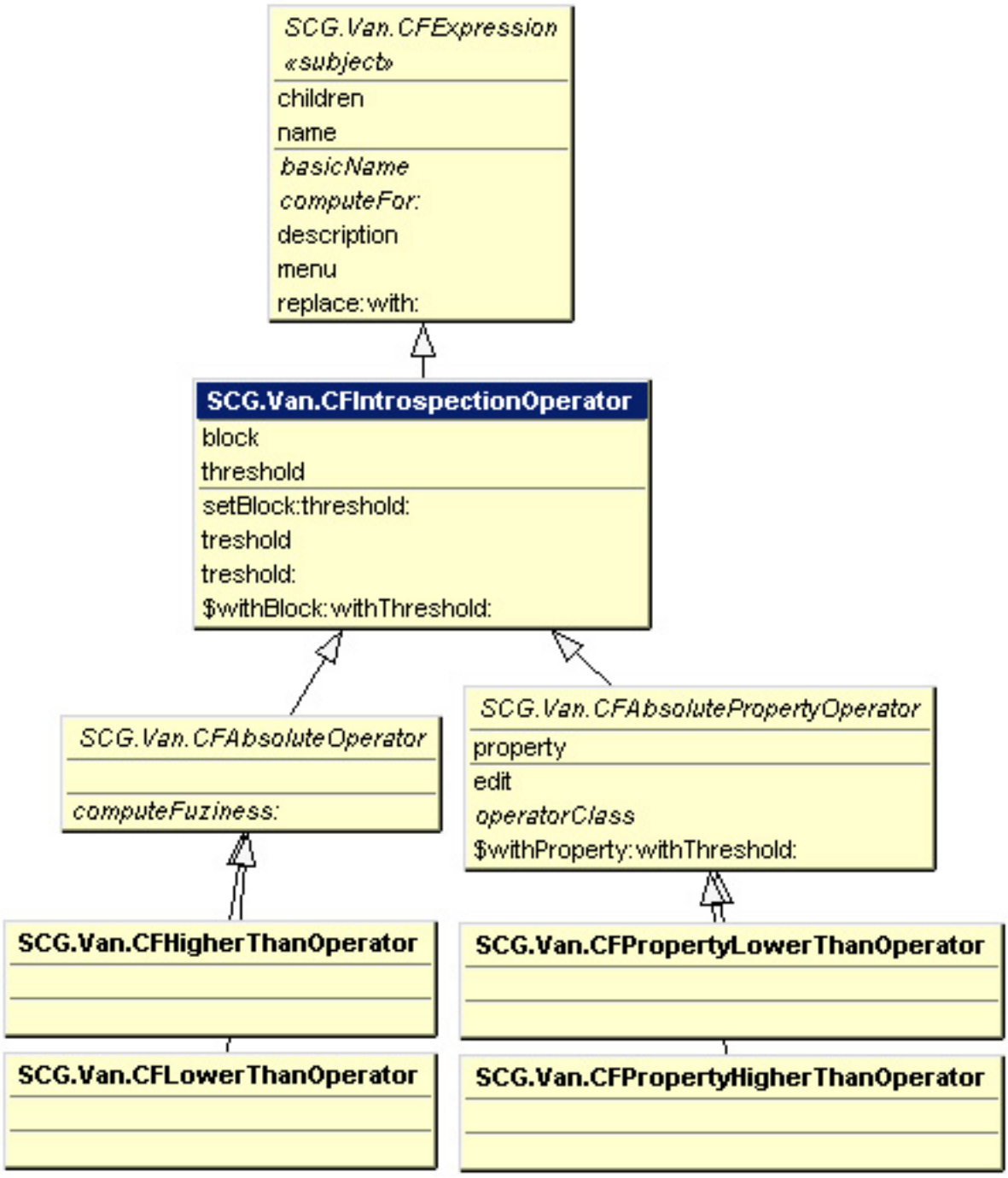

Figure A.1: The IntrospectionOperator Hierarchy

CFAbsoluteOperator

CFAbsoluteOperator is the class of the operators that take an absolute value as threshold. It is the base class for HigherThan and LowerThan operators and implements the common behavior for them. The common behavior is related to computing the rating for an entity. As we can see in the following code we use the *"template method"* design pattern [**?**]

```
SCG.Van defineClass: #CFAbsoluteOperator superclass:
#{SCG.Van.CFIntrospectionOperator} instanceVariableNames: ''
category: 'VanModestCFOperators'

CFAbsoluteOperator>>computeFor: anEntity
        ^self computeFuziness: (block value: anEntity).

CFAbsoluteOperator>>computeFuziness: aValue
        ^self subclassResponsibility.
```

CFHigherThan and CFLowerThan Operators

CFHigherThan and CFLowerThan must provide an implementation for the hook methods their superclasses defined. The most important is CFAbsoluteOperator>>computeFuzziness: aValue

```
SCG.Van defineClass: #CFHigherThanOperator superclass:
#{SCG.Van.CFAbsoluteOperator} instanceVariableNames: '' category:
'VanModestCFOperators'

CFHigherThanOperator>>computeFuziness: aValue
        threshold = 0 ifTrue: [^100].
        ^(aValue / threshold * 100) asInteger.

SCG.Van defineClass: #CFLowerThanOperator superclass:
#{SCG.Van.CFAbsoluteOperator} instanceVariableNames: '' category:
'VanModestCFOperators'

CFLowerThanOperator>>computeFuziness: aValue
        aValue = 0 ifTrue: [^0].
        ^(threshold / aValue * 100) asInteger
```

An example of usage for the CFLowerThan and CFHigherThan operators can be seen in the CFExpressionRepository class which is used to store conformity strategies. One such strategy is *GodClass*

```
CFExpressionRepository>>buildCFGodClassExpression
        | atfdHigherThan tccLowerThan wmcHigherThan godClassExpression noaHigherT

        atfdHigherThan := CFHigherThanOperator
            withBlock: [:class | class property: #ATFD]
            withThreshold: 40.
        wmcHigherThan := CFHigherThanOperator
            withBlock: [:class | class property: #WMC]
            withThreshold: 75.
        tccLowerThan := CFLowerThanOperator
            withBlock: [:class | class property: #TCC]
            withThreshold: 0.20.
        noaHigherThan := CFHigherThanOperator
            withBlock: [:class | class property: #NOA]
            withThreshold: 20.
        godClassExpression := CFAndOperator
            withLeft: (atfdHigherThan)
            withRight: (CFOrOperator
                withLeft: wmcHigherThan
                withRight: (CFAndOperator withLeft: tccLowerThan
                    withRight: noaHigherThan)).
        ^CFNamedOperator
            withName: 'God Class'
            withExpression: godClassExpression.
```

CFAbsolutePropertyOperator

This class and its two subclasses are used for a simpler use of the absolute operators when used programmatically. Normally, an expression should be parameterized with a block which takes an entity as parameter and computes the property that is to be used by the operator. Because usually the property is nothing more than a moose entity's property, the CFAbsolutePropertyOperator takes only the name of the property avoiding the more complex construction used otherwise.

```
SCG.Van defineClass: #CFAbsolutePropertyOperator superclass:
#{SCG.Van.CFIntrospectionOperator} instanceVariableNames:
'property ' category: 'VanModestCFOperators'

CFAbsolutePropertyOperator>>withProperty: aSymbol withThreshold:
```

```
aNumber
        ^(self new)
            property: aSymbol;
            threshold: aNumber

CFAbsolutePropertyOperator>>computeFor: anEntity
        | op |
        op := self operatorClass
            withBlock:
                [:entity |
                entity propertyNamed: property asSymbol]
            withThreshold: threshold.
        ^op computeFor: anEntity

CFAbsolutePropertyOperator>>operatorClass
        ^self subclassResponsibility.
```

Subsequently we look at the composition operators as they are represented in the Figure **??**

CFCompositionOperator

The composition operators take two expressions and compute the rating as a function of the ratings of the two expressions. We can see this from the constructor and from the CFCompositionOperator>>computeFor: method (method defined in the *public* protocol of the class).

The CFCompositionOperator>>computeFor: is implemented by delegating on subclasses the implementation of the hook method CFCompositionOperator>>fuzzyOperationOn:and:.

```
SCG.Van defineClass: #CFCompositionOperator superclass:
#{SCG.Van.CFExpression} instanceVariableNames: '' category:
'VanModestCFOperators'

CFCompositionOperator class>>withLeft: leftDSFuzzyExpression
withRight: rightDSFuzzyExpression
        ^self new initialize
            setLeft: leftDSFuzzyExpression
            right: rightDSFuzzyExpression
```

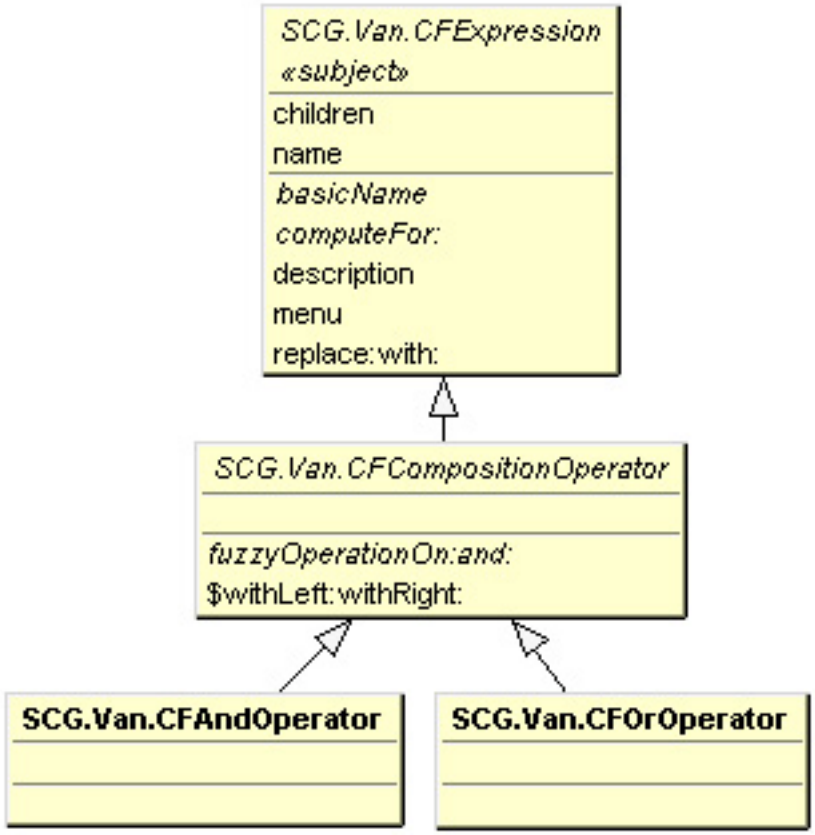

Figure A.2: The CompositionOperator Hierarchy

```
CFCompositionOperator>>setLeft: leftDSFuzzyExpression right:
rightDSFuzzyExpression
        children add:  leftDSFuzzyExpression.
        children add:  rightDSFuzzyExpression.

CFCompositionOperator>>computeFor: anEntity
        ^self fuzzyOperationOn: (self left computeFor: anEntity)
            and: (self right computeFor: anEntity)

CFCompositionOperator>>fuzzyOperationOn: firstPropertyValue and:
secondPropertyValue
        "Each subclass will implement its
        specific function definition in fuzzy logic. "
        ^ self subclassResponsibility.
```

The two subclasses of CFCompositionOperator have no more to do than implement the hook method called *operatorClass* as we see in the following code.

```
SCG.Van defineClass: #CFPropertyHigherThanOperator superclass:
#{SCG.Van.CFAbsolutePropertyOperator} instanceVariableNames: ''
category: 'VanModestCFOperators'

CFPropertyHigherThanOperator>>operatorClass
        ^CFHigherThanOperator.

SCG.Van defineClass: #CFPropertyLowerThanOperator superclass:
#{SCG.Van.CFAbsolutePropertyOperator} instanceVariableNames: ''
category: 'VanModestCFOperators'

CFPropertyLowerThanOperator>>operatorClass
        ^CFLowerThanOperator.
```

CFAndOperator and CFOrOperator

The CFAndOperator and CFOrOperator only have to implement the hook method provided by their parent class, namely CFCompositionOperator>>fuzzyOperationOn:and:. We can see that the implementation is straightforward

```
SCG.Van defineClass: #CFAndOperator superclass:
#{SCG.Van.CFCompositionOperator} instanceVariableNames: ''
```

```
category: 'VanModestCFOperators'

CFAndOperator>>fuzzyOperationOn: firstPropertyValue and:
secondPropertyValue
        ^firstPropertyValue min: secondPropertyValue.

SCG.Van defineClass: #CFOrOperator superclass:
#{SCG.Van.CFCompositionOperator} instanceVariableNames: ''
category: 'VanModestCFOperators'

CFOrOperator>>fuzzyOperationOn: firstPropertyValue and:
secondPropertyValue
        ^firstPropertyValue max: secondPropertyValue.
```

We end our presentation of the class hierarchy with the direct subclasses of CFExpression seen in Figure **??**

CFTrue and CFFalse

The CFExpression class is the root of a composite recursive structure. As leafs of such a structure we have the CFTrue and CFFalse classes which could have been as well defined as singletons. However, because the objects of this kind are very small we didn't consider implementing the singleton pattern. The only method in these classes' interface is computeFor: anEntity which without caring which is the entity always returns a value of 100 for True and 0 for False.

CFNamedExpression

In CFExpression, as it can be seen from the UML diagrams, there is a *name* method which returns *basicName* a subclassResponsibility. Each subclass of CFExpression implements its specific *basicName*. However there are times when an expression needs to have a distinct name that the one of its uppermost operator. For these situations CFNamedExpression wrapper keeps the functionality of the old expression but adds a name for the expression. This name can be modified via a graphical user interface.

```
SCG.Van defineClass: #CFNamedOperator superclass:
#{SCG.Van.CFExpression} instanceVariableNames: '' category:
'VanModestCFOperators'
```

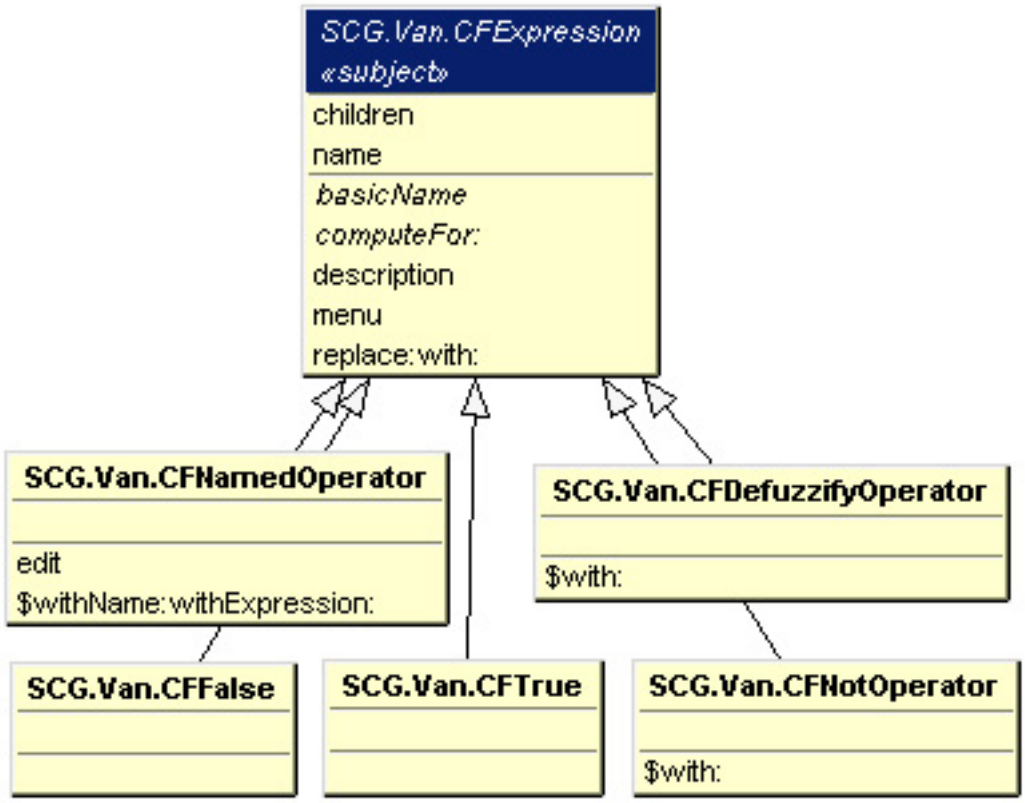

Figure A.3: Subclasses of CFExpression

```
CFNamedOperator>>withName: aName withExpression: anExpression
        ^(self new)
            name: aName;
            setExpression: anExpression;
            yourself.

CFNamedOperator>>printString
    ^self name.

CFNamedOperator>>computeFor: anEntity
        ^self children first computeFor: anEntity.

CFNamedOperator>>edit
        | editName |
        (SimpleDialog initializedFor: nil)
            setInitialGap;
            addMessage: 'Name'
            textLine: (editName := self name asValue)
            type: #string boundary: 0.4;
            addGap;
            addOK: [true];
            addGap;
            openDialog.
        self name: editName value.
```

CFDefuzzyfy Operator and CFNotOperator

CDefuzzyfyOperator is a discretization mechanism. It converts the fuzzy values of the conformity expressions to 0 and 100 corresponding to the CFTrue and CFFalse truth values. CFNotOperator inverts the value of the current subexpression, complementing the value relative to 100 if the value is less than 100 and resulting in 0 otherwise. Both operators are unary operators

```
SCG.Van defineClass: #CFDefuzzifyOperator superclass:
#{SCG.Van.CFExpression} instanceVariableNames: '' category:
'VanModestCFOperators'

CFDefuzzifyOperator>>with: anExpression
        ^self new initialize
```

```
            setChild: anExpression.

CFDefuzzifyOperator>>computeFor: anEntity
        (self children first computeFor: anEntity) >= 100
            ifTrue: [^100]
            ifFalse: [^0]

SCG.Van defineClass: #CFNotOperator superclass:
#{SCG.Van.CFExpression} instanceVariableNames: '' category:
'VanModestCFOperators'

CFNotOperator>>with: expression
        | newOp |
        newOp := self new.
        newOp children add: expression.
        ^newOp.

CFNotOperator>>computeFor: anEntity
        ^0 max: (100 - (children first computeFor: anEntity)).
```

CFExpressionRepository

CFExpressionRepository is, as its name suggests, a repository of expressions. We will not present all the expressions defined in the repository because of lack of space. We will look only at the way a *GodClass* conformity strategy is built

```
SCG.Van defineClass: #CFExpressionRepository superclass:
#{Core.Object} instanceVariableNames: '' category:
'VanModestCFOperators'

CFExpressionRepository>>buildCFGodClassExpression
        | atfdHigherThan tccLowerThan wmcHigherThan
        godClassExpression noaHigherThan |

        atfdHigherThan := CFHigherThanOperator
            withBlock: [:class | class property: #ATFD]
            withThreshold: 40.
        wmcHigherThan := CFHigherThanOperator
            withBlock: [:class | class property: #WMC]
```

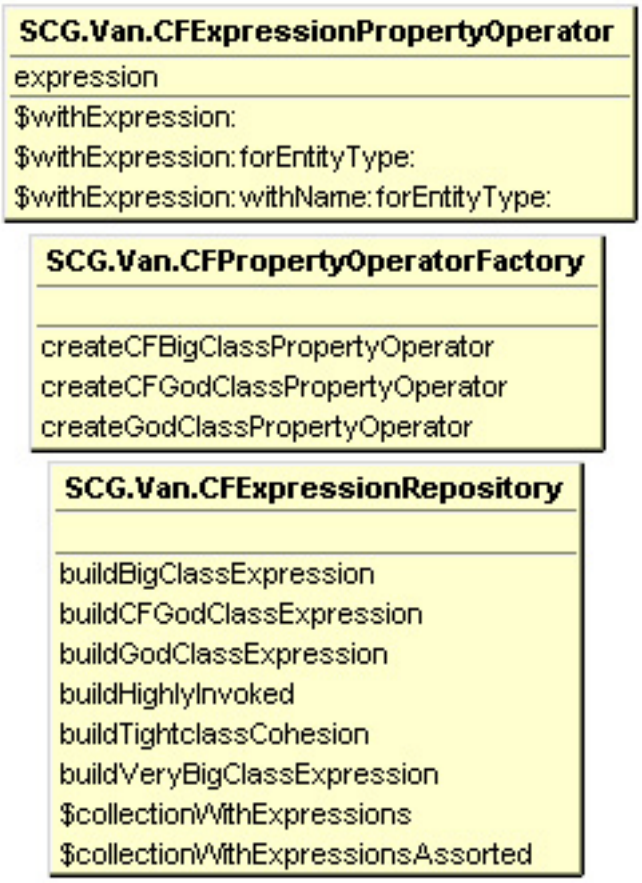

Figure A.4: Classes that connect to *Moose*

```
        withThreshold: 75.
    tccLowerThan := CFLowerThanOperator
        withBlock: [:class | class property: #TCC]
        withThreshold: 0.20.
    noaHigherThan := CFHigherThanOperator
        withBlock: [:class | class property: #NOA]
        withThreshold: 20.
    godClassExpression := CFAndOperator
        withLeft: (atfdHigherThan)
        withRight: (CFOrOperator
            withLeft: wmcHigherThan
            withRight: (CFAndOperator
                withLeft: tccLowerThan
                withRight: noaHigherThan)).
    ^CFNamedOperator
        withName: 'God Class'
        withExpression: godClassExpression.
```

CFPropertyOperatorFactory and CFExpressionPropertyOperator

CFPropertyOperatorFactory is a subclass of PropertyOperatorFactory. Every now and then the entity types are reinitialized. At the reinitialization of the entity types the properties associated to each of entity types are redefined. For their definition the hierarchy under PropertyOperator is searched for methods of the form *create*Operator* which are executed. These methods are the places where the new properties are defined.

We have defined a few conformity expressions as properties of the class entities. Therefore all the classes in a loaded model will have as property the rating for the corresponding expression. We look here at the definition of the *CFGodClass* conformity expression

```
SCG.Van defineClass: #CFPropertyOperatorFactory
    superclass: #{SCG.Moose.PropertyOperatorFactory}
    instanceVariableNames: ''
    category: 'VanModestCFOperators'

CFPropertyOperatorFactory>>createCFBigClassPropertyOperator
        ^CFExpressionPropertyOperator
            withExpression: CFExpressionRepository new buildBigClassExpression
            withName: #'BigClass [Modified]'
```

```
            forEntityType: #FAMIXClass.
```

We see in the definition of the *createCFBigClassPropertyOperator* method that we return an object of class CFExpressionPropertyOperator. This class is a kind of PropertyOperator. It is a special kind of PropertyOperator because one entity's property is computed as the rating of that entity conforming to an expression, expression by which the operator is parameterized.

```
SCG.Van defineClass: #CFExpressionPropertyOperator superclass:
#{SCG.Moose.PropertyOperator} instanceVariableNames: 'expression '
category: 'VanModestCFOperators'

CFExpressionPropertyOperator class>>withExpression: anExpression
withName: aString forEntityType: anEntityType
        ^self new
            initialize
            expression: anExpression;
            targetEntityType: anEntityType;
            propertyName: aString;
            yourself.

CFExpressionPropertyOperator>>computeFor: anEntity
        ^expression computeFor: anEntity.
```

A.2 Magnet View

In this section we will see some of the relevant details regarding the implementation of Magnet View. The most important thing to be understood is that we had to respect some constraints that CodeCrawler imposed in order to be able to extend the framework. The constraints were related to node plugins, layouts and node figures. All of them will be presented in this section.

Running CodeCrawler

After the user selects the expressions that will form the view in the Magnet View Runner all the necesary data is known for the view to be started. However for this there is a clear sequence of steps to be made as it can be seen in the runViewer method of the MagnetViewRunner class. The important steps in this method are detailed in the next sections.

```
SCG.Van defineClass: #MagnetViewRunner
    superclass: #{UI.ApplicationModel}
    instanceVariableNames: 'Expression Editor
    ExpressionEditor workingGroup viewExpressions '
    category: 'UIApplications-New'

MagnetViewRunner>>runViewer
    | cc graph |
    self viewExpressions listHolder value size = 0 ifTrue: [^self].
    PropertiesForCorrelationViewManager uniqueInstance
        expressions: viewExpressions listHolder value.
    graph := CCGraphManager uniqueInstance createNewGraph.
    CCModestPropertyNodePlugin generateAllPluginsForGraph: graph.
    CCModestGenericNodePlugin generateAllPluginsFor: workingGroup
        forGraph: graph.
    CCModestPropertyEdgePlugin generateAllPluginsForGraph: graph.
    self registerViews.
    cc := CodeCrawler.CodeCrawler new.
    cc spawnWithSpecName: 'Correlation View'
```

Creating the Graph

You see we first create a new graph. Then we let the Property nodes be created. Then the Entity nodes. For creating the entities we had to properly

subclass ItemNodePlugin as we can see in the UML diagram from figure

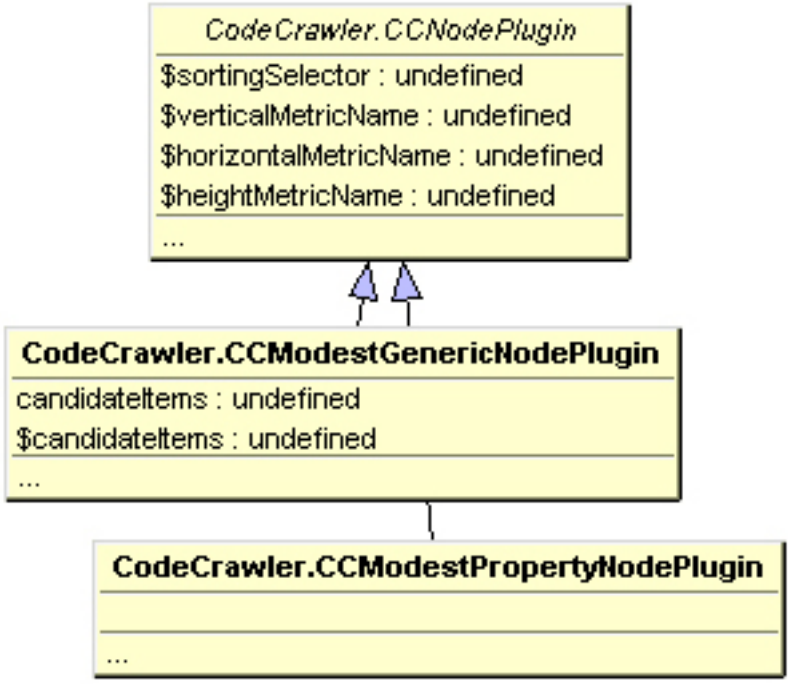

Figure A.5: To extend CodeCrawler we had to subclass CCNodePlugin

The generateAllpluginsAndGraphs is the hook method and it has the role of creating nodes for every entity and adding them to the graph. There are two variants each takes the graph as parameter but one has an extra parameter representing the entities for which there should be nodes created.

```
Smalltalk.CodeCrawler defineClass: #CCModestGenericNodePlugin
    superclass: #{SCG.Van.CCVanNodePlugin}
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    category: 'VanModestCodeCrawlerExtensions'

CCModestGenericNodePlugin class>>generateItemsAndPluginsForGraph:
aGraph
    self candidateItems do:
            [:each |
```

```
            | plugin |
            plugin := self new entity: each.
            aGraph addNode: (CCNode new plugin: plugin).
            self increaseProgressBar]


Smalltalk.CodeCrawler defineClass: #CCModestPropertyNodePlugin
    superclass: #{CodeCrawler.CCNodePlugin}
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    category: 'VanModestCodeCrawlerExtensions'

CCModestPropertyNodePlugin>>generateItemsAndPluginsForGraph: aGraph
    self candidateItems do: [:eachExpression | | plugin |
        plugin := (self new entity: eachExpression).
        aGraph addNode: (CCNode new plugin: plugin).
        self increaseProgressBar.
    ].

Smalltalk.CodeCrawler defineClass: #CCModestPropertyEdgePlugin
    superclass: #{CodeCrawler.CCEdgePlugin}
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    category: 'VanModestCodeCrawlerExtensions'

CCModestPropertyEdgePlugin>>generateItemsAndPluginsForGraph:
aGraph
    self withPropertyNodesFor: aGraph
        do:
            [:propertyNode |
            self withGenericNodesFor: aGraph
                do:
                    [:genericNode |
                    | newPlugin edge |
                    edge := CCEdge
                        fromNode: propertyNode
                        toNode: genericNode.
                    newPlugin := self new
                                entity: (propertyNode plugin entity
                                computeFor: genericNode plugin
```

```
                    entity).
        edge isNil
            ifFalse:
                [edge plugin: newPlugin.
                aGraph addEdge: edge]]]
```

Now that the graph is generated there is need to associate figures with the nodes and arrange those figures conforming to the attraction laws. This is done in the layout class. The layout class is also subclassed from a layout class provided by the framework, namely CCGenericLayout.

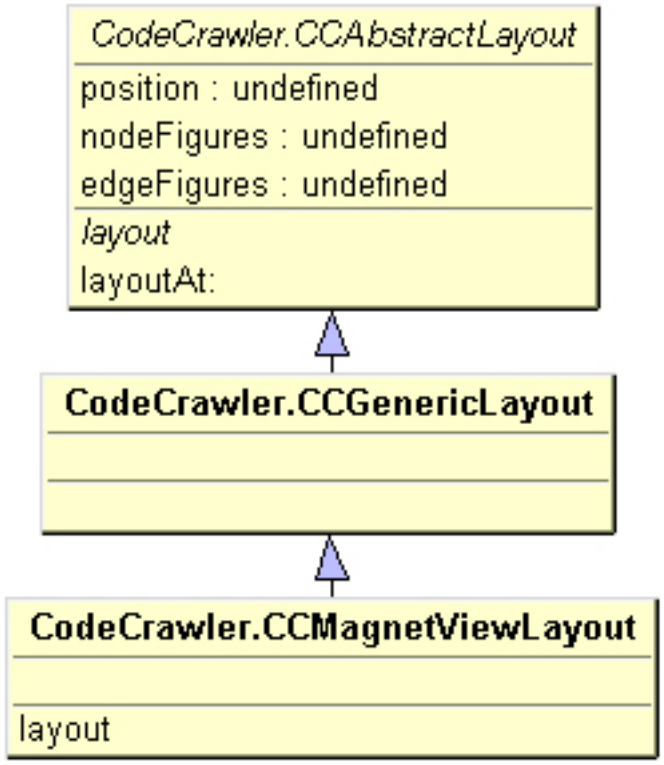

Figure A.6: Class CMagnetViewLayout and position in hierarchy. It is easy to see from the diagram that the only abstract method defined in CCAbstractLayout is layout which is also the only public method in CMagnetViewLayout.

Creating a new layout

As it can be seen from the diagram, there is only one hook method which is *layout*. From its code it can be seen that it positions first the attractors and then the attractees. The attractors are arranged in a circle and the attractees ar arranged according to the algorithm presented in the chapter describing the functionality of MagnetView.

```
CCMagnetViewLayout>>layout
    self positionProperties.
    self positionEntities.

CCMagnetViewLayout>>positionProperties
        self coloredCircleWithFigures:
            self allPropertyNodeFigures andRadius: 320

CCMagnetViewLayout>>coloredCircleWithFigures: figures
andRadius: rad
    | angleTemp angle center |
    center := Point x: rad y: rad.
    angleTemp := 2 * Float pi / figures size.
    angle := 0.
    figures do:
            [:each |
            | point color |
            point := center + (Point r: rad theta: angle).
            angle := angle + angleTemp.
            color := ColorValue
                        hue: angle / (2 * Float pi)
                        saturation: 0.8
                        brightness: 0.99.
            each color: color.
            each translateTo: point]

CCMagnetViewLayout>>positionEntities
    | xCoord yCoord |
    self allEntityNodeFigures do: [:entityNodeFigure | entityNodeFigure figure resetBands]
    self allEntityNodeFigures do:
            [:entityNodeFigure |
            xCoord := self computeXPositionFor: entityNodeFigure.
            yCoord := self computeYPositionFor: entityNodeFigure.
```

```
            entityNodeFigure translateTo: xCoord @ yCoord.
            self allPropertyNodeFigures do:
                    [:each |
                    | expression entity conformity |
                    expression := each item plugin entity.
                    entity := entityNodeFigure item plugin entity.
                    conformity := expression computeFor: entity.
                    entityNodeFigure figure
                        addBandWithColor: each currentColor hue
                        andIntensity: (1 min: conformity / 100)].
                    entityNodeFigure zIndexPriority:
                    entityNodeFigure figure totalIntensity].
    self disperseEntities.
```

Appendix B

Refactoring Methods View

One of the methods in need of refactoring detected with the help of *Methods In Need Of Refactoring* view was Root::Smalltalk::Machine class.lightSwitch. We list it here

```
lightSwitch
    "Answer an example state machine."
    "
    | myMachine |
    myMachine := Machine lightSwitch.
    myMachine event: (MachineEvent named: #on).
    myMachine event: (MachineEvent named: #off).
    myMachine event: (MachineEvent named: #toggle).
    myMachine
    "

    | switch |
    switch := (Machine named: #lightSwitch)
        add: (MachineState named: #on);
        add: (MachineState named: #off);
        yourself.

    (switch atState: #on)
        addEntryBlock: [:machine |
            Transcript cr; show: 'Entering ' , machine currentState printString];
        addReentryBlock: [:machine |
            Transcript cr; show: 'Re-entering ' , machine currentState printString];
        add: (
```

```
            (MachineTransitions forEvent: #on)
                add: (
                    MachineTransition new
                        actionBlock: [:machine :event |
                            Transcript cr; show: 'Light is already on: ' ,
                                machine printString , ' ' , event printString]
                    )
        );
        add: (
            (MachineTransitions forEvent: #off)
                add: (
                    MachineTransition new
                        actionBlock: [:machine :event |
                            Transcript cr; show: 'Light being turned off: ' ,
                                machine printString , ' ' , event printString];
                        nextState: (switch atState: #off)
                )
        );
        add: (
            (MachineTransitions forEvent: #toggle)
                add: (
                    MachineTransition new
                        guardBlock: [:machine :event |
                            Dialog confirm: 'Really toggle?'];
                        actionBlock: [:machine :event |
                            Transcript cr; show: 'Light being toggled off: ' ,
                                machine printString , ' ' , event printString];
                        nextState: (switch atState: #off)
                )
        ).

    (switch atState: #off)
        addExitBlock: [:machine |
            Transcript cr; show: 'Exiting ' , machine currentState printString];
        add: (
            (MachineTransitions forEvent: #on)
                add: (
                    MachineTransition new
                        actionBlock: [:machine :event |
                            Transcript cr; show: 'Light being turned on: ' ,
```

```
                        machine printString , ' ' , event printString];
                    nextState: (switch atState: #on)
            )
    );
    add: (
        (MachineTransitions forEvent: #off)
            add: (
                MachineTransition new
                    actionBlock: [:machine :event |
                        Transcript cr; show: 'Light is already off: ' ,
                            machine printString , ' ' , event printString]
            )
    );
    add: (
        (MachineTransitions forEvent: #toggle)
            add: (
                MachineTransition new
                    guardBlock: [:machine :event |
                        Dialog confirm: 'Really toggle?'];
                    actionBlock: [:machine :event |
                        Transcript cr; show: 'Light being toggled on: ' ,
                            machine printString , ' ' , event printString];
                    nextState: (switch atState: #on)
            )
    ).

^switch
```

After looking at it for understanding, we applied the "Extract Method" refactoring. Eventually we have decreased its size and in the same time increased its comprehensibility. The increase in comprehensibility is due to the simpler structural complexity and to the tips introduced by applying suggestive names for the extracted methods.

```
lightSwitch
    "Answer an example state machine."

    "
    | myMachine |
    myMachine := Machine lightSwitch.
```

```
myMachine event: (MachineEvent named: #on).
myMachine event: (MachineEvent named: #off).
myMachine event: (MachineEvent named: #toggle).
myMachine
"

| switch |
switch := self createSwitch.
(switch atState: #on)
    addEntryBlock: self entryBlock;
    addReentryBlock: self reentryBlock;
    add: ((MachineTransitions forEvent: #on)
                add: self transitionFromOnToOn);
    add: ((MachineTransitions forEvent: #off)
                add: (self transitionFromOnToOff: switch));
    add: ((MachineTransitions forEvent: #toggle)
                add: (self transitionFromOnToggle: switch)).
(switch atState: #off)
    addExitBlock: self exitBlock;
    add: ((MachineTransitions forEvent: #on)
                add: (self transitionFromOffToOn: switch));
    add: self transitionFromOffToOff;
    add: ((MachineTransitions forEvent: #toggle)
                add: (self transitionFromOffToggle: switch)).
^switch
```

Bibliography

[BECK] K. Beck. **Smalltalk. Best Practice Patterns** Prentice Hall, 1997.

[BROO] F. P. Brooks. **The Mythical Man-Month**. Addison-Wesley, Reading, Mass., 1975.

[BIMW] T. Biggerstaff, B. Mitbander, D. Webster. **Program Understanding and the Concept Assignment Problem**. Communications of the ACM, May 1994.

[CHIK] E. J. Chikofsky and J. H. Cross II. **Reverse Engineering and Design Recovery: A Taxonomy**. IEEE Software, pages 1317, January 1990.

[FAMO] H. Bar, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. **The FAMOOS Object-Oriented Reengineering Handbook**.
European Union under the ESPRIT program Project no. 21975 (FAMOOS), October, 1999.

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

[FENT] N. Fenton and S. L. Pfleeger. **Software Metrics: A Rigorous and Practical Approach**. International Thomson Computer Press, London, UK, Second edition, 1997.

[FOOT] B. Foote and J. W. Yoder. **Big Ball of Mud**. In Proceedings of PLop97, 1997.

[GHJV] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1994.

[HSEL] B. Henderson-Sellers. **Object Oriented Metrics. Measures Of Complexity**. Prentice Hall, 1996

[JOHN] R. E. Johnson. **Documenting Frameworks using Patterns**. In Proceedings OOPSLA 92, ACM SIGPLAN Notices, pages 6376, October 1992.

[LANZ] M. Lanza. **Combining Metrics and Graphs for Object Oriented Reverse Engineering**. Institut fur Informatik und angewandte Mathematik, Universitat Bern.

[MARN] R. Marinescu. **Measurement and Quality in Object-Oriented Design** (PhD thesis). "Politehnica" University of Timisoara, 2002

[MOOS] Stephane 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.

[PARN] D. L. Parnas. **Software Aging**. Department of Electrical and Computer Engineering McMaster University, Hamilton, Ontario, Canada Addison-Wesley, 1999.

[PRICE] Price, B.A., Baecker, R.M., and Small, I.S. **A Principled Taxonomy of Sofware Visualization**. Journal of Visual Languages and Computing

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

[RDGR] D. Ratiu, S. Ducasse, T. Girba, R. Marinescu. **Using History Information to Improve Design Flaws Detection**. In *Proceedings of CSMR, 2004*.

[RATI] D. Ratiu. **Time-Based Detection Strategies**. Diploma Thesis at *Politehnica Timisoara*.

[SMC] W. P. Stevens, G. J. Meyers, L.L. Constantine. **Structured Design**. IBM Systems Hournal, 1974.

[WEIS] Mark Weiser. **Programmers Use Slicing When Debugging**. Commununications of the ACM, 1982.