# Measurement and Quality in Object-Oriented Design

BY
Radu Marinescu

Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Faculty of Automatics and Computer Science of the
"Politehnica" University of Timişoara

Timişoara,
October 2002

Advisors:
Prof. Dr. rer.nat. Gerhard Goos
Prof. Dr. ing. Ioan Jurca

# Measurement and Quality in Object-Oriented Design

BY
Radu Marinescu

Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Faculty of Automatics and Computer Science of the
"Politehnica" University of Timişoara

Timişoara,
October 2002

Advisors:
Prof. Dr. rer.nat. Gerhard Goos
Prof. Dr. ing. Ioan Jurca

*What is not measurable make measurable.*

**Galileo Galilei (1564 - 1642)**

# ACKNOWLEDGMENTS

I consider myself fortunate and privileged to have Prof. Gerhard Goos as one of my advisors. I am deeply indebted to him for shaping my path to research by guiding me with his extensive knowledge and with his insightful discussions and questions. At the same time, I want to thank him for the whole financial support that made possible my regular research visits at Forschungszentrum Informatik (FZI) Karlsruhe during all these years.

I am honored and especially grateful to have Prof. Ioan Jurca as the other advisor of my thesis. I am particularly thankful for all the essential guidance and permanent encouragement that I received from him. I owe him so much for every occasion in which he trusted me fully, even when I shared with him my craziest dreams. Without his full support, probably none of those dreams would have become reality.

My thankful admiration goes to Dr. Joachim Weisbrod for decisively influencing the beginning of my scientific career. Without his guidance, his support, and his enthusiasm for all of my early achievements, I would have never reached the point where I stand today.

I owe a lot of gratitude to Doru Gîrba for all our fascinating and fruitful discussions (especially during meals). I am very thankful for the enthusiasm with which he shared and enlarged my vision, for permanently motivating and helping me to go on. Without him, most of my visions would have remained abstract, and my ideas shallow. Thank you Doru for the privilege of collaborating with you, and for being the great friend that you are.

This work would not have been possible without the whole environment in the PROST research group at FZI Karlsruhe. First of all, I want to thank Benedikt Schulz for the many times he challenged me when I was losing focus, but also for encouraging me every time I was struggling. His highly competent advices, his permanent availability and his genuine friendship will always remain very precious to me. I greatly enjoyed working with Markus Bauer. Thank you, Markus, for all the fruitful discussions and for all the great help I got from you on such many occasions. I am very grateful for the privilege of having worked together with all my other colleagues at PROST, especially with Christoph Andriessens, Holger Bär, Dr. Oliver Ciupke, Jörn Eisenbiegler, Thomas Genßler, Dr. Helmut Melcher, Laura Olson and Olaf Seng. I want also to thank Dr. Thomas Lindner for his support and encouragement while he was at FZI, but also after he moved. Last, but not least, I want to thank Mrs. Gabriele Groß and Hilke Meffert for the miraculous way in which they always succeeded in solving my complicated administrative/bureaucratic issues.

Working with the members of the LOOSE research group (LRG) in Timişoara is such an extraordinary experience. I want to thank them for all the lessons I learnt as a result of our thrilling discussions both in regular meetings and in private talks. The team-spirit in LOOSE and the maturity of its members is so very refreshing for me. Individually, I want to thank Daniel Raţiu and Mircea Trifu for the three years we spent together working on the implementation and

refinement of the meta-model that eventually became part of this thesis. I want to thank them both for always giving their best and sacrificing their extra-time just for the joy of working together. I am indebted to Dani in a special way for challenging and assisting me continuously through the last days (and nights) to make this thesis syntactically "perfect". It is hard to express how much I owe Răzvan Filipescu for reviewing most of this thesis. His amazing English skills, doubled by his special patience in reading essentially improved the quality of the written text and lent it the current elegance of expression. I also want to express my sincere thanks to Iulian Dragoş and Petru Mihancea for their highly competent and careful reviews of several parts of the thesis, which improved the writing from both a semantical and a syntactical point of view.

Special thanks to Adrian Trifu for all our unforgettable discussions that essentially sharpened my arguments and for his remarkable reviews of several parts of this thesis. I also want to thank Adi for introducing me to the fascinating world of space shuttle launchings and for all great time we had in Karlsruhe eating broasted chicken and ice-cream.

Sincere thanks to Ciprian Chirilă and Antonio Rusu for their great contribution to implementing and refining my ideas. Working together with them helped me a lot in making these ideas applicable.

Very special thanks to all my friends who didn't let me forget that life is more than writing a PhD thesis. I am deeply indebted to Martin and Inge Linder for all their precious love in the Lord and for making me feel at home every time I was staying with them in Karlsruhe. I want to thank Radu and CăTă, and of course Liana, for sharing over the years all my joys and disappointments and for all encouragement and refreshment I received through them in so many ways.

I would like to warmly thank my parents for all their love, for all their mental and financial support and for believing in me even when I didn't. In particular I am very thankful to my mom for the "academic" answers to all the profound questions of my childhood and to my dad for teaching me to approach all things systematically.

My most special thanks are for Cristina for the unique way in which she loves and understands me day by day. I am so grateful to her for the many times she put aside her wishes and plans, and encouraged me to finish writing this thesis. Thank you for teaching me daily the "mysterious equations of love". I am so proud and grateful to have you by my side.

Above all, I thank God for all the talents He blessed me with, for His merciful and faithful guidance over all these years and, most importantly, for saving me and granting me a spirit of wisdom and revelation to see His plan. After all, it is only this that fundamentally counts.

Timişoara,
October 31, 2002

Radu Marinescu

# Contents

# Chapter 1

# Introduction

The gap between qualitative and quantitative statements, concerning object-oriented software design can be bridged using higher-level, goal-driven methods for measurement interpretation.

*The goal of this work is to develop methods and techniques that provide a relevant interpretation of measurement results applied to the investigation of object-oriented software design.* In this context, the central focus of this dissertation is to support the quality assessment and improvement of existing object-oriented systems, by bridging the gap between qualitative and quantitative statements.

## 1.1 Problem Definition

As Opdyke pointed out: "design is hard" [Opd92]; and object-oriented design in spite of all the optimistic claims is not a bit easier. In addition to the inherent difficulties of design, software industry began to place much more emphasis on the quality of software products. In order to improve its quality we need to be able to control it. This has in turn led to an increasing interest in software measurement. Thus, the context and the motivation for this work come from the fields of object-oriented design and software measurement. We will therefore discuss them next pointing out the issues that are relevant for this work.

### 1.1.1 Object-Oriented Design

Object-oriented programming supports essential software development goals like maintainability and reusability [Mey88] [Boo94] through mechanisms like encapsulation of data, inheritance and dynamic binding. Yet in the beginning of object-orientation many software companies had the naïve dream that the use of objects will automatically increase the quality of their software and greatly decrease the time spent on development and maintenance. It was hoped that by introducing object-oriented mechanisms software systems will become more flexible, more extensible and understandable, and that they will therefore be

easier to maintain.

Today, the industry is confronted with a large number of software systems in use, in the size of millions of lines of code. By their inherent size, complexity and development times, they have reached the suitable shape for object-orientation. Yet, most of these systems lack all of the aforementioned qualities: they are instead monolithic, inflexible and hard to extend. We identified the following *causes* for this situation:

- *Time Pressure.* Often systems tend to start with a clear and rigorous design, but then the developers are confronted with a lot of time constraints. This fight against the clock forces them to choose the fastest design solution and not the one that keeps the integrity of the design.
- *Changing Requirements.* Requirements change in ways that cannot be anticipated in the initial design. These changes often require essential modifications on the architectural level. In many cases those who implement the changes are not aware of the initial design and because of this the design becomes blurred.
- *Immature Object-Oriented Designers.* Most legacy systems of the "first generation" were written by programmers that had less understanding of the principles of object-oriented design. The ignorance of those principles has led to a lot of poorly designed code.

As a conclusion, we may state that object-oriented programming is a basis technology, that supports quality goals like maintainability and reusability but just knowing the syntax elements of an object-oriented language or the concepts involved in the object-oriented technology is far from being sufficient to produce good software. A good object-oriented design needs design rules and practices that must be known and used. Their violation will eventually have a strong impact on the quality attributes. The market forces emphasize more and more the need for quality in software system, but as De Marco points out: "you cannot control what you cannot measure"[DeM82]. Thus the question that comes to our mind is: can we quantify by the interpretation of software measures the principles and rules of object-oriented design?

### 1.1.2 Software Measurement

Software measurement is concerned with mapping attributes of a software product or process to a numeric value. These numbers become relevant concerning quality when compared to each others or to standards [Som95]. This statement touches a key issue of measurement interpretation: are measurement data relevant by themselves, or do numbers become relevant only in the context of an interpretation model? This thesis aims to show that numbers become relevant to the assessment of design quality only when they are related to the principles and rules of good design.

As already stated, the context of this work is intimately related to one of the major goals in software engineering, namely improving the control of the software development process. This goal can be concretely attained at the code and design level by:

- improving the quality of software products;
- anticipating and reducing maintenance requirements
- evaluating the productivity impacts of new tools and techniques

As an immediate result of this direction in software engineering research, software metrics have been brought to the attention of many software engineers and researchers. Software metrics can provide a quantitative means to control the quality of software products. Back in 1990 Card already emphasized [CG90] that metrics should be employed for the evaluation of of software design from a quality point of view.

In spite of the important role played by software measurement, there is still a gap between how we *do* measure and how we *could* measure. Some of the current problems with software measurement are discussed below:

- There are a large number of metrics, to the point where we can really speak about an inflation of metrics. But there are a lot of problems with the definitions of metrics: oftentimes metrics definitions are imprecise, confusing or incomplete which makes most of them hardly usable [Mar97a] [BDW99]
- Above the problems concerning metrics definitions are the issues related to the *interpretation* of measurement results. In most of the cases no interpretation model or very empirical ones are provided, so that the applicability and reliability of measurement results is drastically hampered.
- Concerning the interpretation of measurement results, there is another issue: in most cases individual measurements used in isolation do not provide relevant clues regarding the cause of a problem. In other words, a metric value may indicate an anomaly in the code but it leaves the engineer mostly clueless concerning the ultimate cause of the anomaly. Consequently, this has an important negative impact on the relevance of measurement results.

### 1.1.3 Problems

The previous experiences of applying metrics for the assessment and improvement of design quality in object-oriented systems are encouraging [Ern96], [Mar01], [LD01] but, as we have seen before, this approach raises a set of new problems. These problems are summarized below:

- The interpretation of individual measurements is too fine-grained if metrics are used in isolation; this reduces the applicability and relevance of metrics usage.
- There is a large gap between design principles and product measures. In other words there is still an important gap between what we measure and what is important in design. Thus, we miss a methodology for expressing principles and rules of design in a quantifiable manner.
- There exists a lack of relevant feedback link in quality models. In other words, we measure, we set thresholds, we identify suspects, but the metric by itself does not provide enough information for a transformation of the code that would improve quality. Thus, the developer is provided only with the problem and he or she must still empirically find the real cause and eventually look for a way to improve the design.
- Many metric-based approaches related to the quality of the design use only simple metrics, mainly because the meta-model used as a basis for the computation of metrics is incomplete. It mostly misses cross-referencing information, which in our view plays an important role in the quality of a system design.

## 1.2 Goal and Approach

*The goal of this work is to develop methods and techniques that provide a relevant interpretation of measurements applied to the investigation of object-oriented software design.* In this context, the central focus of the thesis is to support the quality assessment and improvement of existing object-oriented systems, by bridging the gap between qualitative and quantitative statements.

A complementary goal of this dissertation is to define the methods and techniques in a manner that would make them useful for the improvement of the design in newly built systems. In other words, we want the software engineer that uses this approach to be provided with explicit and quantifiable indications on what good design is, indications that would allow him to better design the next system he or she will be working on.

In order to reach these goals we proceed as follows: we first provide a mechanism for interpreting measurement results both in isolation (the filtering mechanism) and in correlation (the composition mechanism). The synthesis of these mechanisms is a higher level interpretation means for measurement results. This is called *detection strategy*.

Having defined this higher-level mechanism for properly extracting and interpreting relevant results from a multiple set of measurement data, we prove that it can be employed for different investigation goals and describe an usable approach for doing so.

As we already stated, our main focus is to provide an interpretation model that would allow us to bridge the gap between *qualitative statements* (i.e. quality as it is perceived from outside the system) and *quantitative statements* (i.e. quality as it is measurable at the source-code level). Therefore, after defining the *detection strategy* mechanism, we will employ it for assessing the quality of object-oriented systems, in a manner that will support also the eventual improvement of the quality. For that a two steps approach can be used:

- **Step 1**: We will prove that design problems in the source-code can be detected at different granularity levels, starting from the method-level, up to micro-architectural flaws (e.g. the lack of applying design patterns). The detection of design problems is based on defining quantified expressions of design principles and heuristics and looking for the design fragments that violate these rules.
- **Step 2**: We propose a mechanism for defining a new type of quality-model based on detection strategies. This new approach has two main advantages over current approaches: the first is the increased simplicity of constructing and understanding such quality trees in which quality is expressed in terms of design principles; the second advantage is an enhanced ability to interpret quality models, because the new approach uses a higher level interpretation mechanism, i.e. the detection strategies.

Concerning the use of this approach for design improvement for newly built systems, the suite of detection strategies for the detection of design problems provides the engineer with the means to learn in a concrete and repeatable manner what is good and what is bad in object-oriented design.

## 1.3 Organization

Chapter 2 defines a precise conceptual framework that will be used further on in this dissertation. The framework covers on the one hand the fundamental concepts used in the object-oriented paradigm and on the other hand it introduces the key concepts of software measurement. Chapter 3 contains a state of the art in the fields of measurement and quality assurance related to object-oriented design. After pointing out the limitations of current approaches, at the end of Chapter 3, the goals and requisites of the thesis are defined.

A new mechanism for defining measurements interpretation rules of any complexity, called detection strategy, is defined in Chapter 4. Together with this mechanism we introduce a method for quantifying informal rules related to design, i.e. transforming such rules into detection strategies. While the main focus of Chapter 4 is on defining a proper mechanism and a method for relevant measurement interpretation, Chapter 5 employs the new interpretation mechanism for detecting design flaws in object-oriented systems. We identify several abstraction layers on which detection strategies can be applied and propose a

set of such strategies for each layer. Chapter 6 defines a new quality model based on detection strategies, as the bridge between qualitative and quantitative statements. A mechanism for describing such models is provided along with the definition of a concrete quality model for evaluating maintainability.

Chapter 7 presents the tool kit designed to support the automatization of the entire approach and evaluate the methods and techniques based on a relevant case-study. Chapter 8 summarizes the thesis with its essential contributions and points out to ongoing issues.

# Chapter 2

# Object-Oriented Design and Measurement

As pointed out in the first chapter this work is going to tackle the issue of using measurement in order to assess and control the quality of object-oriented design. This dissertation is therefore related to two major fields: software measurement and object-oriented design. The goal of this chapter is to present the foundations of these domains. During the rest of this work we will refer to the concepts introduced here.

The chapter is structured in three parts: the first part is introducing object-oriented programming together with its key mechanisms for the management of software complexity. In the second part of this chapter we concentrate on the question of what does good object-oriented design and on the criteria for obtaining and assessing it. After discussing the object-oriented design, the last part of this chapter deals with the foundations of software measurement. During this final part the special emphasis lies on the aspects related to the interpretation of measurement results. The final section is dedicated to product metrics defined for the object-oriented paradigm.

## 2.1 Object-Oriented Programming

The essential factor that influenced the evolution of programming paradigms was the necessity to deal with the increasing complexity of software programs [Sch97] [CY91b]. Object-oriented programming provides us with a set of proper mechanisms for the management of this complexity, namely: data abstraction, encapsulation, modularity, inheritance, and polymorphism. In this section we will discuss these mechanisms.

Booch defined object-oriented programming as follows [Boo94]:

**Definition 2.1** *Object-oriented programming is an implementation method in which programs are organized in object collections that cooperate among themselves, each object representing an instance of a class; each class is part of a class hierarchy and all classes are related through their inheritance relationships.*

Analyzing the definition above we find three important elements of object orientation:

- objects and not algorithms are the fundamental logical blocks;
- each object is an instance of a class;
- classes are linked among themselves through inheritance relationships.

In the context of the previous definition, we can now introduce Sommerville's definition [Som95] of *object-oriented design*:

**Definition 2.2 (Object-Oriented Design)** Object-oriented design *is a design strategy where system designers think in terms of 'things' instead of operations or functions. The executing system is made up of interacting objects that maintain their own local state and provide operations on that state information.*

### 2.1.1 Data Abstraction

One of the fundamental ways used by all people in order to understand and comprehend a complex issue is by using abstractions. A good abstraction is one that underlines all the aspects that have relevance to the perspective from which the object is being analyzed while at the same time suppressing or diminishing all the other characteristics of the object. In the context of object-oriented programming Booch offers us the following definition of an abstraction [Boo94]:

**Definition 2.3** *An abstraction expresses all the essential characteristics that make an object different from some other object; abstractions offer a precise definition of the object's conceptual borders from an outsider's point of view.*

In conclusion in the process of creating an abstraction our attention is focused solely towards the *exterior aspect* of the object and as such on the *object's behavior* while at the same time ignoring the implementation of this very behavior. In other words abstractions help us distinguish clearly between "*what* an object does" and "*how* the object does it".

An object's behavior is characterized through a sum of services or resources the object offers to some other fellow objects. Such a behavior in which an object (server) offers services for other objects (clients) is described in the so-called *client-server model*. The entirety of the services offered by a server object constitutes the object's *contract* or *responsibility* towards other objects. Responsibilities are fulfilled by means of certain *operations* (also called: methods

or member functions). Each object's operation is characterized by a *unique signature* composed from: a name, a list of formal parameters and a return type. The sum of an object's operations and their corresponding rules for calling constitute the *object's protocol.*

### 2.1.2 Encapsulation

Just as abstractions are used for identifying an object's protocol, encapsulation deals with selecting an *implementation* and treating it as a secret of that particular abstraction. The encapsulation process will be viewed therefore as the action of *hiding the implementation* from most client objects. In a more concise way we can define encapsulation as follows:

**Definition 2.4** *Encapsulation is the process of splitting the elements that form the structure and behavior of an abstraction into individual compartments; encapsulation is used for separating the "contractual" interface from its implementation.*

The definition above makes clear that an object has two distinct parts: the object's *interface* (protocol) and the *implementation* of this interface. Abstraction is the process that defines the object's interface and encapsulation defines the object's representation (structure) together with the interface implementation. The concealment of an object's structure and method implementation make up the so-called *information hiding* notion.

Encapsulation provides a set of advantages:

- By separating the object interface from the object's representation one can modify the representation without affecting the various clients in any way because these depend on the server object's interface and not its implementation.
- Encapsulation allows you to modify programs efficiently, with a limited and localized effort.

### 2.1.3 Modularity

The purpose of splitting a program into modules is to reduce the costs associated with redesign and verification issues by allowing you do this for every module independently [Par72] [BPP81]. The classes and objects obtained after the abstraction and encapsulation processes must be grouped and then deposited in a physical form called a module. Modules can be viewed as physical containers in which we declare the classes and objects that result after the logic level design. These modules form therefore the program's physical architecture. A program can be split into a number of modules that can be compiled separately but that are connected (coupled) among themselves. The languages that support the module concept also make the distinction between the module's interface and its implementation. We can say that encapsulation and modularization go hand in hand.

### 2.1.4 Inheritance

Abstractions are a good thing but in most non-trivial applications we will find a greater number of abstractions that we can simultaneously comprehend. Encapsulation helps us with this complexity by hiding the interior of our abstractions. Modularity helps by offering the means of grouping abstractions that are logically linked among themselves. All these, although useful, are not enough. A group of abstractions often forms a *hierarchy* and by identifying this hierarchy we can greatly simplify the problem understanding.

The most important class hierarchies in the object paradigm are: the class hierarchy (*"is a"* relationship) and the object hierarchy (*"part of"* relationship)

#### Class Hierarchy

Inheritance defines a relation among classes in which a class shares its structure and behavior with one or more other classes (we talk about simple and multiple inheritance). The existence of an inheritance relationship is the difference between object-oriented programming and object based programming.

From a semantic point of view inheritance indicates an *"is a"* relationship. For example a bear "is a" mammal so there is an inheritance relationship between the bear and mammal classes. Even as a programming issue this remains the best test for detecting the inheritance relationship between two classes A and B: A inherits B only if we can say that "A is a kind of B". If A " is not a" B, then A shouldn't inherit B. In conclusion inheritance implies a hierarchy of the generalization/specialization type in which the class that derives specializes the more generalized the structure and behavior of the class from which it was derived.

#### Object Hierarchy

Aggregation is a relationship between two objects in which one of the objects is part of the other object. From a semantic point of view, aggregation indicates a *"part of"* relationship. For example there is such a relation between a wheel and a car because we can say that "a wheel is part of a car"

### 2.1.5 Interfaces and Polymorphism

#### Interfaces

As we mentioned earlier, the sum of all function signatures for the functions that can be called by clients of that particular object class form the class's interface. Interfaces are fundamental in object-oriented systems. The objects are known inside the system only through their interfaces. There is no other way of finding out something about the object or asking it to do something except by using its interface. An object's interface says nothing about its implementation – therefore different objects can implement the same interface in different ways.

This means that two objects with identical interfaces can have wildly different implementations!

### Binding

When a certain operation is requested, the way in which the operation will be fulfilled depends not only on the operation itself but also on the object that will receive and execute the request by calling one of its member functions. This happens because there can be more than one objects that can respond to a particular request. In other words, the requested operation specifies the desired service and the concrete object represents the individual implementation of that service. The association between a requested operation and the object that will provide the concrete implementation of the operation through one of its member functions is called binding. Depending on the moment when this binding takes place, we differentiate between two types of binding:

- Static binding (early binding) - the association is created at compilation time. This binding is based on the types system known to the compiler through the various class declarations and the corresponding fixed (and therefore rigid) association of a class for each object.
- Dynamic binding (late binding) - the association isn't created when the program is compiled but rather it takes place when the program is running (at run-time)

### Polymorphism

In this manner, when binding dynamically, the request for an operation does not lead to the automatic correspondence between that operation and a certain implementation, the correspondence takes place only when the program is running. The main advantage of dynamic binding is the possibility of substituting objects that have identical interfaces at run-time. The option of using some object in another object's stead when both objects share the same interface is called *polymorphism*. Polymorphism is therefore one of the fundamental concepts of object-oriented programming.

## 2.2 Good Object-Oriented Design

In the previous section we introduced the key mechanism involved in object-oriented design. But, as in chess, knowing the chess pieces and the moves doesn't make you a good chess player. In this section we will therefore discus what a good design is and what makes the difference between a good and a bad design.

The quality of a design has an essential impact on the whole development process. Considering the lifecycle of a software system, the design phase is responsible for no more than 10 - 15% of the total effort; yet, up to 80% of

the costs are invested in the correction of erroneous design decisions that arise during this phase [BMP87]. So, what is good design? Coad defines good design as follows:

> *A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime.* [CY91b].

Thus, a good design is reflected by the minimization of costs, i.e. the costs of creating the design, transforming it into a proper implementation, testing, debugging and maintaining the system. Coad also emphasizes the fact that from the formerly mentioned cost categories, the most substantial one is related to maintenance, therefore he concludes: *"the most important characteristic of a good design is that it leads to an easily maintained implementation.*

More recently, Pfleeger also discusses the characteristics of a good software design in following terms:

> *High-quality designs should have characteristics that lead to quality products: ease of understanding, ease of implementation, ease of testing, ease of modification, and correct translation from requirements specification. Modifiability is especially important, since changes to requirements or changes needed for fault correction sometimes result in design change.* [Pfl98]

All these statements above drove us to the following couple of conclusions:

- It is hard to comprehend and quantify the "goodness" of a design by itself; therefore we have to apply the biblical principle: "*by their fruit you will recognize them*", i.e. we can get an understanding of the quality of the design only by regarding its "fruits": testing efforts, maintenance costs and the number of reusable fragments.

- We need criteria for evaluating a design not in order to build "perfect" software but to help us avoid badness. Therefore, good design is a matter of avoiding those characteristics that lead to bad consequences [CY91b].

It is impossible to establish an objective and general set rules that would lead automatically to high-quality design if they would be applied. But on the other hand heuristic knowledge reflects and preserves the experience and quality goals of the developers. They also help the beginners to evaluate and improve their design. Therefore, we are going to discuss next the most relevant criteria of good object-oriented design[1] and show the reflection of these criteria in terms of design rules and heuristics. According to those criteria, a good object-oriented design should have a manageable complexity, should provide a proper data abstraction and it should reduce coupling while increasing cohesion.

[1] We considered a criterion to be relevant if several authors identified it as such.

### 2.2.1 Low Coupling

In an object-oriented design, coupling is *"the interconnectedness between its pieces"* [CY91b]. Coupling is an important criterion when evaluating a design because it captures a very desirable characteristic: a change to one part of a system should have a minimal impact on other parts; also the understanding of one module should require the understanding of few other modules.

We are interested in coupling from the perspective of quality evaluation because an excessive coupling plays a negative role on many external quality attributes[2]. We identified the following impact of high coupling on quality attributes:

- *Reusability.* The *reusability* of classes and/or subsystems is low when the coupling between these is high, because a strong dependency of an entity (class, subsystem) on the context where it is used, i.e. the rest of the given system, makes the entity hard to reuse in a different context.
- *Modularity.* Normally a module (subsystem) should have a low coupling to the rest of the modules. A high coupling between the different parts (modules) of a system has a negative impact on the modularity of the system and it is a sign of a poor design, in which the responsibilities of each part are not clearly defined.
- *Understandability* and *Testability.* A low self-sufficiency of classes makes a system harder to understand. When the control-flow of a class depends on a large number of other classes, it is much harder to follow the logic of the class because the understanding of that class requires a recursive understanding of all the external pieces of functionality on which that class relies. It is therefore preferable to have classes that are coupled to a small number of other classes.

Coad and Yourdon identify two categories of coupling: coupling between objects by method invocations (*interaction coupling*) and coupling between classes through inheritance relations (*inheritance coupling*) [CY91b]. Budd proposes a more refined classification of coupling types [Bud91], ending up with a *ranked list* of coupling types (see Table 2.1) in which some coupling types are more acceptable or desirable than others. The five types of coupling identified by Budd are:

- *Internal data coupling* occurs when one class is allowed to modify the local data values (instance variables) in another module (class).
- *Global data coupling* occurs when two or more modules (classes) are bound together by their reliance on common global data structures.
- *Sequence (control) coupling* occurs when one module has to perform operations in a certain fixed order, but the order is controlled by another module.

[2]See also section 6.1 for a more detailed discussion on quality attributes

- *Parameter coupling* occurs when one module must invoke services and routines from another, and the only relationships are the number and type of parameters supplied and the type of the value returned.

- *Inheritance coupling* describes the relationship a class has with its parent class (or classes[3]).

| Type of coupling | Rank |
|---|---|
| Internal data coupling | Strongly undesirable |
| Global data coupling | Undesirable |
| Sequence coupling | Undesirable |
| Parameter coupling | Benign option |
| Inheritance coupling | Useful, but sometimes dangerous |

Table 2.1: A ranked list of the different coupling types [Bud91]

The different aspects of coupling are quantified using a large variety of metrics. In Section 2.4.1 we summarize the metrics on interaction coupling and in Section 2.4.2 the measure related to the inheritance lattice.

### Rules and Heuristics on Coupling

The principle of low coupling is reflected by all the authors that propose design rules and heuristics for object-oriented programming. Although having different forms or emphases they all converge in saying that coupling should be kept low. We will illustrate next, by citing three different design rules on coupling, each of these coming from a different author:

> **Law of Demeter**. *A method M of an object O should invoke only the methods of the following kinds of objects: itself; its parameters; any objects it creates/instantiates or its direct component objects* [LH89]

> *Keep the complexity of a message connection as low as possible. If a message connection involves more than three parameters, examine to see if it can be simplified* [CY91b] [JF88].

> *When we make a change we want to be able to jump to a single clear point in the system and make the change. When you can't do that you* [..] *either one class suffers many different changes* (divergent change) *or one change affects many classes* (Shotgun Surgery). *Arrange things so that, ideally, there is a one-to-one link between common changes and classes* [FBB+99].

[3] if we consider the case of multiple inheritance or the transitive closure of its parents.

We argued above about the necessity of having low-coupled systems. Yet, a tension exists between the aim of having low coupled systems and the fact that an amount of collaboration among objects (and thus coupling) is necessary in all non-trivial systems. Therefore, the goal in improving the quality of a system from this point of view is to reduce any unnecessary coupling.

### 2.2.2 High Cohesion

Cohesion is the complementary aspect of coupling. It describes the "degree to which the elements of a portion of design contribute to the carrying out of a single, well-defined purpose" [CY91b]. High cohesion at the class level signifies that all of the elements of the class are strongly related. The balance between low coupling and high cohesion is usually called *modularity* [Mey88] [Ern96]. The lack of cohesion affects essentially the quality of a system by reducing its:

- *Reusability.* Usually the lack of cohesion is due to the fact that more than one functionality is incorporated in a single class. Such classes are hard to reuse because of they are mixed with other functions that are usually of no interest for the context in which they are going to be reused.
- *Understandability.* Because classes with low cohesion do not focus on one and only one piece of functionality they are very hard to understand. Normally such classes include a large number of methods that do not all belong semantically together. Therefore, it is often hard to group the related methods in order to understand the services offered by the class.
- *Modularity.* If we consider cohesion at the module (subsystem) level, then a weak cohesion means that the system is not properly divided in subsystems, thus it has a lack of modularity.

#### Rules and Heuristics on Cohesion

Similar to coupling, there are many rules and guidelines that underline the need of high cohesion in methods, classes, and also inheritance relations. In this context, Coad and Yourdon state the following design rule on *methods cohesion:*

> *A method should carry one, and only one, function. A method that carries out multiple functions is undesirable. One clue is the size of the method: a method requiring 150 program statements, or one with deeply nested blocks should be looked upon with suspicion!* [CY91b].

Most of the design rules related to cohesion address are at the *class* level. Riel defines the following set of cohesion heuristics, which are also stated by several other authors [Lak96] [JF88]:

- *A class should capture one and only one abstraction.*
- *Keep related data and behavior in one place.*
- *Spin off non-related information into another class, i.e. non-communicating behavior.*

The third type of cohesion is the one related to the inheritance relations. A subclass is misplaced in a class hierarchy if semantically it is not a specialization of its base class. A sign of low inheritance cohesion is the selective use of the interface from the base class. Riel addresses this issue in several heuristics [Rie96]. Next we quote next a description of the *Refused Bequest* "bad-smell" that captures best the symptoms of a low-cohesive inheritance relation:

> **Refused Bequest**. *If a subclass inherits methods and data from its parents, but does not use them, then it "smells" bad. The smell of refused bequest is much stronger if the subclass is reusing behavior but does not want to support the interface of the superclass* [FBB+99].

Riel phrases the *Refused Bequest* flaw, in terms of a design heuristics, by the following statement "*Do not override a base class method with a NOP (NOP-eration) method*" [Rie96].

There are several metrics that quantify the principle of tight cohesion. Therefore, in Section 2.4.3 we present the definitions of the most important metrics on cohesion, with a special emphasis on metrics that measure class cohesion.

### 2.2.3 Manageable Complexity

Software systems tend to depart more and more from the principle of simplicity and become increasingly complex. The increase in size and complexity drastically affects several quality attributes, especially understandability and maintainability:

- *Understandability*. Classes that are huge and complex are hard to understand by humans especially if the class is also low cohesive, incorporating more than one functionality.
- *Maintainability*. Because of the difficulty of understanding the code, such classes are also hard to maintain, as any change to a part requires in principle the comprehension of the whole class.
- *Reliability* and *Testability*. A class that is too complex is not only hard to understand, but because of this it also hard to test, which in effect makes classes more error-prone and consequently reduces their reliability.

In Wirth's "plea for a lean software" [Wir95], the author states that *"reducing complexity and size must be the goal in every step in system specification, design, and in detailed programming"*. Design rules and heuristics that are related to the issue of size and complexity require the limitation of the size of modules (classes, methods, packages) and the simplification of the complexity of the control-flow. Guidelines for object-oriented design hint at the maximum number of attributes or methods in a class, at the depth of the inheritance hierarchy or at the size of a class interface. We selected three design rules to illustrate the guidelines on reduced complexity.

> *A class with 50-100 methods is probably a too complex abstraction and it should be split* [JF88]
>
> *Distribute system intelligence horizontally as uniformly as possibly, that is, top-level classes in a design should share the work uniformly.* [Rie96]
>
> *Reduce the number of arguments, as messages with 6 or more arguments are hard to read. Reduce the size of methods. Eliminate case analysis.* [JF88]

The reader may observe from the design rules above, that they are very easily quantifiable. As a consequence an important number of (object-oriented) metrics were defined in the literature to measure the aspect of complexity. We summarized them in Section 2.4.4. Reading through the definitions of these metrics offers an even deeper understanding of the different aspects of complexity.

### 2.2.4 Proper Data Abstraction

As we have seen in Section 2.1.1, data abstraction is an important characteristic of an object-oriented design. The most important quality attribute that is influenced by data abstraction is *understandability*, because system with a proper data abstraction display a good-level of modularity, which makes them easy comprehensible. The abstraction process is first applied after the analysis phase in order to distill in the initial *object-model*, – in which objects are representations of concrete entities from the problem domain – the abstract classes that model a more general concept.

The design rules related to choosing a proper data abstraction can be grouped mainly in two categories: the first category is related to the *complexity of the identified abstractions*; the second category deals with the *morphology of class hierarchies*. The design guidelines in the first category measure the degree of abstraction for a given class. Thus, a class that represents an improper abstraction may either contain too many or no reasonable abstraction. If a class is too complex, it is very probable that it captures more than one abstraction. Such a class is probably not only excessively complex, but also non-cohesive. Thus, we observe that in this point the cohesion, complexity and abstraction good-design criteria converge. In this context, Johnson and Foote emphasize that "*a class should represent a well-defined abstraction, not just a bundle of methods and variable definitions*" [JF88].

The other category of design guidelines is related to the shape and composition of class hierarchies. Several authors [Rie96], [Lak96], [JF88] emphasized that hierarchies should be deep and narrow, with the top of the hierarchy being abstract, and that subclasses being specializations of their base-classes, avoiding inheritance to achieve code reuse.

Note that all the previously cited design rules are mapping the issue of proper data abstraction to the structural aspects of the design. Therefore, design metrics can capture and quantify deviations from the abstraction criteria of good-design. As a consequence of the previous discussion on this criteria, the metrics that are adequate for this purpose are those related to the size and complexity of abstractions (see Section 2.4.4) and those that measure the inheritance hierarchy (see Section 2.4.2).

## 2.3 Software Measurement

We open this section with a set of general definitions on measurement, and measurement related concepts. The definitions are based on [FP97]:

**Definition 2.5 (Measurement)** Measurement *is defined as the process by which numbers or symbols are assigned to attributes of entities in the real world in such way as to describe them according to clearly defined rules.*

This definition of measurement requires some explanations and several further definitions. The concepts used in this definition such as *entity* or *attribute* will be defined next.

**Definition 2.6 (Entity)** *We define an* entity *as the subject of the measurement process. An entity might be an object, or a software specification or a phase of a project.*

**Definition 2.7 (Attribute)** *An attribute is a feature or property of the entity. For example, an attribute of a software specification is its length, and an attribute of a project phase may be its duration.*

Informally, the assignment of numbers and symbols must preserve any intuitive and empirical observations about the attributes and entities. In most situations an attribute may have different intuitive meanings to different people. Thus we have to define a model:

**Definition 2.8 (Model)** *A model is the expression of a viewpoint concerning the entity being measured.*

Once a model has been chosen it becomes also possible to determine relations between the attributes that describe the entity being measured. The need for defining good models is particularly relevant in software engineering measurement. For example, even a simple metric like the length of a program requires a well defined model of programs, which enables us to identify unique lines unambiguously. Similarly, the measurement of the effort spent on a phase of a project needs a clear "model" of the phase, which at least makes clear when the process begins and ends.

### 2.3.1 Classification of Measurements

We will classify the measurement activities using two different criteria. The first classification uses as criteria *the way the measurement is realized.* There are two types of measurement corresponding to this type of classification:

- *Direct Measurement* of an attribute is a measurement that does not depend on the measurement of any other attribute. In most of the cases the direct measurement is simpler.
- *Indirect Measurement* of an attribute is the measurement that involves the measurement of one or more other attributes. The indirect measurement is normally more sophisticated.

The second classification analyzes the problem from another viewpoint. The criteria used for this second classification is the *use of measurement.* Corresponding to this criteria there also two broad uses of measurement:

- *Assessment Measurement* of an attribute is a measurement that determines the current measure of an attribute, that is the value of the attribute at the present moment of time.
- *Predictive Measurement* of an attribute is a measurement that depends on a mathematical model relating A to some existing measures of $A_1$, ..., $A_n$. It gives a prediction of the future measure of $A$ based on the present measures of $A_1$, ..., $A_n$.

### 2.3.2 Representational Theory of Measurement

Although there is no universally agreed upon theory of measurement, most approaches are devoted to resolving the following three main issues [Fen94]:

- Which types of attributes can be measured? (representation problem)
- How to define measurement scales? (scales and scale types)
- What kind of statements about measurements are meaningful? (meaningfulness)

We will present a brief overview of the *representational theory* of measurement, pointing to the manner in which it addresses these three issues.

#### Empirical Relation System

Direct measurements of a particular attribute possessed by a set of entities must be preceded by the intuitive understanding of that attribute. This intuitive understanding leads to the identification of the empirical relations between the entities.

**Definition 2.9** *The set of entities $C$, together with the set of empirical relations $R$, is called an empirical relation system $(C, R)$ for the attribute in question.*

**Example**: For the attribute "height of people" we cam imagine empirical relations like "is tall", "taller than" and "much taller than".

### Representation Condition

The measurement of the attribute that is characterized by the empirical relation system $(C, R)$ requires a mapping $M$ into a numerical relation system $(N, P)$. Specifically, $M$ maps entities in $C$ to numbers (or symbols) in $N$, and empirical relations are mapped to numerical relations in $P$, in such way that all the empirical relations are preserved. This is the so-called representation condition, and the mapping $M$ is called representation. Formally we may express the representation $M$ as follows:

$$M : (C, R) \rightarrow (N, P)$$

**Definition 2.10** *The representation condition asserts that the correspondence between empirical and numerical relations goes two ways. Suppose for example the binary relation is mapped by $M$ to the numerical relation $<$. Then, formally, we have the following expression of the representation condition:*

$$\forall x, y \in C \text{ and } "\prec" \in R, \exists\ "<" \in P : x \prec y \Leftrightarrow M(x) < M(y)$$

**Example**: We will use again the previous example. Suppose $C$ is the set of all people and $R$ contains the relation "taller than". A measure $M$ of height would map $C$ into the set of real numbers $\mathcal{R}$ and "taller than" to the relation ">". The representation condition states that person $A$ is taller than person $B$ if and only if $M(A) > M(B)$.

### Scale. Representation Problem

Based on the aspects defined and discussed previously in this section we can give now the definition of the scale:

**Definition 2.11** *Being given an empirical relation system $E = (C, R)$, a numerical (formal) relation system $F = (N, P)$ and a measure $M : E \rightarrow F$, the triplet $(E, F, M)$ is called a scale.*

Having defined all these elements, we can look back at the issues that should be solved by a theory of measurement, and we can already answer the first question: which types of attributes can be measured? This question can be expressed formally as follows: being given an empirical relation system $E$ and a formal relation system $F$, the question that appears is: is there a representation so that $(E, F, M)$ is a scale? This problem is called the *representation problem*. If such a measure exists, then the attribute described by the empirical relation system $E$ is measurable. In other words, an attribute of a set of entities is measurable if and only if it is possible to find a representation that satisfies the representational condition.

If the representation problem is solved, then we can examine the next issue. Supposing that for an empirical relation system $E$ and a formal relation system $F$ we have found a measure, we can ask how unique this measure really is. For example, there are many different measurement representations for the normal

empirical relation system for the attribute of height of people. However, any two representations $M$ and $M'$ are related in a very specific way: there is always a proportionality relation. This problem is called the uniqueness problem. Therefore we define any admissible transformation as follows [FP97]:

**Definition 2.12** *The transformation from one valid representation into another is called an admissible transformation. Formally, this definition can be expressed as follows: let $(E, F, M)$ be a scale. A representation $M'$ is an admissible transformation, if $(E, F, M')$ is also a scale.*

### Hierarchy of Scale Types

There are different scale types and they can be ordered in a hierarchy that reflects the richness of knowledge concerning the empirical relation system [Fen94]. We normally start with a crude understanding of an attribute and a means of measuring it. Accumulating data and analyzing the results leads to the clarification and re-evaluation of the attribute. Consequently, the set of empirical relations is refined and enriched and the accuracy of measurement is improved. In increasing order of sophistication, the best known scale types are: *nominal*, *ordinal*, *interval*, *ratio* and *absolute* (see Table 2.2).

If we know the scale type, we can determine rigorously what kind of state-

| Scale Type | Admissible Transformation | Statistical Operation | Example |
|---|---|---|---|
| Nominal | 1-1 mapping from M to M' | Density of values | labelling, |
| Ordinal | Monotonic incr. function: $M(x) \geq M(y) \Rightarrow M'(x) \geq M'(y)$ | Median, " > " relation | air quality, preferences |
| Interval | $M' = aM + b(a > 0)$ | mean, standard deviation | relative time temper.(C,F) |
| Ratio | $M' = aM(a > 0)$ | geometrical mean | length, temper.(K) |
| Absolute | Identity $M' = M$ | all above | number of students |

Table 2.2: Scales of Measurement [FP97]

ments about measurements are meaningful, and which are not. A statement involving a measurement is *meaningful* if its truth or falsity remains unchanged under any admissible transformation of the measures involved. The notion of meaningfulness also enables us to determine what kind of operations we can perform on different measures. For example, it is meaningful to use means for computing the average of a set of data measured on a ratio scale, but not on an ordinal scale. Medians are meaningful for an ordinal scale but not for a nominal one (Table 2.2).

## 2.4 Object-Oriented Design Metrics

The goal of this section is to discuss the most important object-oriented metrics defined in the literature in the context of good object-oriented design. In order to offer a systematic approach, we selected four internal characteristics that are essential to object-orientation, – i.e. coupling, inheritance, cohesion and structural complexity – and classified the metrics based on these criteria. At the same time, for each characteristic, we discussed its impact on the external quality attributes (e.g. maintainability, testability, reusability).

### 2.4.1 Measures of Coupling

There are various ways of counting coupling. Next we briefly present these types of definitions, illustrating them by some relevant measures found in the literature.

#### Number of Collaborators

The declaration of an object of a remote class creates a *potential* collaboration between the two classes. This is measured by metrics like $Fan-Out$ [TS92], $DAC$ [LH93] or $CBO$ [CK94] ($Fan-Out \equiv CBO$). If two classes are collaborators, then a value of one is added the Fan-Out irrespective of how many messages flow between the two collaborators.

#### Number of Services

A second way of measuring coupling is this: when two classes collaborate, count the number of *unique* services accessed, i.e. the number of distinct remote methods invoked. One measure that counts the number of remote methods is $RFC$ [CK94], defined as the total number of methods that can be invoked from a given class: $RFC = NLM + NRM$ where $NLM$ represents the number of local methods and $NRM$ the number of remote methods. Thus, the $NRM$ term of $RFC$ is a coupling measure based on counting the remote services.

#### Number of Accesses

If one remote method is accessed from different parts of the client class, each access could be counted once. This is the approach taken by Li and Henry in defining the $MPC$ metric, which is the *"number of send statements defined in a class* [LH93] (also proposed in [LK94]). A similar type of definition is used by Rajaraman and Lyu in defining coupling at the method level. Their *method coupling* $MC$ measure [RL92] is defined as the number of non-local references in a method.

### 2.4.2 Measuring the Inheritance Lattice

The need to measure inheritance structure is emphasized by all authors. They suggest that the measurement should refer to the depth and the node density of

the hierarchy lattice. The measurement can be done both at the class and the system abstraction level. At the class level, Chidamber and Kemerer define the *Depth of Inheritance Tree* ($DIT$) [CK94] measure, also known in literature as *nesting level* [LK94] or *class-to-root depth* [TS92]. At the system level maximum and average $DIT$ can be defined [HS96].

At the class-level the *number of classes inheriting a specific operation* [TS92], the *number of overridden methods* (NORM) and *new added methods* [LK94] can also be defined. Related to these measures, Lorenz and Kidd [LK94] define the *Specialization Index* (SIX) metric as:

$$SIX = \frac{NORM \cdot DIT}{NOM}$$

where $NOM$ represents the total number of methods for the class. This measure is useful in differentiating between *implementation sub-classing* (low values for SIX) and *specialization sub-classing* (high values of SIX).

Chidamber and Kemerer [CK94] introduce the *number of children metric* (NOC), suggesting that classes high in the class hierarchy should have more subclasses than those lower down. Lorenz and Kidd [LK94] talk about an extension of this metric that counts not only the immediate subclasses, but also all the descendants for a given class ($NOD$), which is in our opinion more relevant than $NOC$.

Yeap and Henderson-Sellers discuss two measures designed to evaluate the potential reuse within class hierarchies, especially for those found or destined to be part of a class library. Both measures are defined at the system level.

The **Reuse Ratio** (U) is given by the number of super-classes divided by the total number of classes. A value near to 1 is characteristic for a linear hierarchy (poor design), and a value near to 0 indicates a shallow depth and a large number of leaf classes. The **Specialization Ratio** (S) is given as the number of subclasses divided by the number of super-classes. A value near to 1 indicates a poor design, while a high value indicates a good capture of the abstractions in the super-classes.

As Henderson-Sellers [HS96] correctly observes, further work on inheritance metrics is urgently required, to address at least the following issues: renaming and redefinition of methods, genericity and of course *polymorphism*, which no one has yet seriously addressed so far. In [Mar99] [Mar98] we took the first steps towards a rigorous approach of inheritance measurements. Another recent contribution on the measurement of polymorphism is found in [PL99].

### 2.4.3 Cohesion Metrics

The definition of object-oriented cohesion metrics had an interesting evolution: it started with an initial metric, called *Lack of Cohesion in Methods* (LCOM),

followed by several improvement proposed by different authors. We are covering them in the coming sections, following the evolution of these definitions.

### Lack of Cohesion in Methods – LCOM

The definition of this metric proposed by Chidamber and Kemerer [CK94] can be informally summarized as follows: $LCOM$ represents the difference between the number of pairs of methods in a class that use common instance variables and the number of pairs of methods that do not use any common variables.

Henderson-Sellers [HS96] notes that there are two major problems with $LCOM$: first, there are a large number of dissimilar examples, all of which give a 0 value for $LCOM$; second, there is no guideline for the interpretation of any particular value. This suggests that the requirements of $LCOM$ have to include the ability to give values across the full range of values, and the measure must give values that can be uniquely interpreted in terms of cohesion.

Consequently, he defines an improved version of $LCOM$, named $LCOM^*$, that considers the notion of perfect cohesion and then represents every particular case as a *percentage* of this perfect value.

### Tight and Loose Class Cohesion

Another critic of the $LCOM$ metric comes from Bieman and Kang in [BK95]. According to them, $LCOM$ is effective in identifying the most non-cohesive classes, but it is not effective in distinguishing between partially cohesive classes. In their paper, the authors propose two cohesion measures that are sensitive to small changes in order to evaluate the relationship between cohesion and reuse.

The definitions of the two measures are based on the access of instance variables by method pairs belonging to that class. Two methods are considered to be *directly connected* if both access at least one common instance variable of the class. Two methods are considered to be *indirectly connected* if they access a common instance variable through the invocation of other methods. Consequently, the two measures are defined as follows:

- *Tight Class Cohesion* ($TCC$) is the relative number of directly connected methods.

- *Loose Class Cohesion* ($LCC$) is the relative number of directly or indirectly connected methods.

In [Mar97a] we have evaluated the utility of these two ($TCC$ and $LCC$) metrics for detecting cohesion related design flaws, and found out that $TCC$ is extremely useful for this purpose, while $LCC$ proved not to be appropriate.

### 2.4.4 Measurement of Size and Structural Complexity

Applying size metrics at the *system level* we can get a good overview of the dimensions of the system, while using complexity metrics we are able to make a first assessment of the structural complexity of the given system. Applying this category of metrics at the *class level* we expect, on the one hand, to detect the classes that play an important role in the design, and on the other hand, to find the classes that are exceedingly large or complex, i.e. classes that tend to become "god-classes" (see Section 5.3.1).

#### Size Metrics

As object-oriented systems have various levels of abstraction (e.g. system level, class level, method level) size measures can be defined for each level. At the system level a commonly used metric is *number of classes in the system* [TS92] [LK94], which can be divided in number of *abstract* and *concrete* classes [TS92]. At the lowest abstraction level, *method size* [Lor93] can be indicative of the "object-orientedness" of a class, as classes with too large methods suggest a traditional conception. A similar indicative may be, at the class level, the *number of methods per class*. A very low value may suggest that some classes should be merged, while excessively high values indicate the need to decompose some of the classes. Li and Henry define two size metrics [LH93]: SIZE1 is the number of semicolons in a class and SIZE2 is the number of attributes plus the number of external methods for a class.

It seems that size and structural complexity measures are mainly based on two simple metrics: *Number Of Attributes* ($NOA$) and *Number of Methods* ($NOM$) per class. Lorenz and Kidd [LK94] differentiate between the number of *instance variable* ($NIV$) and the number of *class variables* ($NCV$), where $NOA = NIV + NCV$.

If counting only the attributes of complex types we come back to the definition of the $DAC$ metric [LH93] discussed earlier in the context of coupling measures. In counting $NOM$ we can differentiate on the one hand between external (public) methods ($NEM$) and internal (private) ones ($NHM$), and on the other hand between instance ($NIM$) and class methods ($NCM$), where $NOM = NEM + NHM = NIM + NCM$.

#### Structural Complexity Measures

Unlike simple data-like attributes, methods are not coherent in the distribution of their size and structural complexity. In order to measure the structural complexity for a class, instead of counting the number of methods, the complexities of all methods must be added together. This is the essence of the *Weighted*

*Method per Class* ($WMC$) metric of Chidamber and Kemerer [CK94]:

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

where $c_i$ is the static complexity of each of the $n$ methods of the class. If $c_i$ is considered unitary, we then are back to the $NOM$ size metric, i.e. $WMC \equiv NOM$. The authors of the metric note that "*complexity is deliberately not defined more specifically here in order to allow for the most general application of the metric*" [CK94]. This measure could be used in reverse engineering for detecting the central control classes in a system, based on the assumption that these classes are more complex than the others (model capture).

# Chapter 3

# Measurement and Design Quality

The quality assessment and improvement of object-oriented design is hard. Software metrics are a powerful means to evaluate and control the quality of design. However metrics also pose some problems of their own: it is hard to provide a relevant interpretation for software measurements.

Therefore, before presenting our approach, we will provide in this chapter the state-of-the-art of existing approaches related either with the qualitative evaluation of the design or with the relevant interpretation of object-oriented metrics.

## 3.1 Classification of Approaches

We have shown that one of the main difficulties resides in the gap that exists between quality as it is perceived and expressed from outside the software system, and the measurable characteristics of the system that can be assessed at the design and implementation level.

Therefore, we need approaches, strategies and techniques that would bridge this gap. Such an approach must have the following characteristics:

- **Relevant measurement interpretation**. The approach should increase the relevance of measurements interpretation, by providing precise metrics definitions and interpretation models that can be efficiently applied for the evaluation of design. But the interpretation of measurements in isolation is in most cases not enough. Therefore, the approach must provide mechanisms for *composing* the interpretation models of individual metrics.
- **Quantification of design principles and rules**. The approach must provide a methodology for transforming informal design rules and prin-

ciples into a quantifiable (measurable) expression. This would allow the engineer to inspect the design and validate it against these rules, at source code level.

- **Connection between external quality and design.** The approach must support a quality model in which external quality attributes are expressed in terms of the design characteristics, more precisely, in terms of design principles and rules that can be measured.

As a consequence of the aforementioned issues, we classified the existing work in this field and identified the following three categories:

1. *Measurements Interpretation.* Contributions that fall in this category are focused on providing proper means for defining metrics and interpreting measurement results. The contributions that fall in this category are mainly concerned with questions like: How to define metrics in an unambiguous manner? How to make measurements interpretation more usable and relevant? In Section 3.2 we are going to cover some of the main efforts related to the enhancement of the reliability and efficiency of metrics definitions and interpretation models.

2. *Design Inspection.* Moving on, we see this current work related to the approaches that apply metrics or complementary techniques for design inspection. Here we identify here two subclasses of contributions: one is dealing with the recovery of design information or design decisions from code (Section 3.3) while the main concern of the other category of contributions is to find ways to identify and localize design problems (Section 3.4). Thus, the approaches related to design inspection are looking for answers to the following question: How to apply metrics in order to *understand* and *evaluate* the design of object-oriented systems?

3. *Quality Models.* Finally, research in this direction is concerned with providing a proper quality model, as a means to bridge the gap between external quality attributes and metrics. In Chapter 6 we propose a new type of quality model that starts exactly from the drawbacks of the traditional approach (i.e. the Factor-Criteria-Metric paradigm). Therefore, in order to increase the readability of this work, we postpone the entire discussion on existing approaches concerning quality models till Section 6.1.

## 3.2 Definition and Interpretation of Metrics

In this section, we are going to discuss the main contributions related to the way metrics are defined and interpreted. We first describe the two frameworks for metrics definition proposed by Briand et.al. and afterwards, we discuss about paradigms for the interpretation of metrics, as these are strongly related with our work.

### 3.2.1 The Strive for Rigorous Metric Definitions

As already mentioned in the first chapter, through the last decades, software metrics have been brought to the attention of the software engineering community, as they provide a quantitative means to control the quality of software. As a consequence, an impressive number of metrics have been defined in the past, to the point where we can really speak about an inflation of such metric definitions. But there are a lot of problems with these definitions. This fact has been emphasized in the past by numerous authors. We summarized the main critiques as follows:

- Many times in the literature we find multiple metrics measuring the same attribute in different manners and offering contradictory results. This is mainly because the measures do not properly capture the attribute to be measured [Fen94] [FP97].
- Often, measures are not accepted in the industry because the concepts they work with are not precisely defined [BSB96].
- Empirically defined measures are "highly parochial, highly limited and highly unscientific" [HS96]. Thus, their use is restricted to the strict context in which they are defined and their proper use in other contexts is extremely reduced.

Because of all these reasons, in recent years several works addressed the issue of metrics definitions. Kitchenham, Pfleeger and Fenton proposed a framework for measurement validation [KPF95] based on identifying the elements of measurement and their properties (structure model), identifying how these elements are defined when constructing a measure (definition models), and defining appropriate theoretical and empirical methods for validating the properties and definition models. Yet, we have only found one reference in the literature, where the framework was applied [HCN98] for the validation of a metrics suite.

Briand et al. noticed that the definition of metrics in general, and object-oriented metrics in particular, lacks a standard terminology and formalism. Consequently, the potential uses of many existing metrics become unclear, and the relation between complementary measures that should be used together are hardly specifiable. Therefore, the authors propose two comprehensive frameworks, one for measuring coupling [BDW99] and the other for measuring cohesion in object-oriented systems [LB98]. The main contribution of that work is that it offers a standardized terminology and formalism that ensures the consistency of all metrics definitions and provides a means to classify metrics belonging to the same category.

Briand et al. affirm that the frameworks are intended to "*support the definition of new measures and the selection of existing ones based on a particular goal of measurement*" [BDW99]. Yet, although a proper formalism is an indispensable basis for this intention, the process of properly selecting and correlating metrics

for a particular goal needs mechanisms situated at a higher level of abstraction than those provided by such a framework. In conclusion, a rigorous framework for metrics definitions is absolutely necessary in order to build higher-level approaches that would eventually serve a given measurement goal, as we will prove in the following chapter (see relation between Section 4.2 and Section 4.3).

### 3.2.2 Goal-Question-Metric Paradigm

As we noted in the previous statement, any measurement activity must have clear objectives [Fen94]. Measurements are often not goal-oriented and therefore, the collected data roves to be of no use in the end. At the beginning of each experiment, it must be clear if the goal of the measurement is assessment or prediction of attributes, which are the entities involved in the experiment and which are the significant attributes to be measured. In this context, the Goal-Question-Metric (GQM) model [BR88] spells out the necessary obligations for setting objectives before embarking on any software measurement activity.

The GQM approach provides a framework involving three steps (Figure 3.1):

1. List the major *Goals* of measurement.
2. From each goal derive the *Questions* that must be answered to determine if the goals are met.
3. Decide what *Metrics* must be collected in order to answer the questions.

The goal indicates the purpose for collecting the data. The questions tell us how to use the data and they help in generating only those measures that are related to the goal. In many cases, several measurements are needed to answer a single question; likewise, a single measurement may apply to more than one question.

What is missing from the GQM approach is the *model* based on which measures are combined to provide an answer to the question. Therefore, the initial GQM paradigm was improved by Shepperd [She90] by an additional component that connects GQM with models and theories, as depicted in Figure 3.1.

The Goal-Question-Metric paradigm is only an abstract approach that needs to be instantiated for different investigation goals. As we will see in the next chapter, a methodology for defining higher-level interpretation rules (Section 4.4) must be based on the improved version of the Goal-Question-Metric approach.

## 3.3 Design Recovery

Design recovery involves the examination of a legacy code in order to reconstruct design decisions taken by the original implementers. Some approaches are based on analyzing both the source code and the executable images, but as we base

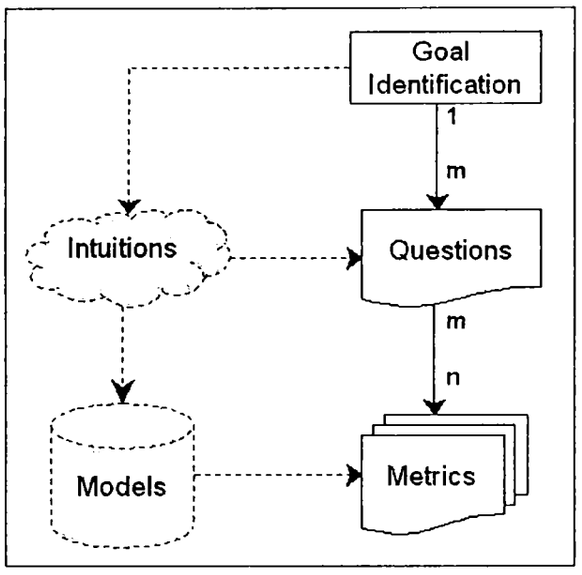



Figure 3.1: The Goal-Question-Metrics Paradigm [BR88]. Dotted elements represent Shepperd's improvement [She90] of the initial approach.

our approach on static analysis, we will consider next only those approaches that capture the design by analyzing the structure of the code.

### 3.3.1 Finding Refactorings via Metrics

A recent paper [DDN00] describes a suite of metrics-based heuristics that can be used for the detection of refactorings operated through successive versions of a software system. For each heuristic, the authors identify a set of adequate metrics, based on the previously discussed GQM approach. These metrics are then combined in a quantifiable rule, based on a step-wise defined "receipt" – as named by the authors. Formally, the "receipt" is expressed in terms of metrics variations between the successive versions of the analyzed system. The metrics used for this purpose are all simple size measures referring to methods, classes and inheritance lattices. Some of the refactorings that the authors claim to detect are: *Split into Superclass*, *Merge with Subclass*, *Move to Subclass*, *Split Method*.

In relation with the goal statement of this dissertation, the main contribution of this work is that it describes a detection technique, which uses object-oriented metrics to identify design fragments that have particular properties. In this context, the idea to express the detection rule as a *"quantifiable heuristic"* by combining metrics (more precisely, metric variations) is especially valuable. Moreover, the heuristics proposed in that paper find those parts of a system that are often refactored; therefore, these parts are supposed to have an un-

stable design. It is probable that such design fragments present some sort of design-flaws. This aspect relates it even more with our "quest" for quality at the level of design.

Yet this approach has also several disadvantages. The fact that the technique is based on several versions is a major limitation, as for most of the systems this precondition cannot be fulfilled. Another drawback, noted even by the authors, is its lack of scalability with respect to the number of changes. Thus, if a number of subsequent transformations were applied on the same code fragment the technique becomes very imprecise. A third limitation is its high dependency on the assumption of name preservation. Consequently, if design entities (e.g. a class) are renamed in later versions, the approach becomes unusable for those entities.

### 3.3.2 The Class Blueprint

Lanza and Ducasse present in [LD01] a new approach on code and design understanding, and more precisely on the understanding of classes based on metrics combined with a new visualization technique, which they call the "class blueprint". The authors decompose the elements of a class into five layers (e.g. the initialization layer, containing the methods used for constructing objects; the interface layer consisting of the methods that can be invoked from outside the class)

In a "class blueprint" the elements of a class are graphically represented as boxes whereby their shape, size and color reflect semantical information. The two dimensions of the box are given by two metrics, – e.g. for attributes, the two dimensions are: the number of accesses from within the class, and respectively the number of accesses from outside the class. The color is used to represent supplementary information, e.g. the layer to which the entity belongs, or the fact that a method is overridden.

Based on the "class blueprint" the two authors propose a categorization of the classes, from two perspectives: first they look at each class in isolation, and then they enhance the categorization by adding the inheritance perspective. Another especially interesting aspect is a technique that allows the identification of three types of *suspicious* class blueprints.

The main contribution of the above approach is that it helps the engineer visually detect in an easy manner the classes with special "property-patterns", during the development or reengineering of a system. The fact that the shape of the boxes representing classes and methods are given by metric values brings it close to the desirable approach described in the beginning of this chapter.

But this approach has also two important limitations: first, the filtering mechanism is missing, i.e. in this approach the engineer must visually detect the

class-patterns from the whole image, which makes the approach hard to use for larger systems. The second limitation is referring to the mechanism by which metrics can be combined in a box: due to the visual representation, the approach limits the number of metrics that can be used to two, which is insufficient for the representation of complex rules. In addition to that, the way of combining metrics in order to express metrics-based rules is limited to a direct and simple combination given by the relation that exists between the height and the width of the box.

## 3.4 Identification of Design Problems

In Section 2.2 we discussed a set of criteria for good design. According to those criteria, a good object-oriented design should have a manageable complexity, should provide a proper data abstraction, and it should reduce coupling while increasing cohesion. Why do we need these criteria? On one hand, we need them because they help us define what good design is; but on the other hand they can also serve a more pragmatic goal: they help us design systems that avoid badness. We are not trying to build *perfect* software systems, but software that can overcome the challenges over its lifetime [CY91a]. Thus, good design is more a matter of avoiding those characteristics that lead to bad consequences.

Therefore, in this section we are covering the existing contributions related to the identification and localization of design weaknesses that lead to systems that are "rotting"[Mar00].

### 3.4.1 Design Principles and Heuristics

Although it is impossible to define a general set of objective rules in order to evaluate the quality of a design, heuristic knowledge reflects and preserves the experience and quality goals of the developers. They also help beginners evaluate and improve their design. Design rules contain criteria for the identification of critical design fragments and they also suggest the code transformation that should be performed (e.g. "If a class has too many methods, split it!").

In the context of this work the terms: *design principle*, *design rule* and *design heuristic* should be understood as defined in [Ern96]:

**Definition 3.1 (Design Heuristic)** *A statement or an information, distilled from the experience of working with a software development methodology, whose application improves the quality of the design is called a design heuristic.*

**Definition 3.2 (Design Rule)** *A design heuristic that states an imperative or an interdiction concerning the characteristics of a design is called design rule.*

**Definition 3.3 (Design Principle)** *A design heuristic that expresses an abstract criterion for the evaluation of a design is called a design principle.*

The impact on quality of applying design principles in object-oriented systems was repeatedly proven by experiments [BBD01], [BBDD97], [AM96]. In a recent experiment conducted by Briand et.al. [BBD01], the authors analyzed the impact of applying design principles such as those defined by Coad and Yourdon [CY91a] [CY91b] on the maintainability of object-oriented designs.

A literature survey showed a constant and important preoccupation for this issue: beginning with the design guidelines initially proposed by Johnson and Foote [JF88] in 1988, enriched by Riel [Rie96] and Lakos [Lak96]. Most of these rules, guidelines and heuristics are synthesized in [Bär98].

While in the past, several authors have proposed criteria for a good object-oriented design [CY91b] [Pfl98] [Mey88], others were concerned with identifying and formulating design principles [Mey88] [Mar00] [Lis88], rules [CY91b] [Mey91], and heuristics [Rie96] [JF88] [Lak96] [LH89] that would help developers fulfill those criteria while designing their systems.

An alternative approach to disseminate heuristical knowledge about the quality of the design is to identify and describe the symptoms of bad-design. This approach is used by Fowler in his book on refactorings [FBB+99] and by the "anti-patterns" community [BMB+98] as they try to identify situations when the design must be structurally improved. Fowler describes around twenty code-smells – or "bad smells" as the author calls them – that address symptoms of bad design, often encountered in real software systems.

All the approaches mentioned in this section have two important drawbacks:

1. The first major limitation of the design rules and guidelines is their *heterogeneous level of abstraction*. The rules, even when proposed by the same author (e.g. Riel [Rie96]), may vary from very concrete and "sharp" rules (e.g. "a class should not contain more than six objects" or "avoid multiple inheritance") to abstract and general guidelines (e.g. "Avoid centralized control" or "Minimize the coupling between classes"). Because of this heterogeneity this heuristic knowledge is hardly applicable in a coherent and systematic manner. In practice, we noticed that concrete rules tend to be fervently applied, while the general guidelines tend to be ignored. But usually the heuristics with a higher impact on the quality of design are unfortunately those that are more abstract, as they touch the essence of good-design in a manner that is more widely applicable. These remarks bring us to the second drawback.

2. One of the main reasons why sharp and concrete rules are more appealing to an engineer is the fact they are easy to quantify. Most of the software design metrics are based or inspired by concrete design rules. Unfortunately, most of the heuristical knowledge is *hard to quantify* in terms of a single metric. Therefore, the quantification of most design heuristics is still very vague.

Although we partially agree with Fowler stating that "no set of metrics rivals informed human intuition" [FBB+99], the latter has a big disadvantage: it does not *scale up* with the dimensions of today's software. Therefore, in order to find and improve design fragments that violate the principles of good design we need mechanisms and methodologies that support the quantification of this heuristical knowledge.

### 3.4.2 Quantification of Design Principles

In a suite of articles [Mar96b], [Mar97c], [Mar97b], [Mar96c] summarized in [Mar00], Martin synthesizes a number of object-oriented design principles. The principles are organized in two levels of abstraction: principles for the class design and principles for package architecture. Beyond the main contribution of putting together the major principles of object-oriented design, Martin's papers bring another contribution which is relevant to our work: the author proposes a set of metrics for quantifying a couple of design principles [Mar97c].

We will illustrate Martin's approach by one of these quantified principles, i.e. *Stability Dependency Principle* (SDP). This principle, related to the stability of packages, is first formulated in abstract terms, as follows: "*Depend in the direction of stability*". In this form, the principle is hard to apply. Yet, the author "distills" the principle and identifies two direct measures: one measuring the efferent coupling ($C_e$) of the package and another measuring the afferent coupling ($C_a$). Based on these two metrics, Martin defines the *instability factor* $I$ as:

$$I = \frac{C_e}{C_e + C_a}$$

and rephrases the principle as follows: "*Depend upon packages whose I metric is lower than yours*". In other words he succeeded in expressing the SDP principle in terms of metrics. This highly increases the usability of this principle, and also allows to automatically check whether a given package structure is "SDP compliant".

Unfortunately, Martin's approach is limited to the quantification of two principles, both defined at the package level. Thus, the author does neither provide nor suggest a systematic methodology for quantifying further design principles or rules.

### 3.4.3 Identification of Structural Problems

In [Ciu99], [Ciu01] we find an alternative approach to problem detection based on violations of design rules and guidelines. The rules are specified in terms of queries, usually implemented as Prolog clauses. This approach focuses on structural properties that can be fully automatically detected, pointing directly to the "critical design fragments". These properties are specified as "sharp" rules, the result of applying such a query being a bipartite decomposition of the entities of the system, i.e. those that violate the rule and the others. Compared to

this method, the metrics-based approach is focused more on "fuzzy" guidelines – e.g. "a class should not be strongly coupled to other classes". This requires a *multi-partite decomposition* of the entities (e.g. "very strong coupled", "normally coupled", "loosely coupled").

Thus, this approach is supposed to detect "dark" parts of the system in a highly-automatized manner, but it can hardly detect the "gray" areas of a project, which are more often encountered in software systems. Therefore, an approach is needed that can also detect the "gray" parts of a design. This type of detection is more difficult because the measurement results always need an interpretation to bring us to the critical design fragments.

Concerning its relation to design measurements, although Ciupke takes advantage of some simple metrics for identifying several design problems, his approach is not based on metrics. In fact, the author delimits his approach from the metrics-based ones, as he doubts both the ability of metrics to serve for the identification of design problems, and the ability of such approaches to be extensible with respect to the set of detectable problems. In the following chapter, we will prove both hypotheses as wrong.

Another limitation of this approach is its omission to explicitly relate the detection of design problems to any higher-level quality goal. Because of this, the approach can't help us understand the *relevance* of the detected problems in respect to the investigation goal. In conclusion, the approach is an important contribution towards the improvement of design structure in object-oriented systems, but because of the before-mentioned reasons it is unable to reach the goal that we stated in the beginning.

### 3.4.4 Frameworks Consolidation Based on Metrics

In her dissertation Karin Erni [Ern96] [EL96] proposes a metrics-based approach for improving the consolidation phase of framework development, by speeding-up the detection of "hot spots" in the framework. Erni noticed that in order to detect design problems, metrics must be used in correlation, and they must be used in conjunction with a quality model. Thus, the author introduces the *multi-metric* concept, as an n-tuple of metrics expressing a quality criterion (e.g. modularity). Thus, Erni suggests that a multi-metric should be regarded as a *quality profile*. In other words, a quality criterion in Erni's approach becomes associated with a multi-metric instead of being associated with a number of metrics [MRW77]. The metrics that are part of a multi-metric may be weighted by the user depending on its their degree of relevance to the quality criteria that they are associated with. Based on the inheritance or dependency contexts, a multi-metric may even characterize even a cluster of related classes.

The main contribution of Erni's work is that it provides one of the first metrics-based approaches for the identification of design problems. In this context, her

emphasis on defining and using a higher-level measurement interpretation mechanism, and her quality-driven approach, makes it a valuable contribution to the state-of-the-art.

But is a *multi-metric* a proper mechanism to bridge the gap between an external quality goal (e.g. reusability) and the quality measured at the level of the design structure? After a detailed analysis of the approach, we noticed several drawbacks that severely limit its further usability:

- *Multi-metrics lack abstraction and encapsulation.* As a higher-level "quality profile", we expected multi-metrics to hide the multitude of component-metrics from the engineers, so that they can "forget" about numbers and reason in the more abstract terms of principles and rules of good-design. But using Erni's approach the engineer must still interpret and reason in terms of numbers. The reader may argue that multi-metrics are the expression of a quality criteria. But this does not add considerably more abstraction than the one provided by the association between a quality criteria and a set of metrics, in a Factor-Criteria-Metric quality model [MRW77]; and therefore, – as we will explain in detail in Section 6.1.3 – this is far from being satisfactory.

- *Multi-metrics lack specificity.* As a (partial) consequence of the previous remark, multi-metrics are unable to describe a higher-level design problem in terms of metrics; multi-metrics allow us to build "metric-clusters", but they do not support the definition of flexible metrics-based design rules. Because of this, it is very hard to find improvement solutions, as the results of a multi-metric are not explicitly associated with a higher-level design problem. Thus, with multi-metrics we are still in the sphere of symptoms, rather than moving to the sphere of real design problems.

- *Trend-analysis is not applicable for single-version systems.* Erni uses trend-analysis techniques to identify the suspects from the results of a multi-set. This is an interesting and useful approach for the cases where multiple versions of the analyzed systems are available. But in most of the cases, especially for legacy systems, the only available artifact is the source-code of a single version of the system. Thus this part of Erni's approach is strictly limited to multi-version systems.

## 3.5 Refined Statement of Goals

This chapter has analyzed the different approaches and techniques related to the assessment of design quality in object-oriented systems using software metrics. We reached now the point where we draw a line, and based on the limitations and drawbacks encountered in the analyzed approaches, we refine the goals of this dissertation. For a systematic approach, we will follow, in a step-wise fashion, the sequence of criteria presented in the beginning of this chapter (Section 3.1)

for the characterization of an ideal approach and the classification of existing approaches:

- **Define composable measurement interpretation models.** In other words, the first precise goal would be to provide a mechanism that supports the composition of interpretation models for metrics in higher-level expressions that would provide more meaningful interpretation results.

- **Define a methodology, usable in practice, for quantifying informal rules related to the design.** Such a methodology must depend only on the source code of the analyzed system, and it must be *scalable*, i.e. it must be especially applicable on large-scale systems. This way, for the first time, we will be able to measure what is important for the design and not only those isolated aspects currently captured by metrics. This would increase the relevance of measurement for the assessment of design.

- **Define a suite of quantified expressions of design rules and heuristics** that can identify can identify poorly-designed fragments. The suite must be usable to detect real *design flaws* and not only *symptoms* of poor design. These quantified rules must be defined so that they are able to capture relevant design problems at different levels of abstraction, from the method level up to the issues related to subsystem-design.

- **Define a new quality model** that allows external quality attributes to be expressed in terms of the quantified expression of good design knowledge, i.e. let the quality attributes "communicate" with design principles and rules, not with large sets of measurement results that are hard interpret. Obviously, if this goal is reached, the new quality model will provide the link between external quality attributes and the metrics-based rules that evaluate the conformance to the principles of good object-oriented design.

# Chapter 4

# Detection Strategies

In the beginning we stated that the goal of this work is to increase the relevance of measurement interpretation in order to support the assessment and improvement of quality in object-oriented systems. In the previous chapter we identified a set of desirable characteristics for an approach that would address this goal. After analyzing several approaches, the following conclusion was reached: in spite of the increasing number of efforts in this field, there is still an uncovered gap between the things that we measure at the design and implementation level, and the things that are really important in terms of quality characteristics. This gap is caused mainly by the fact that the abstraction level of all mechanisms currently used for interpreting metrics results is too low to capture the design aspects that are relevant for the assessment and especially for the improvement of design quality.

Through the next three chapters, the current one included, we propose a complete approach for closing this gap between qualitative and quantitative statements concerning object-oriented design. The approach is based on higher-level, goal-driven methods for measurement interpretation.

In the given context, this chapter introduces a new concept and mechanism, called *Detection Strategy*, which supports the definition of higher-level interpretation rules, in order to increase the relevance of measurement results. Any interpretation is deemed relevant only if it provides the engineer with information that is useful in the context of the investigation goal.

The chapter starts with a description of the phases of the measurement process (Section 4.1) used throughout this work. In any discussion regarding the measurement process two questions impose themselves as particularly relevant:

- What is the object of our measurements?
- How do we interpret the results in a relevant manner?

Therefore, in the following part of the chapter (Section 4.2), we define the boundaries of our measurement activities in terms of a meta-model for object-oriented systems.

The last part of the chapter (Section 4.3) deals with the problem of relevant measurement interpretation. In this context we introduce the concept of *Detection Strategy* as the proper mechanism for bridging the gap between higher-level investigation goals and measurements, by defining quantified rules that are capable of expressing an investigation goal. We will both present the anatomy of a detection strategy and also provide a methodology for defining instances of this mechanism.

## 4.1 A Goal-Driven Measurement Process

When assessing the quality of object-oriented design by using metrics a number of two aspects can be identified: a precise *investigation goal* and a precise approach, i.e. *a metrics-based one*. On the other hand, the only available (and reliable) input is the *source-code*. In this context we need a *goal-driven measurement process* to describe the phases of going from the source code to the results expected for the investigation goal that drives the measurements. The phases of such a process are described in Figure 4.1.

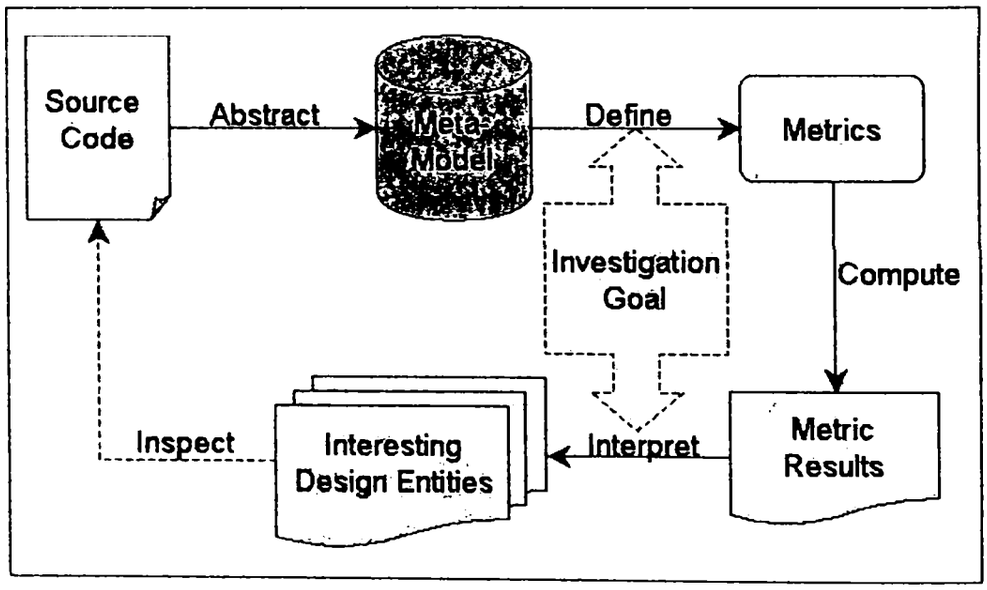



Figure 4.1: The phases of a goal-driven measurement process

### 4.1.1 Investigation Goals

Before describing the phases of the measurement process we want to take a closer look to the issue of *investigation goals*. The investigation goal is the *driving force* for applying metrics, the reason why the source code is analyzed using metrics. Generally speaking, metrics can serve a wide range of investigation goals, from cost and effort estimation to evaluation and comparison of tools and methods. But focusing on product metrics and especially on those that quantify the *structure of object-oriented programs* at least the following investigation goals can be identified:

- *Code Understanding.* In order to understand and to manipulate an object-oriented legacy system, it is necessary to capture its design, its architecture and the relationships between the different elements of its implementation. A common problem with many systems is the lack of proper documentation. Consequently, we need reverse-engineering techniques to extract design information from the code. [BBC+99a].

- *Identification of Reusable Components.* In the last decade important efforts were invested in extending the scope of reuse beyond the traditional reutilization of different libraries, up to the identification and extraction of components found in legacy systems [CB88] [Tri01], or even to extract frameworks [BBC+99a]. Thus, a possible goal for investigating the structure of a system would be to find these reusable components.

- *Detection of Design Weaknesses.* Probably the most frequent goal, and also the one that is closest to the interest of this thesis, is the use of metrics to detect flaws that occur at design level.

### 4.1.2 Phases of the Process

The first phase of the measurement process deals with the transformation of source code into a more abstract representation form, in which only those elements are kept that are relevant in respect to the measurement activities to be performed. The first phase of the process is thus an *abstraction phase.* The resulting representation is usually called *meta-model* or *semantic graph.* A meta-model has the main advantage of being easier to manipulate and understand than the code itself. All the metrics are defined as queries on the meta-model.

The second phase of the process is the definition of metrics. Although uncountable number of metrics has been defined in the literature, most of them have confusing definitions [She90] [Zus92]. The main cause for this is the lack of a rigorous definition of the measured entities [KPF95], i.e. of a precise meta-model. In our view, the definition of metrics is strongly related to an investigation goal. Metrics should be defined out of the necessity to contribute to the quantification of a specific investigation goal.

The third phase of measurement requires the *computation* of metrics. Metrics computations are defined as sequences of queries on the meta-model. There are different concrete implementations imaginable, depending on the implementation of the model, but in all cases the meta-model can be seen as a repository of design information, while metrics are queries defined on this repository. As a result of computing a metric, a number is associated to each program element of a certain type, measured by that metric.

The last phase, i.e. the *interpretation* of measurement results, is probably the most critical step as it cannot be fully automatized, like the previous phase which involved the computation of metrics. This interpretation is based on an *interpretation model* for that particular metric. During the interpretation phase, the initial data set of measurement results is filtered, keeping only those program elements (and their correspondent metric value) that are supposed to be interesting for the engineer, in conformity with the interpretation model, whereby the interpretation models are dictated by the investigation goals. As depicted in Figure 4.1 this phase is decisively influenced by the investigation goal, as the entire interpretation is driven by it. This explains why we often find in the literature definitions of metrics together with interpretation models that are irrelevant for our context. The lack of relevance is due to the divergence of the investigation goals. The interpretation process cannot be completely automatized, and therefore the interpretation of results must be manually validated in the initial source-code (see Fig 4.1).

## 4.2 A Meta-Model for Object-Oriented Systems

In informal terms, a meta-model is an attempt to describe the world around us for a particular purpose [Pid02]. Two aspects of the previous definition are essential: "an attempt to describe the world" and "for a particular purpose". In the context of this work, the "world" is the structure of object-oriented programs, while the "particular purpose" is identified with our goal to analyze the quality of software by static analysis techniques [JR00] applied on the structure of object-oriented systems.

The structure of an object-oriented system consists of a set of *design entities* (e.g. classes, methods) and the relationships existing between them (e.g. inheritance, containment). On the other hand, our meta-model disregards dynamic elements (e.g. objects), as they cannot be determined before runtime. In addition, many important systems cannot be run independently, and so, there's no access to dynamic information.

Based on the considerations above we provide the following definition of a meta-model:

**Definition 4.1 (Meta-Model)** *A meta-model for an object-oriented system is a precise definition of the design entities and their types of interactions, used for defining and applying static analysis techniques.*

Thus, a good meta-model should therefore capture only those types of entities that are relevant for the analysis, together with the properties of those entities and the relationships that exist between them. In a measurement context, the meta-model defines the boundaries of our measurement activities. In our view a meta-model is composed of the following elements:

- *Design Entities* (e.g. classes, packages)
- *Properties* of design entities (e.g. the visibility level of attributes)
- *Relations* between the entities (e.g. methods access attributes)

By analyzing the different design entities that appear in object-oriented systems, we reached the conclusion that they belong to different categories and consequently have different compositions. The constructs and rules used to describe the meta-model are therefore distilled in the form of a *meta-meta-model.* The meta-meta-model is depicted in Figure 4.2[1].

The design entities that appear in our meta-model fall into two categories:

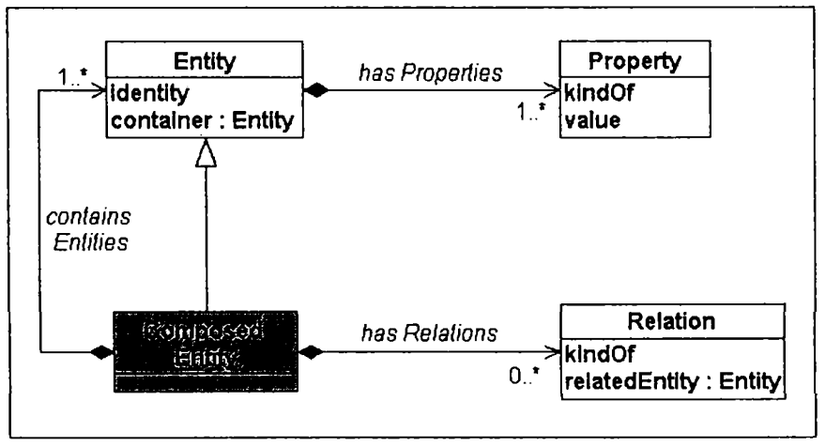



Figure 4.2: The meta-meta-model capturing the constructs and rules of the meta-model

*primitive* and *composed* entities. Primitive entities have an identity (i.e. a name that identifies them, like "Class", "Package" etc.) and a set of properties. In addition to these, composed entities have a list of contained entities, which are primitive or composed entities.

[1]Relating it to similar efforts, our meta-meta-model can seen as a simplified view of the models defined in [KPF95]

In order to facilitate the usage of the meta-model, we decided to express it in terms of sets and relations. Each entity type will be represented by a set that contains all the entities of that type. The entities that make up our meta-model are depicted in Figure 4.3.

The meta-model is composed of 7 + 1 entities: three of these are composed

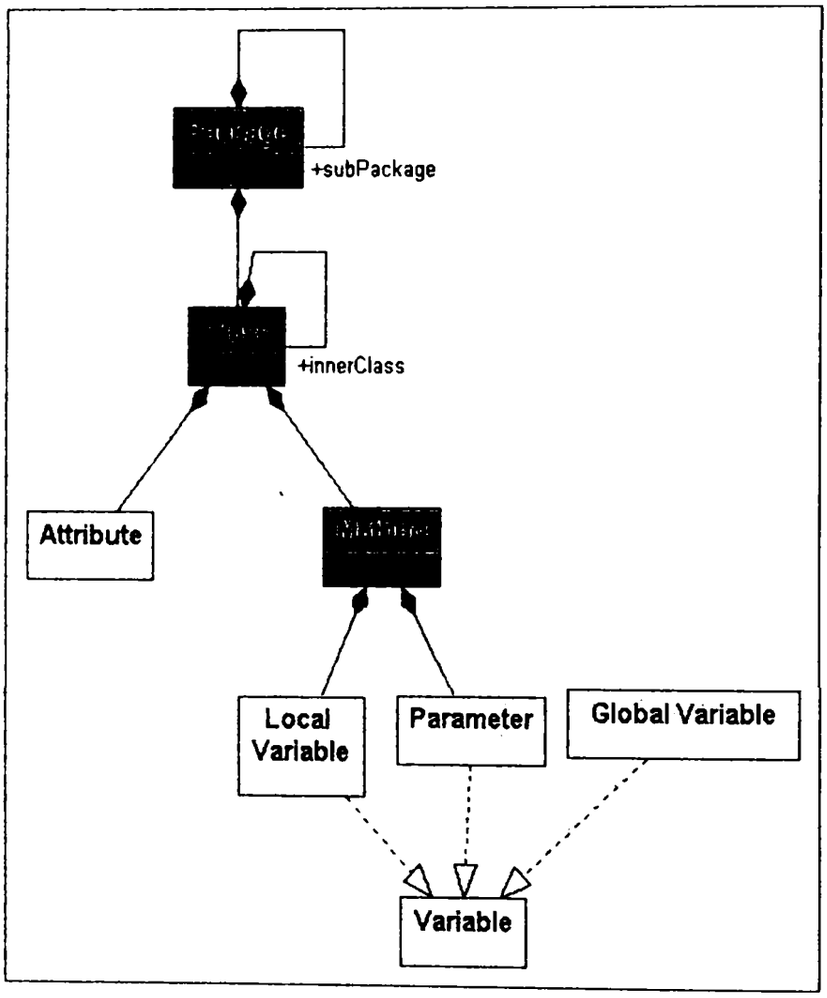



Figure 4.3: The entity types that make up our meta-model together with the containment relations that exist between them. The dark-colored entities represent the composed entity types, while the light-colored ones depict the primitive entity types.

(Package, Class and Method), while the other four are primitive (Attribute, Parameter, Local Variable and Global Variable). The eighth one (Variable) is an abstract entity that was introduced in order to factorize the common properties of all variables.

In the context of metrics-based analysis techniques, the aforementioned classification of design entities has a particular relevance: it provides a pertinent explanation about why metrics are defined and computed only for some entity types (i.e. packages, classes, and methods). In terms of primitive and composed entities a rule is shaping up stating that reasonable metrics are defined and computed only for composed design entities. The explanation resides basically in the distinctive aspects that exist between the two, i.e. the fact that a composed entity can contain other entities and that it can have relations with other entities (see Figure 4.2). As direct measurements (see Section 2.3.1) are mainly "counting"[2] the different entities contained in, or related to a measured entity, it becomes obvious why the object of measurement is restricted to composed design entities.

### 4.2.1 Composed Entities

As mentioned at the beginning of this section, a composed entity type is one that contains other entities. The aforementioned meta-model supports the following composed entities: *Package*, *Class* and *Method*. In order to describe a composed entity we have to specify its properties, containment structure (i.e. the entities that it contains), relations and container entity. The properties of an entity are classified in a set of categories. For each category a complete set of potential values was identified. The values in each category are mutually exclusive. The properties of any entity can therefore be seen as an *n-tuple* of values, where $n$ represents the number of property categories. An overview of the properties and containment relations for the composed entities is provided in Figure 4.4.

In this section we will introduce one by one the three composed entities that appear in our meta-model. The backbone of the presentation is given by the elements enumerated above. In order to enhance readability, a final section was dedicated (Section 4.2.3) to present the types of relations that a composed entity might have with other design entities. Some of the elements introduced next will be defined in a formal manner strongly relying on the formal definitions provided by Briand et.al. in [BDW99], [LB98].

#### Classes

From the point of view of the static structure, an object-oriented system consists of a set of classes, $C$. A class is a compound structure encapsulating data and functional elements [BDW99].

**Properties.** We identified three property categories for a class. Thus any class in our model will be associated with a 3-tuple of properties: one for each category. The three categories of properties are:

[2]This "counting" can be filtered based on the properties of the counted entities, so that only subsets are counted

| Entity Type | Container | Contained Entity Type | Properties: Kind Of | Properties: Possible Values |
|---|---|---|---|---|
| **Class** | Package | • Method<br>• Attribute<br>• Class (inner) | Abstraction | • concrete<br>• abstract<br>• interface |
| | | | Visibility | • normal<br>• inner<br>• public (Java) |
| | | | Reusability | • user-defined<br>• user-extended<br>• library |
| **Package** | — | • Class<br>• Package | — | |
| **Method** | Class | • Parameter<br>• Local Variable | Visibility | • private<br>• protected<br>• package (Java)<br>• public<br>• free (C++) |
| | | | Kind | • normal<br>• constructor<br>• destructor<br>• accessor<br>• friend (C++) |
| | | | Instantiation | • class (static)<br>• object (instance) |
| | | | Reuse | • normal (defined)<br>• overridden<br>• inherited<br>• library |
| | | | Abstraction | • concrete<br>• abstract |
| | | | Binding | • static<br>• dynamic (virtual) |

Figure 4.4: The Properties of Composed Entities

- *Abstraction property.* From the abstraction's perspective, a class can be: *concrete*, *abstract* or *interface-definition*. A concrete class is one that provides an implementation for all its methods. On the opposite side we find interfaces, which define just a set of method signatures without providing an implementation for any of them. Between the two extremes we find abstract classes, which contain at least a method without implementation; the abstract classes may also encapsulate data (as concrete classes do) and contain implementation of methods. Some languages (e.g. JAVA) provide distinct language mechanisms to differentiate between all these three types of classes; others (e.g. C++) although differentiating between concrete and abstract classes, do not provide any extra mechanism for interfaces. In such a language an interface is modelled by external conventions as an abstract class that contains only abstract methods.

- *Visibility property.* Different classes may have different visibility levels, going from *inner*, through *package* and up to *public* classes[3]. An inner class is one that is defined within another class, being directly visible only from within that class. A class with a "package" visibility is only accessible for the classes belonging to the same package, while "public" classes are those that are visible and accessible from any package in the system. The visibility property is relevant especially for coupling-related metrics.

- *Reusability property.* The last type of class properties is related to the degree of reuse for that class. The reusability may vary from *library* classes that are completely reusable, without any implementation or effort required on the side of the developer, up to classes that are designed and implemented by a developer "from scratch" (*user-defined*). An intermediate degree of reuse is given by classes that partially reuse an already implemented class by extending it and overriding some of the previously defined functionality (*user-extended*). As the reuse of a class involves a second part (the reused class) we will analyze this issue in some more detail in Section 4.2.3, where we discuss the relations among composed entities.

**Containment Structure.** A class may contain three types of entities: *Attributes*, *Methods* and other *Classes*. While the attributes for a class represent the data set encapsulated within the class, the set of methods specifies the behavior of the class. Finally, the classes contained in a class are the inner classes, those earlier mentioned in this section.

**Container Entity.** As seen in Figure 4.3, the container entity of a class is a package. Having made this statement, it may appear that we are ignoring inner classes. In reality, they are not neglected as this aspect is covered by the *visibility* property.

### Methods

Usually a class $c \in C$ contains a set of methods $M(c)$ that implement the control flow of the program. Other names used for "method" in different programming languages are: operation, service or member-function. The *set of all methods in the system*, $M(C)$, is defined for notational purposes.

**Properties.** From the functional point of view methods are the central element and therefore posses a rich set of properties. By classifying them, a number of six categories of properties was identified, and this means that each method will be characterized by a 6-tuple of properties - one property of each category.

[3]This visibility property is mainly based on the Java language mechanisms, although inner classes are also known in C++ [ES90]. Yet, even though not language independent, we decided to include it based on the most inclusive principle. In languages like C++ we could only differentiate between inner classes and the rest of the classes, which would then be considered to be *public*

- *Visibility property.* Methods can have different visibility levels varying from methods that are accessible only from the class they are defined into (*private*), followed by the methods that are accessible from derived classes (*protected*), then continuing with methods that are visible only to classes from the same package (*package*) and ending with the methods that build the interface of the class (*public*). In addition to these, the global (free) functions that appear in hybrid languages like C++ (*global*) are also considered.

- *Role property.* A method can have different roles within a class: with the exception of the usual methods (*normal*), a class has two special kinds of methods related to the creation (*constructor*) and destruction (*destructor*) of its instances. Another kind of special methods are the get/set methods (*accessor*) through which the data encapsulated in the class can be manipulated[4].

- *Instantiation property.* Considering the mechanisms for method-invocations, one notices that methods can be either called via objects (*object* methods) or be invoked independently of the existence of any instance (*class* methods).

- *Reuse property.* There are several kinds of methods with the respect to the reuse property: if a method is implemented from scratch, then it is considered a (*normal (new)*) method, where no reuse occurred; if a method of a class is not implemented at all but it is *inherited* from an ancestor of the class, then a total code reuse took place. A special case of reuse is one in which only the signature of a previously defined method is reused, meaning that the new method overrides another one inherited from an ancestor class (*overridden*). Finally, there is also a fourth category of methods, i.e. the library methods that are simply used without requiring any implementation effort on the developer's side.

- *Abstraction property.* As seen before in the class-related property, it is mainly the abstraction property of methods that decides the homonymous property for classes. A method can be provided with either a concrete implementation (*concrete*), or it may consist only of the method-signature that makes it an abstract method.

- *Binding property.* This method property captures the way an implementation is bound to the invocation of the method with a given signature. There are two known possibilities: either the method is bound at compile-time (*static*) or it is late-bound (*dynamic (virtual)*) meaning that we are dealing with a polymorphic call.

Based on the reusability property category, the following definitions can now be provided [BDW99]:

[4]We do not discuss here the quality impact of using accessor-methods. This aspect is partially covered in Section 2.2 and in Section 5.3.1

**Definition 4.2 (Declared Methods. Implemented Methods)** *For each class $c \in C$, let*

- $M_D(c) \subseteq M(c)$ *be the set of methods declared in c, i.e. methods that c inherits but does not override, or virtual methods of c*
- $M_I(c) \subseteq M(c)$ *be the set of methods implemented in c, i.e. methods that c inherits but overrides or non-virtual, non-inherited methods of c.*
- *where* $M(c) = M_D(c) \bigcup M_I(c)$ *and* $M_D(c) \bigcap M_I(c) = \emptyset$

**Definition 4.3 (Inherited, Overriding and New Methods)** *For each class $c \in C$, let*

- $M_{INH}(c) \subseteq M(c)$ *be the set of inherited methods of c*
- $M_{OVR}(c) \subseteq M(c)$ *be the set of overriding methods of c*
- $M_{NEW}(c) \subseteq M(c)$ *be the set of non-inherited, non-overriding methods*

**Containment Structure.** In this meta-model, a method contains two types of entities: *Parameters* and *Local Variables*; both of these are primitive entities and are described in the next section.

**Definition 4.4 (Parameters. Local Variables)** *For each method $m \in M(C)$ let $Par(m)$ be the set of parameters of the method m, and $LocVar(m)$ be the set of local variables declared in the implementation of method m.*

**Container Entity.** The container entity for a method is a *Class* entity. An exception from this rule occurs in languages that allow methods to exist without any encapsulation in a class (e.g. C++). In the model considered here these methods are recognized by the value of their *visibility* property, which is set to *free*.

### Packages

In large systems classes are too small-grained to support the complete structuring of the system. In order to achieve this, packages (or subsystems) are used for a further, higher-grained level of system decomposition. This can be provided as part of a programming language (e.g. the *package* concept in JAVA) or as an external convention (e.g. by using a hierarchical directory structure).

**Properties.** Because packages are only a simple, high-level structuring mechanism that in some cases is not even reflected in a language construct, there are no properties associated with it.

**Containment Structure.** A package (subsystem) may contain a set of other *Packages* and/or *Classes*. The relation is usually, but not necessarily, hierarchical.

**Container Entity.** No other entity type that contains the *Package* entity is defined because the system is considered as being decomposed in packages.

### 4.2.2 Primitive Entities

The primitive entities that are considered in this model are: *Attribute*, *Parameter*, *Local Variable* and *Global Variable*. It becomes obvious from this enumeration that all the primitive entities are in fact those program elements that hold data. These items are usually called *Variables*.

Primitive entities being all variables, share a set of properties but are also based mainly on their *container entity*: the container of any *Attribute* entity is therefore a *Class* entity, while the container of the *Parameter* and *Local Variable* entities is a *Method*. The *Global Variable* entity is a special case of variable that occurs only in hybrid languages like C++ and it was added to the meta-model because this work also addresses systems implemented in such languages. Just like the free methods, a *Global Variable* has no container except for the package that contains the file in which the global variable is defined.

The properties of all the primitive entities are summarized in Figure 4.5.

| Entity Type | Container | Applicable to | Properties | |
|---|---|---|---|---|
| | | | Category | Possible Values |
| Parameter | Method | All Variables | Type | ▪ built-in<br>▪ user-defined<br>▪ library |
| Local Variable | | | Aggregation | ▪ simple<br>▪ array |
| Global Variable | — | | | |
| Attribute | Class | Attributes only | Visibility | ▪ private<br>▪ protected<br>▪ package (Java);<br>▪ public |

Figure 4.5: The Properties of Primitive Entities

**Properties.** Two categories of properties that characterize all the variables were identified. An additional category (i.e. Visibility) was introduced for a complete description of the *Attribute* entity. In this manner, any parameter, local variable or global variable will be characterized by a 2-tuple of properties, while attributes will be described using a 3-tuple, as follows:

- *Type property*. Each variable must have a type. The type can be a built-in type, provided by the programming language (*predefined*), or it can be a *user-defined* type. Special cases of user-defined types are those that are part of a *library*. Sometimes in metrics definitions we want to distinguish between these two situations.

- *Aggregation property.* The aggregation property addresses the storing capacity of a variable. From this point of view a variable may store a single value (*simple*) or a set of values (*array*).
- *Visibility property.* Similar to methods, an *Attribute* entity, being a member of a class, can have different visibility levels: it can be completely hidden within the class (*private*); it can be accessible also from derived classes (*protected*); it can be made visible only to classes from the same package (*package*) or it can be declared as part of the interface of the class (*public*).

In addition the previous properties, an *Attribute* entity, may be implemented in a class or inherited. Therefore, we can now provide the following definitions [BDW99]:

**Definition 4.5 (Declared Attribute. Implemented Attribute)** *For each class* $c \in C$, *let* $A(c)$ *be the set of attributes of class* $c$. $A(c) = A_D(c) \bigcup A_I(c)$ *where*

- $A_D(c) \subseteq A(c)$ *is the set of attributes declared in* $c$, *i.e. attributes that* $c$ *inherits;*
- $A_I(c) \subseteq M(c)$ *is the set of attributes implemented in* $c$, *i.e. non-inherited attribute.*

### 4.2.3 Relations Between Entities

In Figure 4.6 we summarize the types of relations that exist between the entities from the meta-model.

| Entity Type | Relation | |
|---|---|---|
| | Kind Of | Second Entity |
| **Class** | *extend* | Class |
| | *implement* | Class [Abstraction == interface] |
| **Package** | — | — |
| **Method** | *access* | Attribute |
| | *access* | Variable |
| | *call* | Method |
| | *override* | Method |
| | *implement* | Method [Abstraction == abstract] |

Figure 4.6: The Relations of the Composed Entities

**Remark.** In order to understand the relations identified in this meta-model, an important remark should be made at the beginning of this section: for each entity, we consider only those relations in which it *directly* interacts with other entities. Based on this principle, the relations "inherited" from the entities contained within a composed entity were disregarded.

For example, although we may speak about "a class accessing a method" or a "class calling another class", these relations appear in our model only for the *Method* entity and not for the *Class* entity, as only the *Method* entity of a class stays in a direct "access"-relation with a *Variable* entity. A corollary of the previously enounced principle states that the relations are transitive, i.e. a composed entity transitively inherits all the relations of its component entities.

In Chapter 2 the different relations that appear in object-oriented systems were discussed (see Section 2.1). Therefore, in this section only a set of notational definitions for these relations will be provided.

### Class Inheritance. Interface Implementation

There can exist two types of direct relations among classes: either a class is the *specialization* of another class, or it is the *implementation* of an interface class. Thus for a class $c \in C$ we can define the following sets of related classes:

- $Parents(c) \subset C$, the set of classes directly inherited by class $c$. In languages that do not allow multiple inheritance, the cardinality of this set is obviously either zero or one. The transitive closure of parent classes defines the set of ancestors: $Ancestors(c) \subset C$ $(Parents(c) \subset Ancestors(c))$
- $Children(c) \subset C$, the set of classes directly inheriting from class $c$. The transitive closure of children classes defines the set of descendants: $Descendants(c) \subset C$ $(Children(c) \subset Descendants(c))$
- $Interfaces(c) \subset \{i \in C | abstraction(i) = "interface"\}$, the set of interfaces implemented by class $c$.

### Method Invocations. Variable References

In order to define measures of the coupling for a class $c$, it is necessary to define the set of methods that are called by any method $m \in M(c)$ and the set of variables referenced by any method of the class $c$. They are defined as follows [LB98]:

**Definition 4.6 (Called Methods)** *Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in CM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of $m$ has a method invocation where $m'$ is invoked for an object of type class $d$, or $m'$ is a class method of type class $d$.*

**Definition 4.7 (Variable References)** *For each $m \in M(c)$ let $VR(m)$ be the set of variables referenced by method $m$.*

## 4.3 Detection Strategy

As we have seen in the beginning of this chapter, the main issue in applying software metrics for the assessment of quality in object-oriented design is to get a relevant interpretation of the measurement results. In the previous chapter (Section 3.2) the limitations of existing interpretation models for individual metrics were pointed out. In order to synthesize the main point of our criticism we will use a medical metaphor: interpretation models can offer an understanding of symptoms reflected in abnormal measurement results, but they cannot bring the understanding of the disease that caused the symptoms. The *bottom-up approach*, – i.e. to go from abnormal numbers to the recognition of design diseases – is impracticable because the symptoms captured by single metrics, even if perfectly interpreted, may occur in several diseases. The interpretation of individual metric results is too fine-grained to point out the disease. This leaves us with a major gap between the things that we measure and the things that are in fact important at the design level with respect to a particular investigation goal.

In order to overcome this limitation, in the rest of this chapter we will introduce a high-level, goal-oriented investigation mechanism that will allow us to define a usable top-down investigation approach. For the first time we will be able to express and detect in a quantifiable manner the things that are important for the assessment and improvement of the design. In this context, answers to the following questions will be provided:

- How to interpret *individual results*? We will discuss the different mechanisms for **data filtering** and provide a concrete methodology for the selection of the adequate filter, based on the assumption that for different measurement goals, different filtering mechanisms should be used.
- How to correlate the interpretation of a *multiple set of results*? Concerning this question the answer is the **composition mechanism** provided by the *detection strategy*.
- How to go from a concrete *investigation goal* to a detection strategy that quantifies that goal?

### 4.3.1 Definition

The main issue in working with metrics is how should we deal with measurement results. How can all those numbers help us improve the quality of our software? As stated earlier, most of the times a metric alone cannot help very much in answering this question and therefore metrics must be used in combination to provide relevant information. But how should we combine metrics in order to make them serve our purposes?

The main goal of the mechanism presented below is to provide the engineers

with a mechanism that will allow them to work with metrics on a *more abstract level*, which is conceptually much closer to the real intentions in using metrics. The mechanism defined for this purpose is called *detection strategy*:

**Definition 4.8 (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.*

A *detection strategy* is therefore a generic mechanism for analyzing a source code model using metrics.

**Remarks**

1. In the context of the previous definition, "*quantifiable expression of a rule*" means that the rule must be properly expressible using software product metrics.

2. One of the main goals of this dissertation is to use detection strategies to express rules that will help us detect *design problems* in object-oriented software system, i.e. to find those design fragments that are affected by a particular design flaw. At this point we want to emphasize that the *detection strategy* mechanism and the whole technique is not limited to problem detection, but it can serve other purposes as well.

   For example different investigation goals could be reverse engineering (model capture, design recovery) [Cas96], detection of design-patterns [Mar01], identification of components in legacy systems [Tri01], etc.

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.

### 4.3.2 The Filtering Mechanism

**Definition 4.9 (Data Filter)** *A* data filter *is a mechanism (a set operator) through which a* subset *of data is retained from an initial set of measurement results, based on the particular focus of the measurement.*

The key issue in filtering data is to reduce the initial data set, so that only those values are retained that present a special characteristic. This is usually called *data reduction* [Hoe54]. The purpose for doing that is to detect those design fragments that have special properties captured by the metric. Considering the initial set as being sorted, a data filter defines a *contiguous data subset*. Thus, in order to define a data filter we need to define the values for the bottom and upper limits of the filtered subset.

The limits of the subset are defined based on the type of data filter. In the

context of measurement activities applied to software, we usually search for *extreme (abnormal) values* or values that are in a *particular range.* Therefore we identify two types of filters:

- *Marginal Filter* – a data filter in which one margin of the result set is implicitly identified with the corresponding limit of the initial data set.
- *Interval Filter* – a data filter in which both the lower and the upper limit of the resulted subset are explicitly specified in the definition of the data set.

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

Figure 4.7: Classification of Data Filters

Based on the previous two definitions, we identify a set of characteristics and further classifications of the two types of data filters (see Figure 4.7). In the following, we will discuss these aspects in more detail.

### Marginal Filters

Depending on how we specify the limit(s) of the resulted data set, marginal filters can be sub-divided into two categories:

1. **Semantical** For the filters belonging to this category we must specify two parameters: a *threshold value* and a *direction.* The threshold value indicates the one margin that must be explicitly specified, while the direction tells us if the threshold value is the upper or the lower limit of the filtered data set. We named this category *semantical* because in the context of measurement interpretation, the selection of the parameters is based on the particular semantic of the metric, captured by the interpretation model for that metric. The quality of a detection strategy strongly depends on the proper selection and parametrization of a filtering mechanism.
2. **Statistical.** In contrast to the semantical filters, statistical ones do not need an explicit specification for a threshold, as it is determined directly from the initial data set by using statistical methods (e.g. box-plots, standard deviation). However, the direction still must be specified just like in the case of semantical filters. Statistical filters are built based on

the assumption that all the measured entities of the system are structurally designed using the same style and therefore the measurement results are comparable.

Through this work we used a set of concrete data filters that can be grouped as follows (see also Figure 4.7), based on the previous classifications:

- **Absolute Semantical Filters**: `HigherThan` and `LowerThan`. These filtering mechanisms can be parameterized with a numerical value representing the threshold. We will use such data filters to express "sharp" design rules or heuristics, – e.g. "a class should not be coupled with more than 6 other classes. Please note that the threshold is specified as a parameter for the filter, while the two possible directions are captured by the two concrete filters.

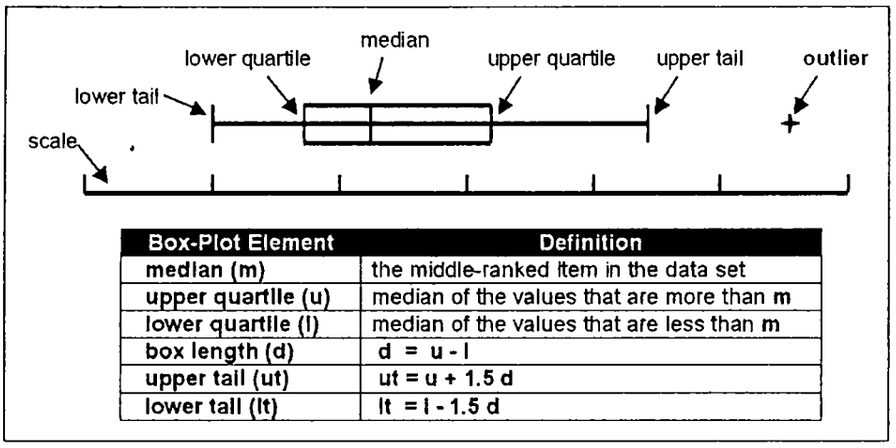
- **Relative Semantical Filter**: `TopValues` and `BottomValues`. These filters are delimitating the filtered data set by a parameter that specifies the *number* of entities to be retrieved, rather than specifying the maximum (or minimum) value allowed in the result set. Thus, the values in the result set will be *relative* to the original set of data. The used parameters may be *absolute* (e.g. "keep the 20 entities with the highest values") or *percentile* (e.g. " retrieve the 10% of all measured entities having the lowest values"). This kind of filters is useful in contexts where rather than indicating precise thresholds we would like to consider the highest (or lowest) values from a given data set. They should be therefore used for the quantification of design heuristics that are rather fuzzy, not specifying precise thresholds (e.g. "methods of high complexity should be split"). For such design rules concepts of "high" and "low" are relative to the initial data set.

- **Statistical**: `BoxPlots`. Box-plots are a statistical method by which the abnormal values in a data set can be detected [FP97] [CCKT83]. Data filters based on such statistical methods, which are of course not limited to box-plots, are useful in the quantification of rules that although "fuzzy", concern with *extreme values* in a data set (e.g. "Avoid packages with an excessively high number of classes"). Again, we have to specify the direction of the outlier values based on the semantics of the design rule that is quantified.

With the exception of box-plots, all the previously mentioned data filters are trivial, well known and need no further explanations. The definition of the box-plots technique is described in Figure 4.8.

### Interval Filters

At first sight, for an interval data filter we need to specify two threshold values. But, in the context of detection strategies, where in addition to the filtering mechanism we also have a composition mechanism, interval filters are in all cases reducible to a composition of two marginal filters of opposite directions.




| Box-Plot Element | Definition |
|---|---|
| median (m) | the middle-ranked item in the data set |
| upper quartile (u) | median of the values that are more than m |
| lower quartile (l) | median of the values that are less than m |
| box length (d) | d = u - l |
| upper tail (ut) | ut = u + 1.5 d |
| lower tail (lt) | lt = l - 1.5 d |

Figure 4.8: The box-plot technique as statistical method used for abnormal measurement values [FP97]

In Figure 4.7 a simple example illustrates how the interval filter `Between(20,30)` can be composed out of two (semantical) marginal filters, i.e. `HigherThan(20)` and `LowerThan(30)`.

### Metrics and Data Filters

Filtering mechanisms are always related to a metric as described next:[5]

```
MetricExpression := "(" Metric "," Filter ")"
Metric           := <problem-specific metric from a repository>
Filter           := HigherThan | LowerThan | TopValues |
                    BottomValues | BoxPlots
```

## 4.3.3 Methodology for Selecting Data Filters

In order to summarize the previous description of individual data filters, in the following, we provide a brief methodology by stating a set of rules. The rules should guide the engineer in deciding which type of data filter should be applied on the results of a particular metric, while quantifying the design rules or heuristics.

**Rule 1** Choose an *absolute semantical filter* when quantifying design rules that specify explicitly concrete threshold values.

[5]The description is given in the form of a set of grammar rules of the SOD language. More details are provided in Section 7.2.4

**Rule 2** Choose a *relative semantical filter* when the design rule is defined in terms of fuzzy marginal values, like *"the high/low values"* or *"the highest/lowest values"*.

**Rule 3** For large systems, parameterize *relative semantical filters* using *percentile* values. On the other hand, use *absolute* parameters when applying a relative semantical filter on small-scale systems.

**Rule 4** Choose a *statistical filter* for those cases where the design rules make reference to extremely high/low values, without specifying any precise threshold.

### 4.3.4 The Composition Mechanism

As stated at the beginning of this section, a detection strategy is the *"quantifiable expression of design rule"*. Thus, in contrast to simple metrics and their interpretation models, a detection strategy should be able to quantify entire design rules. As a consequence, in addition to the filtering mechanism that supports the interpretation of individual metric results, we need a second mechanism to support a correlated interpretation of *multiple result sets* - this is the *composition mechanism*. The composition mechanism is based on a set of operators that "glue" different metrics together in an "articulated" rule.

**Definition 4.10 (Composition Operators)** *The operators used to compose a set of metrics into an "articulated" rule are called* **composition operators.**

We defined three composition operators: `and`, `or` and `butnot`. The *composition operators* can be seen from two different viewpoints:

- the *Logical Viewpoint*. From the logical perspective, the three operators are the reflection of the "joint-points" of a design rule quantified by the detection strategy, in which the "operands" are in fact a description of design characteristics (symptoms). They allow easy reading and understanding of the detection strategy because through the composition operators the quantified rule becomes similar to the original, informal one. From this point of view, for example, the `and` operator suggests the co-existence of both the symptom described on the left side of it as well as the existence of the symptom presented on the right side.
- the *Set Viewpoint*. The set viewpoint helps us understand how the final result of a detection strategy is built (Figure 4.9). Through the filtering mechanism the initial set of measurement results for each metric involved in the strategy ($M_1 \ldots M_n$) is reduced to a set of design entities (and their measured values) that were considered interesting in conformity with the interpretation model ($F_1 \ldots F_n$). After that, the several filtered sets must be combined using the composition operators. Thus, in terms of set operators, the `and` operator will be mapped to *intersection ("$\bigcap$")*, the `or` operator to *reunion("$\bigcup$")* and the `butnot` operator to *set minus ("$\setminus$")*.

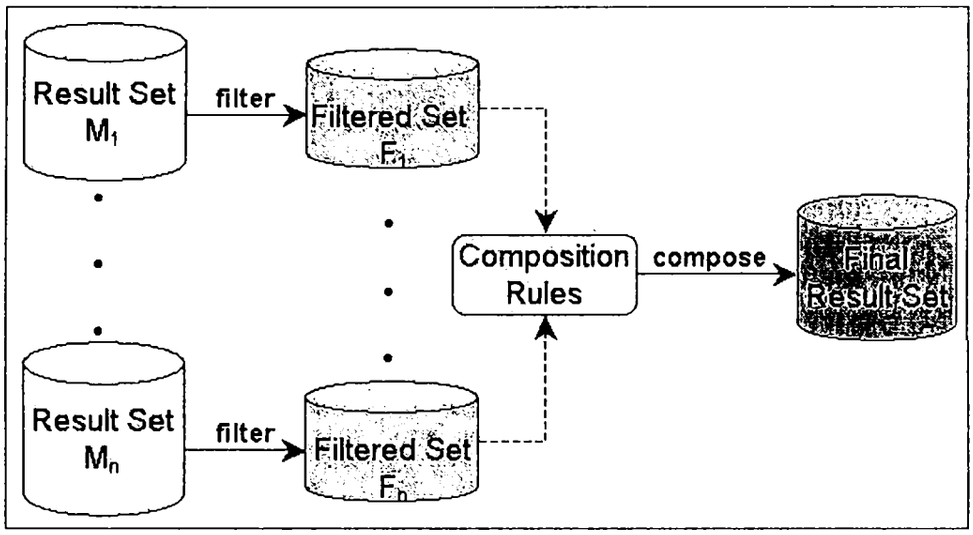



Figure 4.9: The filtering and composition mechanisms from a set viewpoint

**An Example: Detecting Data-Classes**

Let's assume that we want to find a set of classes that exhibit their data in the interface. We decided to use two metrics: the first one to count the number of public attributes (NOPA) and the other one to count the number of accessor-methods (NOAM). We decide that the classes we want to find are those with the most public data from the project, but that they should not have less than 3 public attributes or 5 accessor-methods. Therefore, the following detection rule (in SOD) is used:

```
DataClasses :=
    (NOPA, HigherThan(3) and NOPA, TopValues(10%)) or
    (NOAM, HigherThan(5) and NOAM, TopValues(10%))
```

## 4.4 Defining Detection Strategies

The starting point for this approach is an informal description of a rule related to the design. This is the rule that we try to quantify. In the recent years we have often found in the literature various forms of descriptions for such rules, especially in the context of reengineering [Cas98]. As we have seen in 3.4, R. Martin discusses the main design principles of object-orientation and shows that their violation leads to a "rotting design" [Mar00].

The approach consists of the following sequence of steps:

1. *Analysis of the Design-Rule.* After choosing the design-rule that should be quantified, the first step is to express the informal description of the rule in a quantitative manner. In other words, we have to describe how the rule

is related to the design entities (e.g. for a rule describing a design flaw we may have class inflation, excessive method length) and the relationships among them (e.g. highly coupled subsystems). The result of this step is the definition of a concrete strategy for detecting the conforming entities based on the analysis of the informal description.

2. *Selection of Metrics.* Based on the quantitative description of the design rule, those metrics must be found or defined that are most suitable for the measurement of the characteristics of the sought design structure. At the end of this step, the detection strategy can be expressed as a specific combination of the selected metrics.

3. *Detection of Candidates.* The third step is to measure the system based on the defined detection strategy, using the chosen metrics. The result of this step is a list of design fragments that are supposed to be conforming to that design-rule.

4. *Examination of Candidates.* The last step is to examine the identified design fragments and to decide, based on the source-code and on other information sources, if the candidates are indeed what we were seeking or if the proposed detection strategy didn't really properly capture the initial (informal) design-rule.

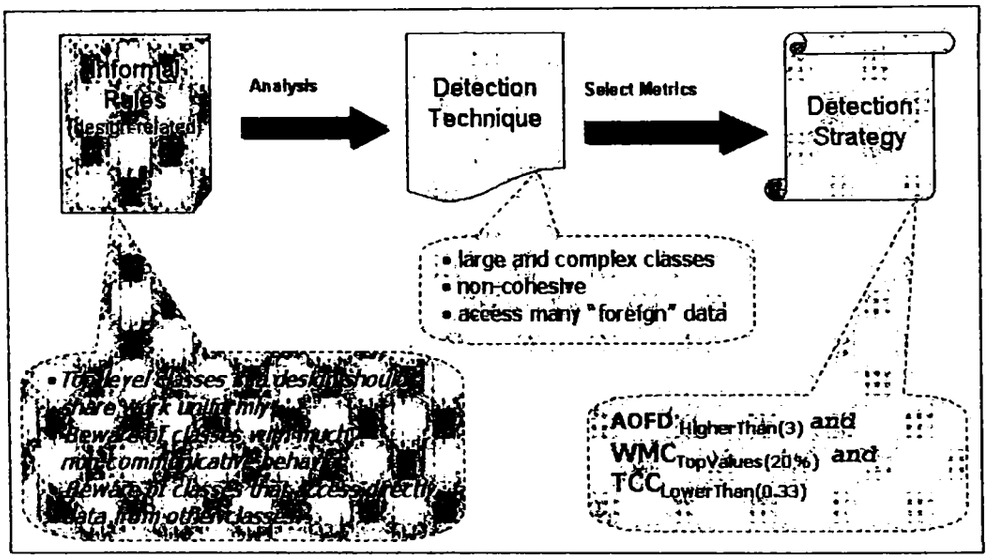



Figure 4.10: The steps for transforming an informal design rule in a detection strategy. In the bottom part of the picture the transformation is exemplified by using the description of the "Behavioral God-Class" design problem [Rie96]

Concerning these four steps, there is an important distinction to be made between them: while the first two steps describe the *definition* of the detection

technique (see Figure 4.10), the last two describe how to *apply* the technique on the examined system. Thus, after a detection technique is defined and validated, only the last two steps must be covered for detecting a particular rule in a given system.

### 4.4.1 Template for Describing a Detection Strategy

We describe each particular detection strategy using a consistent format. This format grants a uniform structure to the detection process, making it easier to understand and apply. The *definition* of a detection strategy is divided into sections, according to the following template.

- *Name.* Each detection strategy should have a name. Similar to patterns [GHJV94], the name of a detection strategy provides the strategy with an identity, which increases its communication level. In order to be useful, the name must capture the particular aspect quantified and detected by the strategy. As we will see in Chapter 6, when strategies are used as parts of quality models, their name becomes an important vehicle of semantical knowledge. In that context it helps the engineer not only find critical design fragments, but also allows immediate reasoning about design improvements.

- *Motivation.* In this section the design-rule is shortly explained with a special focus on its impact on the structure and quality of the design (e.g. flexibility, maintainability, etc). From this description we derive the characteristics of the structural design entities (e.g. classes, methods, subsystems) that are affected by the corresponding design problem.

- *Strategy.* This section describes the concrete strategy to use in order to find the conforming entities. The strategy is based on the characteristics of the design-entities that were presented in the *Motivation* section.

- *Metrics.* As our approach is a metrics-based one, we need to select those metrics that we are going to use for the detection of a particular design rule. This section introduces the metrics that are needed to support the detection strategy described above. For each metric three aspects must be provided: the *definition* of the metric, its *interpretation model* and the specification of *outliers.* Note that a metric might be used in several detection techniques and while the definition of the metric is independent of the context in which it is used, its interpretation model and sometimes the outliers are related to the technique.

# Chapter 5

# Detection of Design Flaws

There is no perfect software design. Like all human activities, the process of designing software is error prone and object-oriented design makes no exception. The flaws of the design structure have a strong negative impact on quality attributes such as flexibility or maintainability. Thus, the identification and detection of these design problems is essential for the evaluation and improvement of software quality.

The goal of this chapter is to show how structural flaws in object-oriented design can be quantified and identified using the concept of detection strategy from the previous chapter. In other words, we will apply the higher-level interpretation mechanism defined in Chapter 4 to a concrete investigation goal, i.e. the detection of design problems.

In order to do this we first define the concept of design flaw in the context of this work (Section 5.1) and then classify these flaws according to the granularity of the design entity affected by them. Based on this classification, we build through the rest of the chapter a coherent suite of detection strategies that address a set of well-known design problems found in the literature. The chapter is closed by several summarizing conclusions.

## 5.1 Design Flaws

In the context of this work we define design flaws as follows:

**Definition 5.1 (Design Flaw)** *The structural characteristic of a design entity or design fragment that expresses a deviation from a given set of criteria typifying the high-quality of a design is called a* **design flaw**.

The different parts of this definition express concisely the boundaries of the object of our further investigation. Therefore we will analyze each part of the definition and provide additional explanations that will increase the precision of the concept of design flaw as used in this work:

- *"The structural characteristic..."*. There are various types of flaws that may affect the quality going from erroneous or error-prone implementations to missing or incoherent documentation. This part of the definition limits our work to the analysis of those flaws that can be mapped to the design structure. Thus, we are not dealing with rules concerning the compliance of the code with coding standards or implementation rules (e.g. "avoid assignments inside conditions", or "capitalize words in class-names"); neither do we refer to flaws related to the quality of the development process or documentation.

- *"... characteristic of a design entity..."*. In conformity with this definition, a design flaw is a negative *property* of a design entity and not the entity itself, as the same entity (or fragment) may be affected by more than one flaw.

- *"... of a design entity or design fragment..."*. Trivially, a design entity is a method, a class or a package[1] of the analyzed system. But a design flaw oftentimes does not affect only an isolated entity, but also several other entities that are connected by different types of association relations. Thus, a *design fragment* stands for a "cluster" of such collaborating design entities.

- *"... a set of criteria typifying the high-quality of design"*. We may classify flaws in *functional* and *non-functional* flaws. This part of the definition delimits our approach from others that are dealing with the detection of functional flaws (i.e. bugs); among these we may mention Zeller's advanced methods of detecting functional flaws (failures) [CZ00] [Zel99]. Therefore, this definition part links our work to the non-functional issues of good-design. As stated in the previous definition, good design is characterized by a set of criteria, as discussed in Chapter 2 (Section 2.2).

- *"... deviation from a given set of criteria"*. Evaluating a given design in terms of *deviations* from a set of good-design criteria requires a quantification of these criteria. As discussed in the previous chapters, on the one hand each criterion is reflected by concrete, paradigm-specific design rules and guidelines (Section 2.2) while on the other hand, using the approach described in the previous chapter makes these concrete rules quantifiable when expressed in terms of *detection strategies* (Section 4.3).

### Remark

Design flaws are hard to define, because sometimes we encounter situations in which a code fragment might be considered flawed in one case while in another case, a similar, mostly identical design fragment is justifiable and may not be considered a design flaw. This brings us to the question of how to differentiate between these cases.

[1] "Design entity" in the context of this definition corresponds to the definition of the *composed design entity* as defined in Section 4.2

From this point of view we identify two types of programming:

- *Implementation of Algorithms and Data Structures.* This includes all those program fragments that implement different algorithms or data-structures that have a well-founded theoretical background. To this category belong for example sorting algorithms, graph-related algorithms or the implementation of container classes. The main characteristic of these program fragments is that they are written in a compact manner and their understanding is only accessible to those who know the theory on which the implementation relies.

- *Organizational Programming.* This refers to those relevant parts of the code that implement (capture) the whole semantic of the application. Here, different algorithms are combined and different data-structures are used in manners determined by the semantical logic of the application. There is no theoretical foundation for this part of the programs.

The difference between the two types of programming becomes relevant when we want to assess the quality of the design and consequently when we go on the quest for design flaws. The main difference lies in the *level of maintainability concern* for the two categories: which is very low for the former category while being extremely high for the latter.

### 5.1.1 Classification

There are several ways in which design flaws can be classified; they could be classified according to their abstraction level or to the good-design criteria they are deviating from. In this chapter we decided to classify design flaws according to the granularity level of the design entity affected by each flaw. Regarding the meta-model, we identified three types of design entities for which relevant metrics can be defined: *methods*, *classes* and *subsystems* (packages). As the detection strategies that we will use to localize the design flaws are based on metrics, the detection of flaws will obviously occur at these three levels of granularity.

In addition to the method, class and subsystem level, we introduce in this chapter a fourth one: the *micro-design* level. Design-flaws that occur at this level affect not only a class, but a cluster of classes. A typical case for such flaws is a design fragment where a particular design pattern should have been applied, but the pattern solution was ignored. In Section 5.5 we will present an approach for the detection of such flaws.

In order to lend a coherent structure to the general discussion on each category of design flaws, we will focus the presentations on the relation to the good-design criteria specified in Section 2.2. As a consequence, we will introduce each detectable design flaw in terms of deviations from these criteria.

## 5.2 Design Flaws in Methods

In the context of an object-oriented system, a method is the smallest design fragment that can be affected by a design flaw. On the other hand, methods are the ultimate containers of the control-flow and behavior of the system. From this point of view, the large majority of the principles and rules of structured programming [PJ88] are applicable at the method level. Yet, because this work is focused on the quality of *object-oriented* design, we decided to address the issues related to good procedural design in a "shallow" manner, taking into account only those aspects that are relevant in the context of object-orientation.

Excepting the specific flaws directly related to structured programming, design flaws at the method level are related either to an unequal distribution of the functionality between the methods of a class, or to a misplacement of a method within a class. In terms of good-design criteria, an unequal distribution of functionality is a deviation from the criteria of *manageable complexity* (Section 2.2.3) while the misplacement of a method affects both the *cohesion* (Section 2.2.2) of the system and is a sign of an improper *data abstraction* (Section 2.2.4).

We identified two design flaws at the method level, one for each of the two aforementioned cases:

1. **Feature Envy** (Section 5.2.1) is one of the "bad-smells" described in [FBB+99] and it is related to the misplacement of a method within a class.

2. **God Method** (Section 5.2.1) is defined as a composition of lower-level symptoms described by several authors ([FBB+99], [JF88] and [Rie96]) that captures the characteristics of a malign method, affected by an excessive size and complexity.

Next, we will describe in detail the detection strategies for both design flaws.

### 5.2.1 Feature Envy

**Motivation** Objects are a mechanism for keeping together data and the operations that process the data. This flaw [FBB+99] refers to those methods that seem more interested in the data of another class than the one of its own class. These methods access directly or via accessor-methods a lot of data from another class. This might be a sign that the method was misplaced and that it should be moved to another class.

**Strategy** The detection is based on counting the number of data members that are accessed (directly or via accessor-methods) by a method from outside the class where the method under scrutiny is defined.

This problem can be solved if the method is moved into the class that has the greatest coupling factor with the aforementioned method. If only a part of the method suffers from a *feature envy* it might be necessary to extract that part into a new method and after that to move that method into the "envied" class.

### Metrics

1. **Access of Import-Data (AID)**
   - *Definition:* AID is the amount of data members accessed in a method, either directly or via accessor-methods[2].
   - *Implementation Details:* These are the attributes of all the classes from which the definition-class of the method is not derived.
2. **Access of Local Data (ALD)**
   - *Definition:* ALD counts the number of the data members accessed in the given method, which is local to the class where the method is defined.
   - *Implementation Details:* Inherited data should be counted too.
3. **Number of Import Classes (NIC)**
   - *Definition:* This metric counts the number of external classes from which the given method uses data.

### Strategy

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

**Remark.** In analyzing this design flaw we detected two alternative detection methods. Here they are:

1. **Count all dependencies**. Another way to detect *Feature Envy* would be to consider *all* the dependencies of the measured method, instead of considering only the *data-members* accessed by a particular method. In this case we would count both the dependencies on the class where the method is defined, and those on the other classes defined in the system.
2. **Ignore dispersion**. We used the NIC metric in the detection strategy because we were focused on detecting those methods that can be easily moved to another class and this involves a reduced dispersion of the classes on which the methods relies. But we might want sometimes to eliminate this restriction and in this case we will again find methods that rely on data taken from *many* other classes. Although in this case moving the method is not obvious, such methods might still require refactoring.

[2] For more comments on accessor-methods please refer to the NOAM metric

### 5.2.2 God Method

**Motivation** Oftentimes a method starts as a "normal" method but then more and more functionality is added to it, until it gets out of control becoming impossible to maintain or understand. Thus, *God Methods* tend to centralize the functionality of a class, in the same way *God-Classes* (see Section 5.3.1) centralize the functionality of an entire subsystem, or sometimes even of a whole system.

**Strategy** We defined the strategy for detecting this design flaw based on the presumed convergence of three simple bad-smells described in [FBB+99]. Thus, we are looking for the following symptoms:

- *Long methods.* They are undesirable because they affect the understandability and testability of the code. Long methods tend to do more than one operation, and they are therefore using lots of temporary variables and parameters, which make them more error-prone.
- *Long Parameter List.* Long parameter lists are hard to understand, they are difficult to use because they tend to change all the time. Long parameter lists are reminiscent of the procedural programming, where parameters were a proper alternative to global data.
- *Switch Statements.* The intensive use of switch statements is in almost all cases a clear symptom of a non object-oriented design, in which polymorphism is ignored[3]. There are two essential problems with switch statements: they make the design inflexible and they tend to duplicate the code.

**Metrics**

1. **Lines Of Code (LOC)**
   - *Definition:* LOC is the number of lines of code in a method, including comments and white-lines.
2. **Number Of Parameters (NOP)**
   - *Definition:* NOP is the number of parameters that build the signature of a method.
3. **Number Of Local Variables (NOLV)**
   - *Definition:* NOLV counts how many local variables are declared within a method.

[3]On the other hand, we should note that the excessive use of polymorphic methods introduces further testability and analysability problems [Bin99]. Yet, the emphasis in the context of this design flaw lies on a very frequent case in which legacy systems migrated from structured to object-oriented programming.

4. **Maximum Number Of Branches (MNOB)**

   - *Definition:* MNOB is defined as the maximum number of `if-else` and/or `case` branches in a method.

**Strategy**

```
GodMethod := (LOC, TopValues(20%)) butnotin (LOC, LowerThan(70)) and
             ((NOP, HigherThan(4) or (NOLV, HigherThan(4))) and
             (MNOB, HigherThan(4))
```

**Remark** Instead of the MNOB metric we could alternatively use McCabe's cyclomatic number [McC76] as a measure of the complexity of the method.

## 5.3 Flaws of Class Design

Most of the flaws described in the literature, in connection with object-oriented design, are related to class-design. Each of the four criteria discussed in Section 2.2 are expressed by a multitude of design rules and heuristics. These rules are at different granularity levels, yet most of them are expressible in the terms of the design structure. We briefly discuss next the most relevant design-flaws for which we defined detection strategies, pointing out for each case the deviations from the good-design criteria and the impact on external quality characteristics:

- **Data Classes** [FBB+99] [Rie96]. Data-classes are dumb data holders and almost certainly other classes are strongly relying on them. The lack of functional methods may indicate that related data and behavior are not kept in one place; this is a sign of an improper data abstraction (Section 2.1.1). Data classes impair the maintainability, testability and understandability of the system.

- **God Classes** (Section 5.3.1). In a good object-oriented design the intelligence of a system is uniformly distributed among the top-level classes [Rie96]. This design flaw refers to those classes that tend to centralize the intelligence of the system. An instance of a god-class performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes. *God-classes* deviate from the principle of manageable complexity (Section 2.2.3), as they tend to capture more than one abstraction (Section 2.1.1); consequently, such pathological classes tend to be also non-cohesive (Section 2.2.2). Thus, *god-classes* have a negative impact on the reusability and the understandability of that part of the system that they belong to.

- **Shotgun Surgery** (Section 5.3.2) [FBB+99]. This bad-smell is encountered every time when a change operated in a class involves a lot of small changes to a lot of different classes. When the changes are all over the

place, they are hard to find and therefore it is very possible to miss an important change. Thus, this design flaw strongly affects the maintainability of the system.

- **Refused Bequest** [FBB+99]. A subclass inherits some of the methods and the data of its parents. This design flaw refers to the case when the subclasses don't use the members of their ancestors that were particularly designed to be reused by the descendants. This might be a sign that the hierarchy is wrong, although if the child class refuses only the inherited implementation things might not be that bad; but if the descendant refuses the interface then the hierarchy is definitely wrong. As we already discussed in Section 2.2.2 this design flaw is a violation of the principle of cohesive inheritance relationships.

### 5.3.1 God Classes

**Motivation** In a good object-oriented design the intelligence of a system is uniformly distributed among the top-level classes [Rie96]. This design flaw refers to those classes that tend to centralize the intelligence of the system. An instance of a god-class performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes. This has a negative impact on the reusability and the understandability of that part of the system. This design problem can be partially assimilated with Fowler's *Large Class* bad-smell.

**Strategy** The detection of god-classes is based on the three characteristics of these classes: they are expected to access a lot of data from "lightweight" classes (either directly or through accessor-methods), they are expected to be large and to have a lot of non-communicative behavior. Like we did before, we will first detect the classes that strongly depend on the data of "lightweight" classes. After that, we will filter the first list of suspects, by eliminating all the small and cohesive classes.

**Metrics**

1. **Access To Foreign Data (ATFD)** [Mar01]
   - *Definition:* ATFD represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods. The higher the ATFD value for a class, the higher the probability that the class is or is about to become a god-class.
   - *Implementation Details:* Inner classes and superclasses are not counted.
2. **Weighted Method Count (WMC)** [CK94]
   - *Definition:* WMC 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).

- *Implementation Details:* We recommend the use of McCabe's cyclomatic number [McC76] for the quantification of method complexity.

3. **Tight Class Cohesion (TCC)** [BK95]

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

**Rule**

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

### 5.3.2 Shotgun Surgery

**Motivation** This design flaw means that a change in a class implies many (small) changes to a lot of different classes [FBB+99]. Changes that are dispersed over many places are hard to find. Thus, *Shotgun Surgery* influences the maintainability of the code. In other words, this design flaw tackles the issue of strong implementation coupling and it regards not only the coupling *strength* but also the coupling *dispersion.*

**Strategy** We want to find those classes in which a change would significantly affect many other places in the source-code by their instability. In detecting the classes most affected by this smell, we will consider not only the coupling *strength*, but also the coupling *dispersion.* We identified three potential "victims" of changes in a class:

- methods that directly access an attribute that has changed
- methods that call a method which has changed its signature
- methods that override a method which has changed its signature

We define the *coupling strength* by the number of places (methods) that must be changed if we operate a modification in a changing class (server-class). Thus, a measure that counts the strength of a coupling is given by the cardinality of the set of methods that access an attribute and of a set of methods that call or override a method of the changing class. An alternative would be to "weight" the coupling on the client methods: for each client-method we count how many distinct members of the server-class have been used.

The other factor that influences the impact of a change is the *dispersion* of the resulting changes, i.e. the number of classes (subsystems) that must be inspected because of a change in a given class.

Based on all the considerations above, the detection technique is now very simple to describe: first, we look for the classes with a *strong* change impact, and

from these we retain for further manual inspection the classes that have a high *dispersion* of the change.

### Metrics

1. **Changing Methods (CM)**
   - *Definition:* CM is defined as the number of distinct methods in the system that would be potentially affected by changes operated in the measured class.
   - *Implementation Details:* The methods potentially affected are all those that access an attribute and/or call a method and/or redefine a method of the given class.
2. **Weighted Changing Methods (WCM)**
   - *Definition:* For each method that would be counted by the CM metric, we attach a *"weight"*. The weight is defined as the number of distinct members from the server-class that are referenced in that method.
   - *Implementation Details:* WCM is computed as the sum of the weights of all the methods affected by changes.
3. **Changing Classes (CC)**
   - *Definition:* The CC metric is defined as the number of client-classes where the changes must be operated as the result of a change in the server-class.

### Rule

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

### Remarks

1. If two classes affect by changes $N$ methods, the class that has its changes spread over more classes is worse than the one that has them all in one class. This is the reason why we used a metric (CC) that measures the dispersion of the changes.
2. We could also imagine this strategy with a different perspective on the CC metric: thus, if all the changes are in one or two classes we should consider the possibility of moving a member to that class.
3. An alternative way to define the *Shotgun Surgery* detection strategy would be to count not only the methods that are coupled to the changing class, but instead to sum up the *strength* of each couple. For this purpose we defined the WCM metric that might substitute the CM metric in the detection strategy.

4. The metrics described before are dealing with probabilities. If we would deal with this problem at the (server) method level (i.e. server-method – Number Of Client-Methods ÷ Number Of Client-Classes) we would find those methods that are really affecting the rest of the system. Thus, the *Shotgun Surgery* detection strategy can be also defined at the method level.

5. This design flaw is apparently related to the issue of *crosscutting concerns*, especially emphasized in Aspect-Oriented Programming (AOP). The relatedness consists in the impact of change: both *Shotgun Surgery* and crosscutting concern deal with the tendency of changes to affect multiple modules. The main distinction between the two is the level of abstraction: the issues related to crosscutting concerns are more abstract than the ones described by the *Shotgun Surgery* flaw. Crosscutting concerns, represent a "particular goal, concept, or area of interest" [Lad02] (e.g. logging, error-handling) and consequently the trouble caused by them is due to a change of *policy* or *approach* related to the given concern. In *Shotgun Surgery* we are dealing with the impact of changes in a *class*.

## 5.4 Flaws of Subsystem Structure

Large-scale software applications require some kind of high level organization. Although the class is a very useful unit for organizing small applications, it is too finely grained to be used as an organizational unit for large applications. As a consequence we need a design entity "larger" than a class in order to organize *large* applications (see Section 4.2). Thus, a *subsystem*[4] is used to represent a logical grouping of declarations that can be imported into other programs [Mar97b].

Martin [Mar97b] [Mar97c] and Lakos [Lak96] have defined principles and design rules that should guide the organization of software at the subsystem level. Moreover, as we have discussed in Section 3.4.2, Martin already made the first steps towards the quantification of the principles of subsystem organization. Therefore, the detection strategies presented in this section should be regarded as a continuation of these efforts.

We identified two classes of flaws that may affect the subsystem structure: *cohesion* and *dependency* (coupling) flaws. The commonality of cohesion flaws is that they tend to group classes in an inadequate manner, which leads either to a few oversized subsystems or to an inflation of very small subsystems. As Martin points out [Mar97b], there is a tension between three package cohesion principles: while the *"Release-Reuse Equivalence Principle"* (REP) and the *"Common Reuse Principle"* (CRP) are focused on facilitating the *reuse*, the *"Common Clo-*

[4]We use here the term "subsystem" considering it synonymous with the Java's *"package"* or Booch's [Boo94] *"class category"* concept.

*sure Principle"* (CCP) is oriented on making the *maintenance* more convenient. We defined detection strategies for the following cohesion flaws:

- **God Package** (Section 5.4.1). By strictly applying the CCP principle, the packages tend to become very large and non-cohesive. Another symptom of this flaw is the large number of clients of the package (i.e. classes from other packages) that use excessively this package.
- **Inflation of Atomic Packages.** By applying the REP and CRP principles, the packages tend to become very small, almost "atomic". In contrast to the *"God-Package"* flaw, clients of packages affected by this flaw tend to use classes from a large number of other packages.
- **Misplaced Class.** In "God Packages" it happens often that a class needs the classes from other packages more than those from its own package. In this case, especially if the class uses mainly another package, we could try moving the class to that package.

At the architectural level, applications can be described as a *"network of interrelated packages"* [Mar00]. Therefore the *dependency flaws* that may occur at this level have a major influence on the quality of the design; these flaws represent violations of the coupling criteria (Section 2.2.1). We identified and quantified the following set of dependency flaws:

- **Unstable Dependency**. In conformity with the *"Stable Dependency Principle"* (SDP) a package should depend only on packages that are more stable than itself [Mar97c]. The violation of this principle has a strong negative impact on the changeability of the software. We could formulate this violation as follows: a package that depends on another one which is more unstable than itself. This design flaw is detectable using the stability metrics defined by Martin.
- **Wide Subsystem Interface** (Section 5.4.2). The flaw refers to the situation where this interface is very wide, which causes a very tight coupling between the package and the rest of the world and this is undesirable. An alternative name for this flaw is *Lack of Façade* and a usual correction method is to apply the Façade design pattern [GHJV94].

We will illustrate the detection technique for subsystem flaws by presenting next the detection strategies for both cohesion and dependency flaws.

### 5.4.1 God Package

**Motivation** By strictly applying the *Common Closure Principle* (CCP) [Mar00], the packages tend to become very large and non-cohesive. Another symptom of this flaw is the large number of clients of the package (i.e. classes from other packages) that use excessively this package. The consequence is that clients of this package must check out everything in that package even if that change does not affect them in any way.

**Strategy** We want to find those packages which are very large and which are heavily used by classes from outside the package under scrutiny. We believe that if the classes that use the package are spread among many packages there will be an increase of the probability that the used package is a "god-package". A final aspect that we want to capture is the cohesion of the package, as "god-packages" are expected to have a lot of classes that don't communicate a lot with each other.

### Metrics

1. **Package Size (PS)**
   - *Definition:* PS is the number of classes that are defined in the measured package.
   - *Implementation Details:* Inner classes are not counted.
2. **Number Of Client Classes (NOCC)**
   - *Definition:* NOCC represents the number of classes from other packages that use the measured package.
   - *Implementation Details:* A class uses a package if it calls methods, accesses attributes or extends a class defined in that package.
3. **Number Of Client Packages (NOCP)**
   - *Definition:* NOCP is the number of other packages that use the measured package.
   - *Implementation Details:* A package uses another package if at least one if its classes is using that package.
4. **Package Cohesion (PC)**
   - *Definition:* PC is defined as the relative number of class pairs from a package between which a dependency exists.
   - *Implementation Details:* A class A depends on another class B, if class A calls methods and/or accesses attributes from class B and/or extends class B. In computing PC, we count these class pairs and divide their number by the maximum number of class pairs. For a package with $n$ classes, the maximum number of class pairs is:

     $$\frac{n(n-1)}{2}$$

     Inner classes should not be counted.

### Rule

```
GodPackage := ((PS, HigherThan(20)) and (PS, TopValues(25%)) and
              (NOCC, HigherThan(20)) and (NOCP, HigherThan(3))
```

### 5.4.2 Wide Subsystem Interface

**Motivation** This flaw is inspired by the motivation for the *Façade* design pattern [GHJV94]. Similar to classes, we can also speak about the *interface of a subsystem*. This interface consists of the classes that are accessed or, more precisely, are accessible from outside the package. The flaw refers to the situation where this interface is very wide, this causing a very tight coupling between the package and the rest of the world, which is undesirable. In most of the cases the interface can be reduced by introducing a Façade class. If this does not help it might be a sign that the "God-Package" flaw has also crept in. In this case the possibility of relocating some of the classes should be cqnsidered.

**Strategy** We have based our definition for this pathological case on the fact that most probably each subsystem is encapsulated in a separate package. Thus, we can talk about a subsystem interface, the same way as we do for classes. The interface of a subsystem is comprised of the set of classes from that package which are accessible from outside the package. This pathological case appears when we are dealing with packages with a rich interface, with lots of classes from the package being accessed from outside, which leads to a very tight coupling between this package and the rest of the system. In other words, this pathological case manifests itself through a package with the following characteristics (Fig. 5.1):

- a large number of classes from the package are accessed from outside the package;
- a high ratio of classes from the package are used

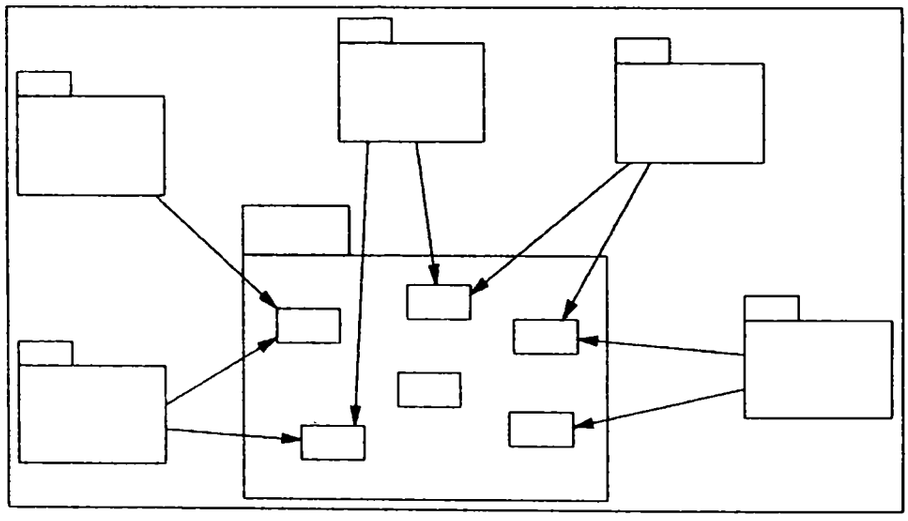


Figure 5.1: The pathological case for *Façade*

**Metrics**

1. **Package Interface Size (PIS)**
   - *Definition:* PIS is the number of classes of a package that are used from outside the package.
   - *Implementation Details:* A class uses a package if it calls methods, accesses attributes or extends a class defined in that package.
2. **Package Usage Ratio (PUR)**
   - *Definition:* The PUR metric is defined as the relative number of classes from the measured package that are used from outside that package.
   - *Implementation Details:* The number of used classes will be divided by the total number of classes in the package, from which we exclude the inner classes, i.e. PS[5]. Thus

$$PUR = \frac{PIS}{PS}$$

**Rule**

```
WideSubystemInterface := (PIS, HigherThan(10)) and
                         (PUR, HigherThan(0.75))
```

## 5.5 Lack of Patterns

Motivated by our intention to quantify and detect common design flaws encountered in object-oriented systems, we developed an approach based on the following idea: because patterns are elegant solutions to common design problems, we want to capture the deficiencies of the design that would result from ignoring the pattern-based solutions in contexts were these would be highly recommendable. The assumption that missing patterns generates poor design is also in conformity with the authors of the best-known book on design patterns [GHJV94] that discuss in the "motivation" section for several patterns the design deficiencies that would result from ignoring the pattern solution. Surprisingly or not, this makes the "classical" catalogue of design patterns a source for identifying and defining new types of design flaws. For these flaws, assuming that we succeeded in detecting them, there is already a solution for the improvement of the design, i.e. the one offered by the pattern.

In [Rus01] we defined in detail the steps of this approach and evaluated its feasibility. We started from the entire catalogue of design patterns [GHJV94] and first we eliminated those patterns for which it was impossible to define a detection strategy based strictly on static analysis information. After this step

[5]for the definition of PS see Section 5.4.1

we kept 10 out 23 patterns. These detection strategies where then applied on several case-studies and based on the experimental results we kept only those that proved to be efficient. We ended up with six patterns for which the aforementioned approach proved to be feasible. These are: *Bridge*, *Façade*, *State*, *Strategy*, *Visitor* and *Singleton*. Note that among the six patterns there are creational as well as structural and behavioral patterns.

The *"Lack of Patterns"* category of design flaws is on a higher level of abstraction than the previous one. In terms of detection strategies this distinction can be expressed as follows: in the case of a detection strategy defined to capture a "*missing pattern*", knowing only the design entity returned by the strategy is not enough to locate a flaw as this always affects a design fragment. Thus, the strategy must explicitly specify the "role" of the detected entity in the flaw and the rule by which the other entities belonging to the flaws fragment can be identified. This is well exemplified on the *"Lack of Bridge"* strategy (Section 5.5.1).

In conclusion, we may say that the approach helped us identify and capture a set of new design flaws. There are still several causes that partially limit its applicability: first, it is not possible to express the flawed structure by metrics for every design pattern; second, symptoms are often overlapping for different patterns; third, during the experiments we encountered many false-positive results.

From the six patterns for which we defined detection strategies we selected two for a detailed analysis: one that detects a missing structural pattern, i.e. *Bridge* (Section 5.5.1) and one detecting a missing behavioral pattern, i.e. *Strategy* (Section 5.5.2).

### 5.5.1 Lack of Bridge

**Motivation** The *Bridge* [GHJV94] design pattern is used to decouple an abstraction from its implementation, so that the two may vary independently. When we have several implementations for a certain abstraction, the usual solution is to use inheritance: an abstract class defines the interface and concrete subclasses provide the implementations. However, this solution is inflexible because the abstraction and its implementations are statically linked. Thus, the following problems may appear:

- if we have a lot of abstractions, organized in a rich hierarchy, by adding the implementation subclasses this hierarchy becomes very complex;
- because of the static link between abstraction and implementation, modifying the abstraction implies the modification of all its implementations;
- when we add a new abstraction, it is necessary to write all the subclasses that provide the implementations for this abstraction.

**Strategy** From the description above it is obvious that the pathological case associated with this design pattern appears as an "inflation" of classes: a rich class hierarchy, in which the subclasses are divided in two categories:

1. *implementation subclasses* — these are concrete subclasses which implement the interface of their superclass;
2. *specialization subclasses* — these are abstract subclasses, representing abstract specializations of their superclass.

Each specialization subclass will have its own subset of implementation subclasses. The situation described above may appear in two different ways which represent the two pathological cases for this design pattern:

1. a "deep" class hierarchy, with implementation classes for every abstract subclass, probably also for the hierarchy superclass (Fig. 5.2);

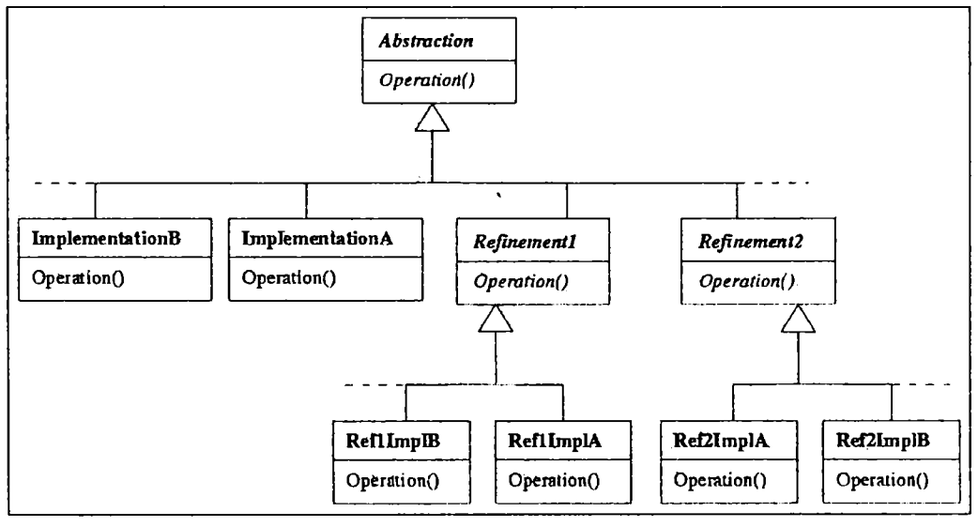



Figure 5.2: The first pathological case for *Bridge*

2. a degenerate version of the previous case: a "shallow and wide" class hierarchy, where most (or all) of the concrete implementation subclasses are derived from the same base class; in this case we have a hierarchy with a lot of classes on the first level (Fig. 5.3).

### Metrics

1. **Number Of Descendants (NOD)**
   - *Definition:* NOD is the number of classes directly or indirectly derived from the measured class.

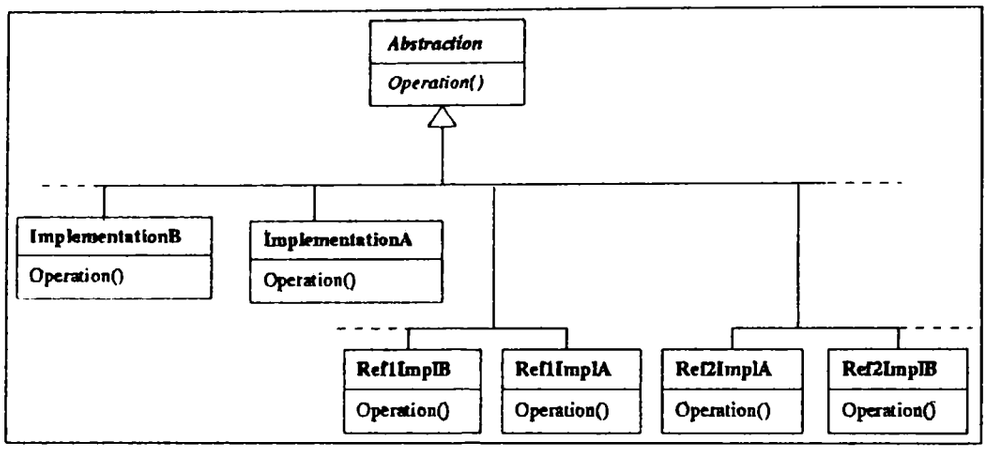



Figure 5.3: The second pathological case for *Bridge*

2. **Height of Inheritance Tree (HIT)**

   - *Definition:* HIT is computed as the maximum number of levels of derivation (inheritance) for the given class.
   - *Implementation Details:* HIT is complementary to the *Depth of Inheritance Tree* (DIT) metric. While DIT measures the longest path *upwards* for a class in the inheritance-tree, HIT measures the longest path *downwards* in the inheritance-tree.

3. **Leaves Ratio (LR)**

   - *Definition:* LR is defined as the number of "leaf-classes" divided by the total number of descendant classes (NOD) in the hierarchy tree rooted by the measured class.
   - *Implementation Details:* A "leaf-class" is a derived class that has no further classes derived from it.

4. **Child Ratio (CR)**

   - *Definition:* If we consider the definitions of *Number Of Children* (NOC) [CK94] and *Number Of Descendants* (NOD), then CR is defined as:

$$CR = \frac{NOC}{NOD}$$

5. **Number Of Public Methods (NPubM)**

   - *Definition:* NPubM is the number of public methods defined in the measured class.
   - *Implementation Details:* Constructors and the destructor of the class should not be counted.

**Rule**

```
LackOfBridge := LackOfBridge-deep or LackOfBridge-shallow
LackOfBridge-deep :=  (NOD, HigherThan(8)) and
                      ((HIT,HigherThan(1)) and (LR,HigherThan(0.66)))
                      and (NPubM, HigherThan(3))
LackOfBridge-shallow := (NOD, HigherThan(6)) and
                        ((CR,HigherThan(0.75)) and (HIT,HigherThan(0)))
                        and(NPubM, HigherThan(3))
```

**Remarks**

- We chose the threshold values for the NOD and the LR metrics, considering the minimal situation of 2 implementation classes and 2 specialization classes. In the case of a deep hierarchy this leads to a minimum number of 8 classes (see Fig. 5.2) and in the case of a shallow one the lowest NOD values is 6 (see Fig. 5.2). In a similar manner we computed the LR and CR threshold values.

- The threshold values mentioned before are calculated for the situation where the pathological cases are "pure". Yet, sometimes you will encounter degenerate situations, where for example there is only one implementation subclass, or where the two pathological cases are mixed. If we want to capture these cases too, we have to *reduce* the thresholds for the different metrics (NOD, LR, CR) because these class hierarchies are not so rich but are a potential danger if the design evolves.

- The role of the NPubM is the following: without it, the strategy would capture also the roots of some class hierarchies that probably are part of the *Strategy* design-pattern. Because we wanted to avoid that situation, we introduced this metric filtering out those root-classes that defined a small interface that is then implemented by a number of subclasses.

### 5.5.2 Lack of Strategy

**Motivation** When we need several algorithms to solve a certain problem, the *Strategy* [GHJV94] design pattern offers the solution of defining a family of algorithms; these algorithms are encapsulated and organized in a class hierarchy, so they can be easily interchanged.

There are two types of poor design that can be imagined related to the ignorance of the pattern solution; they differ by the method used to implement the algorithm family. Thus, we identified the following two pathological cases:

1. All the algorithms are encapsulated in one big class (fig. 5.4); this class will be very complex, with some methods having large conditional structures, namely those methods in which the algorithms are implemented.

Also, there is the possibility that these methods will access algorithm-specific attributes, which are probably not accessed anywhere else. Thus the cohesion of these classes is relatively low due to the algorithm-specific data.

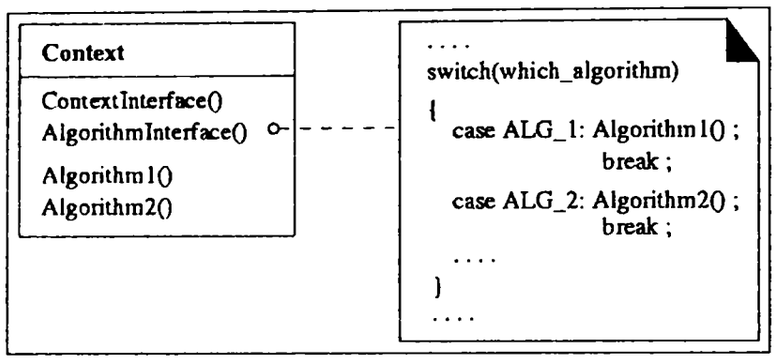



Figure 5.4: The first pathological case for *Strategy*

2. Another possibility is the appearance of a class hierarchy (probably shallow and wide), where each subclass offers a different implementation of the algorithm (fig. 5.5). It is very likely that the only difference between the base class and its subclasses the overriding of the methods that implement the algorithm.

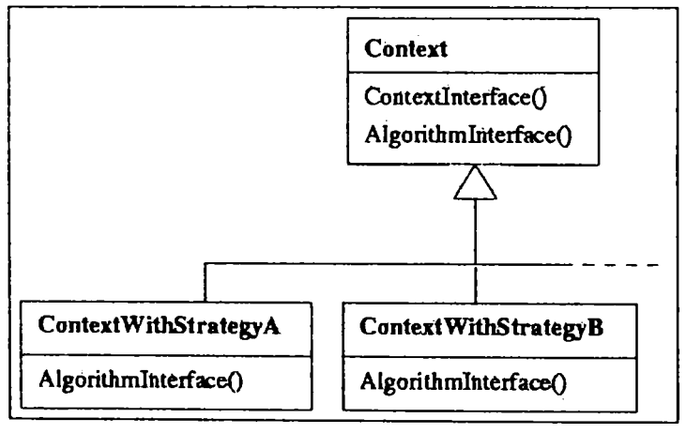



Figure 5.5: The second pathological case for *Strategy*

**Strategy** For the first pathological case described before we are looking for classes with a high level of complexity (methods with large conditional structures) and low cohesion.

The second pathological case implies class hierarchies with a special structure:

the subclasses have a low number of public methods of their own, most of them overridden (to implement the algorithm).

### Metrics

1. **Weighted Method Count (WMC)** [CK94]

   - *Definition:* WMC is the sum of the statical complexities of all methods in a class. If this complexity is considered unitary, WMC measures in fact the number of methods (NOM).
   - *Implementation Details:* We recommend the use of McCabe's cyclomatic number [McC76] for the quantification of method complexity.

2. **Tight Class Cohesion (TCC)** [BK95]

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

3. **Number Of Public Methods (NPubM)** [LK94]

   - *Definition:* NPubM is the number of public methods defined in the measured class.
   - *Implementation Details:* Constructors and the destructor of the class should not be counted.

4. **Number Of Methods (NOM)** [LK94]

   - *Definition:* NOM is the number of methods defined (not inherited) in the measured class.
   - *Implementation Details:* Constructors and the destructor of the class should not be counted.

5. **Average Number of Overridden Methods (ANOM)**

   - *Definition:* The ANOM metric is defined as the average number of methods from the measured class that are redefined in the classes directly derived from it.
   - *Implementation Details:* ANOM is computed by counting the number of methods that were overridden in each class derived from the base class that we are measuring, by summing them up and dividing the sum by the number of derived classes.

**Rule**

```
LackOfStrategy := LackOfStrategy-OneClass or
                  LackOfStrategy-ClassHierarchy

LackOfStrategy-OneClass :=
      (WMC, HigherThan(20) and (WMC, TopValues(25%))) and
      (NOM, HigherThan(20)) or (TCC, LowerThan(33%))

LackOfStrategy-ClassHierarchy :=
      (ANOM, HigherThan(1.0)) and
      (NPubM, HigherThan(3))
```

**Remarks**

1. We introduced the NOM metric in the `OneClass` sub-strategy in order to be sure to also "catch" the classes that might have a higher cohesion than the threshold, but which are very large.

2. We introduced the NPubM metric in the `ClassHierarchy` sub-strategy, in order to eliminate the cases of class hierarchies where the base class defines a small interface that is then implemented by all the subclasses (e.g. in case of applying the *Strategy* pattern the algorithms hierarchy tree would have such a shape).

## 5.6 Conclusive Remarks

Looking back at the entire chapter there are several remarks and conclusions that summarize the entire discussion:

- Detection strategies help us encapsulate the detection process for a given design flaw. In this context the name of the strategy is essential because it allows the engineer to reason in the abstract terms of *what* must be detected and not in the chasm of *how* it is detected.

- We managed to define strategies for all levels of abstraction from the lowest one (the method level), up to level of micro-architectures (i.e. clusters of classes).

- The catalogue is far from being complete. The intention was to illustrate how detection strategies can be applied for finding design flaws. Further strategies should be defined to extend the detectable design problems following the methodology defined in the previous chapter (see Section 4.4) and applied over and over again through this chapter.

- The fact that we identified just structural design problems is not an intrinsic limitation of the detection strategy concept. The limitation is due to the fact that the metrics used through this work are exclusively design

metrics that rely on the static design structure. In other words the upper limit of the definable detection strategies is set by the limits of the meta-model. Thus, the approach is applicable also to other types of design flaws (e.g. code duplication) as long as the meta-model contains the necessary information to detect the flaws.

- Using a medical metaphor, detection strategies are means to detect a design *"disease"* based on a correlation of *"symptoms"*. Each symptom is captured by a metric, more precisely by the interpretation model for a given metric. The composition mechanism (Section 4.3.4) of the detection strategy allows a flexible composition of the symptoms in a quantifiable expression of a design "disease", which is a design flaw.

- Sometimes symptoms overlap and therefore we encountered in this chapter metrics that appeared in several strategies.

- In most cases a design is not affected by a singular design flaw. Therefore, in order to obtain a real picture of a design's quality these detection strategies should not be used in isolation. In order to give their highest benefit, detection strategies need a coherent framework that would relate them to quality. In other words they must be used in the context of a quality model. In the coming chapter we will define a new type of quality model, that improves the existing approaches by taking advantage of the detection strategies defined in this chapter.

# Chapter 6

# Factor-Strategy Quality Models

We measure because we want to evaluate and eventually improve the quality of the design. We want to bridge the gap between how quality is perceived and how it is assessed through measurements at the design level. This chapter proposes a new approach to quality models called – *the Factor-Strategy Quality Model.* More precisely, this approach is based on detection strategies that quantify rules directly related to the quality of design, as the ones we described in Chapter 5

This new quality model may be regarded in a way as the end-result of all the mechanisms and techniques defined in the previous two chapters and the accomplishment of the initially stated goal of this dissertation.

The chapter starts with a presentation of the Factor-Criteria-Metric (FCM) approach including a discussion on the limitations of this well-known quality model. After that, we introduce the concepts of the *Factor-Strategy approach* to quality models as an alternative that eliminates the main drawbacks of the FCM paradigm. It improves the FCM approach by expressing and evaluating quality factors in terms of measurable expressions of good-style design heuristics and rules. In this context, we also introduce the mechanisms for evaluating a FS quality model. The last part of this chapter deals with the issues related to the construction of FS quality models and exemplifies them by proposing a quality model for maintainability. The chapter ends with an evaluative conclusion.

## 6.1 Quality Models

Without an assessment of product quality, speed of production is meaningless. This observation has led software engineers to develop *models of quality* whose measurements can be combined with those of productivity. In this section we discuss the decompositional approach to quality models, known as the *Factor-*

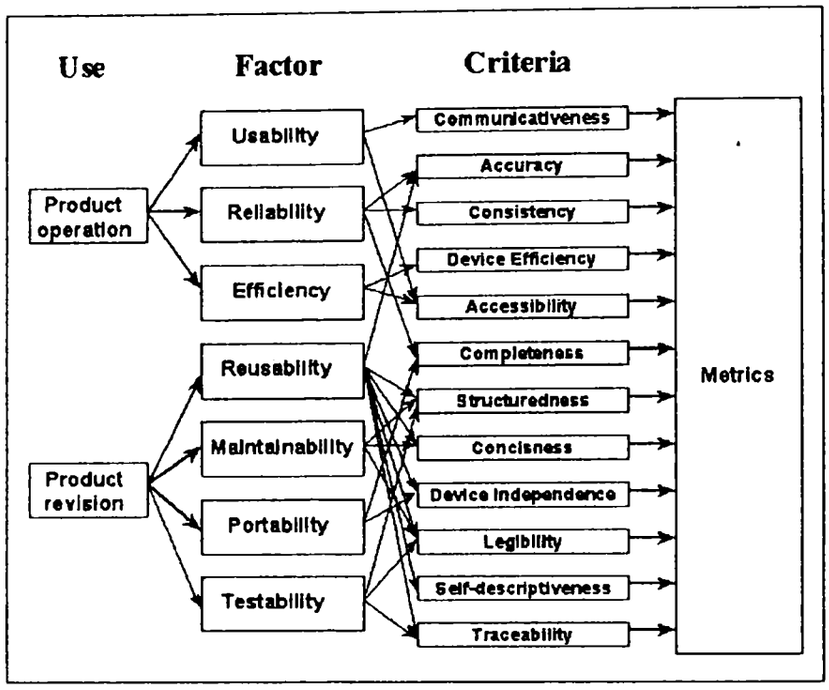



Figure 6.1: A FCM Quality Model [FP97]

*Criteria-Metric*, and point out the limitations in applicability of this approach together with a very recent contribution that addresses these limitations.

### 6.1.1 Factor-Criteria-Metric Models

One of the most frequently used quality models is the decompositional approach used both by McCall [MRW77] and Boehm [BBK78], commonly known as the *Factor-Criteria-Metric* quality model (FCM). FCM models are usually constructed in a tree-like fashion, similar to the one depicted in Figure 6.1. The upper branches hold important high-level *quality factors* related to software products, such as reliability and maintainability, which we would like to quantify. Each quality factor is composed of lower-level *criteria*, such as structuredness and conciseness. These criteria are easier to understand and measure than the factors themselves, thus actual metrics are proposed for them. The tree describes the relationships between factors and criteria, so we can measure the factors in terms of the dependent criteria measures. This notion of divide-and-conquer has been implemented as a standard approach to measuring software quality [ISO91]. Thus, there are three elements that build a FCM model:

1. *quality factors* – i.e. high-level key attributes that express the quality as perceived by the user.

2. *quality criteria* – i.e. lower-level attributes that are closer to the developers' perspective. Each quality factor is expressed in terms of quality criteria based on the belief that there is an isomorphic relation between the quality of software as perceived by the user and the quality of code and design; in other words the internal characteristics of the code have an effect on external quality attributes.

3. *quality metrics* – i.e. a set of code-based metrics that each quality criteria is mapped into.

### 6.1.2 Example: A FCM Model for Maintainability

In order to illustrate the concepts of the FCM quality model, we will now present as an example a quality model inspired from a widely-spread commercial tool for software auditing and quality assurance [Tel00]. The quality model is depicted in Figure 6.2. We have chosen this example not only because it illustrates the FCM concept, but also because its evaluation mechanisms are similar to the ones we used in this work.

The quality goal of this model is *maintainability* that is decomposed in four quality factors, in conformity with the decomposition found in [ISO91]. These factors are: analysability, changeability, stability and testability. As we are going to revisit this example later in this chapter (Section 6.4.3) we will present next the definitions of the four maintainability factors, as they appear in [ISO91]:

**Definition 6.1 (Analysability)** *Attributes of software that bear on the effort needed for diagnosis of deficiencies or the causes of failures, or for identification of the parts to be modified.*

**Definition 6.2 (Changeability)** *Attributes of software that bear the effort needed for modification, fault removal or for environmental change.*

**Definition 6.3 (Stability)** *Attributes of software that bear on the risk of unexpected effect of modifications.*

**Definition 6.4 (Testability)** *Attributes of software that bear on the effort needed for validating the modified software.*

After decomposing maintainability in the four quality criteria, the model associates a set of metrics with each criterion, in order to provide a quantifiable expression for it. Based on the values computed for each of these metrics, a score is calculated for each quality criteria by using a mathematical formula. In other words, a quality criterion can be regarded as a higher-level indirect measurement, computed from the metrics associated with it.

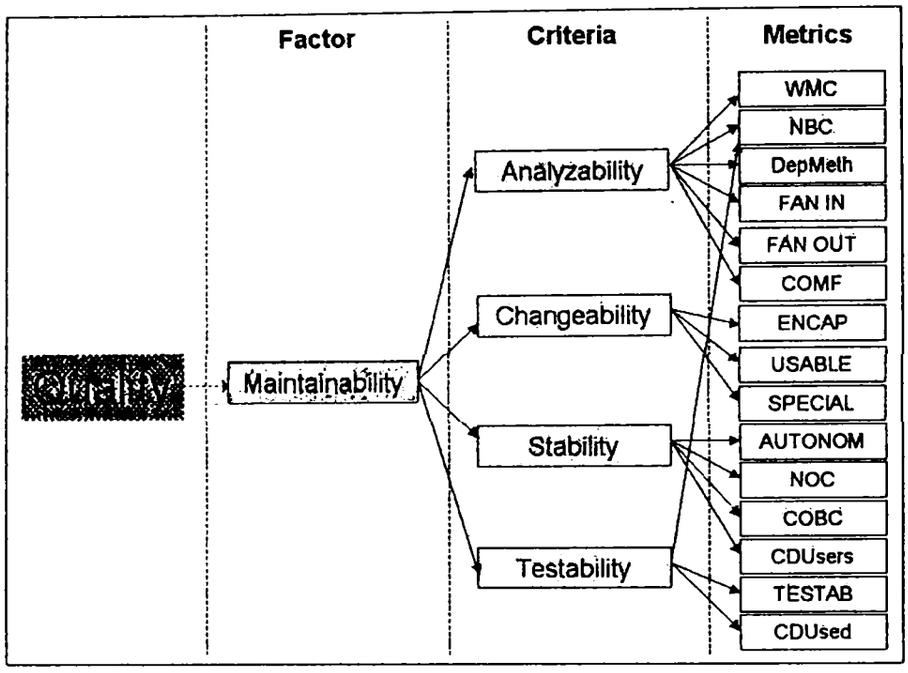



Figure 6.2: The TELELOGIC quality model at the class level. In this model there is only one quality factor, as the focus of the tool lies on the assessment of maintainability.

For example, *testability* for a class is considered to depend on the unit test effort for the class's methods, the number of times this test effort should be repeated (number of inherited classes) and the number of used classes. Consequently, testability is computed (see Figure 6.2) as the mathematical sum between two direct metrics – i.e. *Number of Base Classes* (NBC) and *Number of Directly Used Classes* (CDUsed) – and an indirect one (TESTAB) which is defined as the sum of: the number of the class's methods, the number of outside attribute uses, and the number of calls to functions defined outside the class.

The *Telelogic* quality model is evaluated in a stepwise manner, as follows:

1. For each criterion a value is computed from the values of the metrics associated with that criterion based on a mathematical formula.

2. The value for each criterion is classified in one of the four quality categories: EXCELLENT, GOOD, FAIR or POOR. Each of these categories has a numeric *quality scores* associated with it.

3. For each factor again a value is computed based on the quality score of the criteria associated with that factor. From the evaluation point of view, the

association between a factor and its criteria is defined by the mathematical formula used to compute the value of that quality factor.

4. The final step is to map the computed values of the quality factors to a quality category, as it was done for the criteria.

The results of evaluating the quality of a system using this quality model are summarized in a report. One of the main functions of the report is to reveal the design entities that cause a reduced quality. Although the report offers indeed a list of classes and methods that reduce quality, it is very hard to understand the *real causes* of the poor quality as the only thing that is provided is a vector of 10-15 metrics values, out of which some are marked to exceed a threshold specified by the quality model. This brings us to one of the major limitations of the FCM quality models: their incapacity to detect the real design causes of poor quality. In the following section we are going to discuss in more detail the limitations of the FCM quality models.

### 6.1.3 Limitations of FCM Quality Models

Although this approach is cited throughout the whole software engineering literature and is implemented in several commercial CASE tools it has two main drawbacks that limit its usability. These limitations are discussed through the rest of this section.

#### Obscure mapping of quality criteria into metrics

The first striking question that came to my mind while analyzing different FCM models was: *how are the quality criteria mapped into metrics?* The answer to this question is essential because it affects the usability and reliability of the whole model.

The question can be rephrased as follows: How is the quality of design quantified? As we have pointed out earlier (see Section 2.2), in order to assess the quality of a design we need a set of *criteria* (e.g. low coupling, high cohesion, manageable complexity, good structuredness) that must be eventually mapped to the design principles and rules that characterize the design paradigm (e.g. object-orientation) that was used while developing the software.

Unfortunately, in the FCM approach this explicit mapping between quality criteria on one hand and rules and principles of design and coding on the other hand is missing. To be more precise, the mapping is not missing, but is implicitly and obscurely contained in the mapping between the quality criteria and the quality metrics. Thus, the answer to the previous question is: quality criteria are mapped into metrics based on a set of rules and practices of good-design. But this mapping is "hidden" behind the arrows that link the quality criteria to the metrics, making it in most of the cases impossible to trace back.

This observation reveals a first important drawback of the FCM quality models: if the model is a fixed one, it will be *hard to understand*, because we can only guess what are the rules and principles that dictated the mapping. In case of a "user-defined" model the model is *hard to define*, because when we mentally model quality we reason in terms of *explicit* design rules and heuristics, keeping the quality criteria *implicitly* contained in the rules. Yet, when it comes to defining a FCM quality model, things are totally on the other way around: quality criteria are explicit while design rules are implicit.

### Poor capacity to map quality problems to causes

The second drawback is related to the efficiency and relevance of interpreting the results from a FCM quality model. After all, we build quality models because we seek an answer to the following questions:

- *Diagnosis.* What are the design and implementation problems that affect the quality of my software?
- *Location.* Where are those problems located in the code?
- *Treatment.* What do I need to change, at the design level, to improve the quality of my software?

When analyzing a software system using a FCM model – and let us consider the ideal case in which the model produces a perfect association between quality factors and the design structure – we get the quality status for the different factors that we are interested in (e.g. the maintainability is quite poor, while portability stays at a fair level). We are also able to identify a set of programming entities (e.g. classes or methods) that are supposed to be responsible for a poor status of a certain quality factor. We can find these suspect entities based on the values and the interpretation rules (e.g. threshold values) of the different quality metrics used in the model. Thus, it appears that FCM solves both the diagnosis and the location issues.

But as soon as we arrive at the question concerning the treatment, we reached the limits of the FCM model, because the model doesn't help us find the *real causes* of the quality flaws detected by it. A treatment can only be imagined when knowing the *disease* not only a set of *symptoms*. The cause of this is the fact that abnormal metric values – even if the metrics are provided with a proper interpretation model – are just *symptoms* of a design or implementation *disease* and not the disease itself.

FCM models confront their user with a large set of measurement results, which in most of the cases are not fully understood, in spite of the interpretation guidelines given for the metrics. The multitude of abnormal values is rather puzzling for the engineer than is assisting him to find the proper code transformation that would improve the design. The fact that the metrics are linked to a quality criterion is not solving the problem, much the same way as knowing that high

fever is related to an infection is not helping to find a cure.

As already mentioned before, a software engineer developing a new system or redesigning a legacy system is reasoning in terms of good-design rules and practices, not in terms of abnormal numbers. Therefore, any improvement of design or implementation, that would eventually bring an overall improvement of quality, is expected to be motivated by the guidelines of good-design. But design principles and rules are only implicit in a FCM model, so that when using the model we are faced with abnormal numbers instead of explicitly dealing with the deviations from good-design practices.

Summarizing, the second major drawback of the FCM model is that it does not offer any relevant feedback concerning the causes of quality flaws, thus hindering an efficient transformation of the code that would improve quality. Consequently, the developer is provided only with the problem and he or she must still empirically find the real cause and eventually look for a way to eliminate the design flaw and improve the design.

### 6.1.4 Hybrid Approach to Quality Models

In [SB01] and [SBL01] the authors criticize the FCM models and propose an improvement to such predictive quality models. The authors emphasize that the use of precise threshold values and their interpretation in the absence of formal models, as well as the crudeness of the derived rules which can only serve to build naïve models are the two diseases of current approaches for building predictive models. They propose a novel approach by building fuzzy decision processes that combine both software metrics and heuristic knowledge [FN00] from the field. The authors claim that this hybrid approach would improve efficiency of quality prediction and provide a more comprehensive explanation of the relationship that exists between the observed data and the predicted software quality characteristics.

A critical view on the paper reveals that while the fuzzification of threshold values seams applicable and well founded [Kos96], the second part containing decomposition of heuristic knowledge is far from being traceable. In addition to that, no case studies are provided in the paper so that the practicability of the approach is not yet proved. In conclusion, we believe that this approach will become useful in the future especially on the side of a proper parametrization of the interpretation models of the metrics. Yet it does not offer a comprehensive approach for an improved quality model.

## 6.2 Factor-Strategy Quality Models

The drawbacks of the FCM quality models are in essence due to the fact that quality is mapped to a large set of measurements that are only implicitly related to the rules and principles of good-design. These drawbacks can be eliminated if

the quality of code and design would be *explicitly* assessed in terms of deviations from a set of quantifiable good-design principles and heuristics. Therefore, based on the *detection strategy* concept introduced in Chapter 4 and its "instances" defined in Chapter 5, – i.e. strategies for problem detection – we propose a new type of quality models, called **Factor-Strategy**. This approach is intended to improve the FCM paradigm with respect to the two major drawbacks discussed earlier.

**Definition 6.5 (Factor-Strategy Quality Model (FS))** *A model used for the assessment of software quality that decomposes quality in a set of high-level quality factors and maps each factor to a set of strategies that quantify deviations from rules of good design is called a* Factor-Strategy Quality Model.

In the previous definition we distinguish the following components of a Factor-Strategy quality model:

1. **Quality Factors** have precisely the same meaning as in the FCM models, i.e. high-level attributes used as an expression of quality as perceived by the user.

2. **Strategies** are the means used to detect and quantify the flaws at the design and implementation level that negatively rebound upon quality factors. A strategy encapsulates a metrics-based mechanism for finding deviations from a particular rule of good design. The strategy in a Factor-Strategy quality model is in fact an instance of the *detection strategy* concept (see Section 4.3) defined for identification and localization of design flaws (see Chapter 5).

In Figure 6.3 we illustrated the concept of a Factor-Strategy model. At first sight it becomes obvious that FS models still use a decompositional approach, but after decomposing quality in factors, these factors are not anymore associated directly with a bunch of numbers, which proved to be of a low relevance for an engineer. Instead, quality factors are now expressed and evaluated in terms of detection strategies, which are the quantified expressions of the good-style design rules for the object-oriented paradigm.

Therefore we may state in more abstract terms that *in a Factor-Strategy model, quality is expressed in terms of principles, rules and guidelines of a programming paradigm*. The set of detection strategies defined in the context of a FS quality model encapsulate therefore the *knowledge-box of good design* for the given paradigm. The larger the knowledge-box, the more accurate the quality assessment is. In our case the detection strategies are defined for the object-oriented paradigm, and thus in the right side of Figure 6.3 we depicted a sample of a *knowledge-box* of object-oriented design.

The knowledge-box as such is indispensable for any quality model. Although not visible at first sight, it is also present in the FCM models. The knowledge-box is not obvious in the FCM approach because of its *implicit character*. The entire

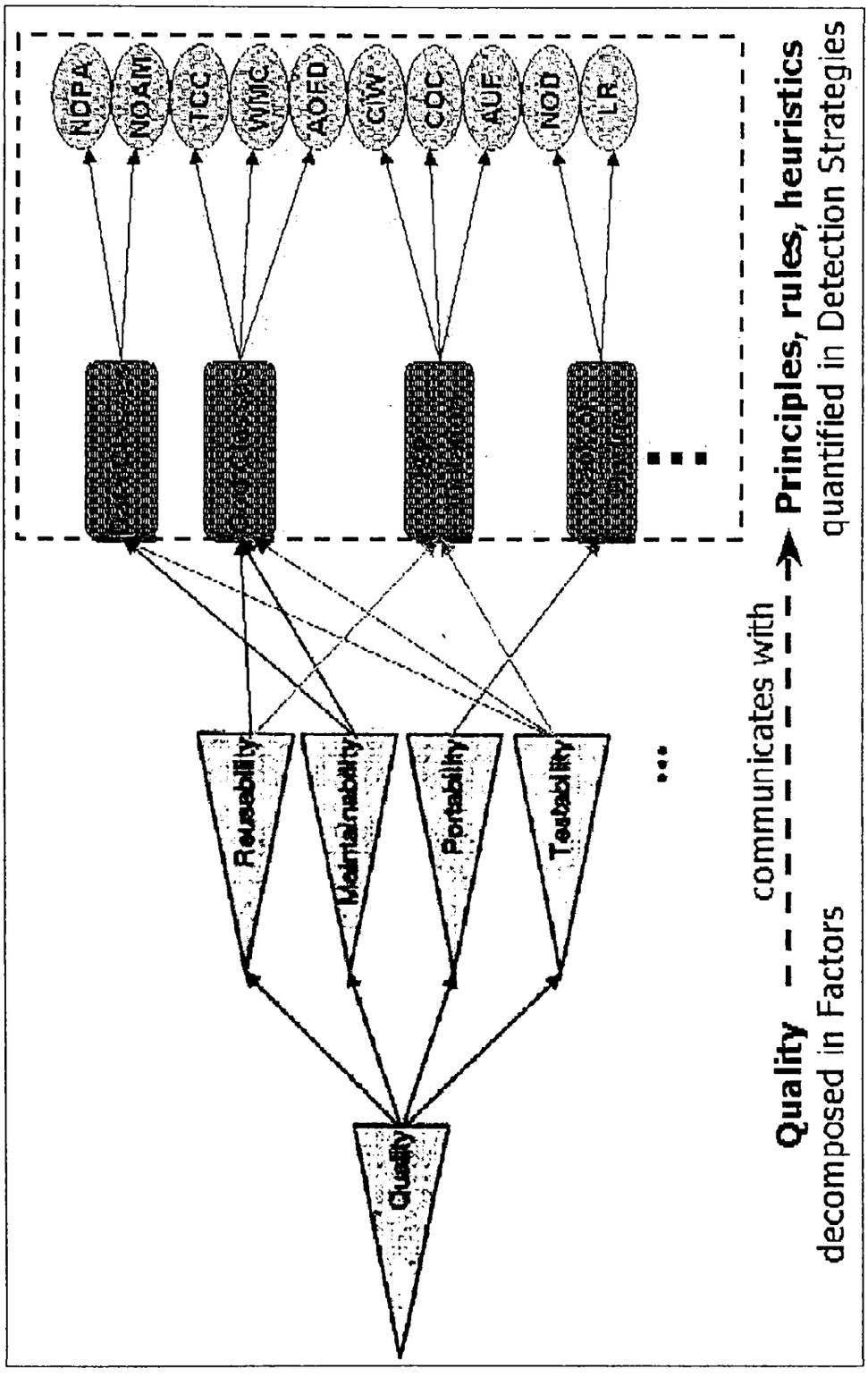



Figure 6.3: A Factor-Strategy quality model. The picture does not represent a complete model. It only illustrates the new concept of a model in which quality is assessed explicitly in terms of deviations from a set of design principles, rules and heuristics, which are quantified in terms of Detection Strategies

knowledge about the programming paradigm is captured in the association of criteria with metrics. Using a metaphor, we might say that the knowledge-box is behind the arrows that link criteria with metrics. Of course, the metrics are also paradigm-specific, but the design and programming principles that the metrics reflect are implicit. They may be inferred from the construction of the model, but they are not explicit. Thus, the novelty of the FS approach resides in the *explicitness* of the principles and rules that support the quality of a system. This explicitness is supported by the use of detection strategies as quantifications of these rules and principles.

## 6.3 Evaluation of Factor-Strategy Models

After defining the new Factor-Strategy quality model, we move to describe the evaluation mechanisms of the model. At the same time we take a closer look at the relationships between the key elements of this new type of quality model.

The assessment of quality for a software system based on a quality model involves two main aspects: the *quantification and ranking* of the quality status of the analyzed system and the *identification of the causes* for low quality. Next, we are going to discuss these two aspects.

### 6.3.1 Quantification and Ranking of Quality Factors

This aspect requires mechanisms for computing a high-level metric for each quality factor (*quantification*) and for mapping this metric value on a scale that would allow us to draw qualitative conclusions on the system with respect to that quality factor (*ranking*).

#### Ranking of Strategy Results

The ranking of a detection strategy with respect to a quality factor is the process by which the results of a detection strategy are mapped to a number that expresses the quality of the analyzed system from the point of view of the design rule quantified by that particular detection strategy. This process is graphically depicted in Figure 6.4.

In this chapter we express a detection strategy ($\sigma$) as a vector of pairs, where each pair consists of a metric ($\mu_i$) that belongs to strategy $\sigma$ and a parameter ($\pi_i$) used in filtering the data associated with that metric,

$$\sigma(< \mu_1, \pi_1 >, \ldots, < \mu_n, \pi_n >)$$

When applying a detection strategy on a software system, we get two results:

1. The set of suspect entities detected by the strategy: $\{\varphi_k^{(\sigma)} | k = \overline{1, n^{(\sigma)}}\}$ where $n^{(\sigma)}$ represents the number of suspect entities for strategy $\sigma$.

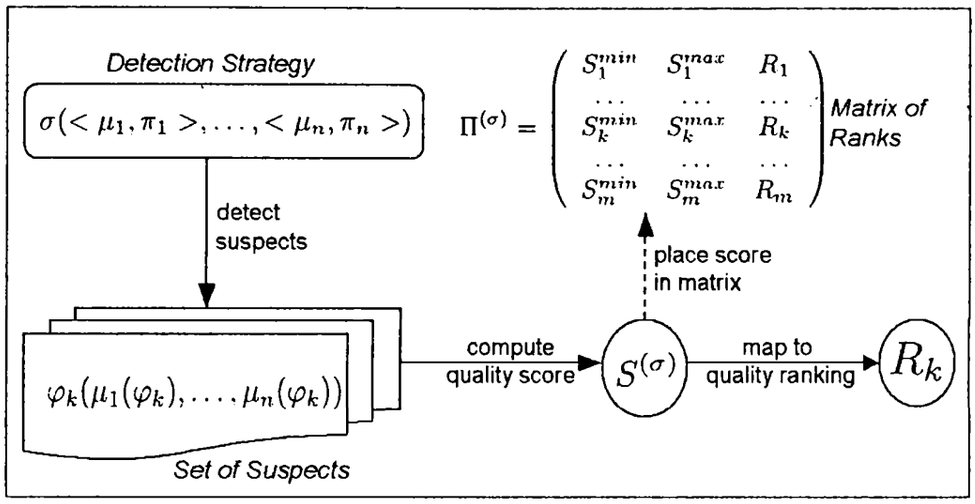



Figure 6.4: The process of computing a quality score from the results of running a detection strategy over the analyzed system and eventually transforming the score into a quality ranking using a matrix of ranks

2. For each suspect design entity ($\varphi_k^{(\sigma)}$), a vector of values representing the measurement results for each of the metrics involved in the detection strategy: $\{\mu_j(\varphi_k^{(\sigma)}) | j = \overline{1, m^{(\sigma)}}\}$, where $m^{(\sigma)}$ represents the number of metrics involved in detection strategy $\sigma$.

Based on this information a *quality score*($S^{(\sigma)}$) for the detection strategy can be computed. This strategy score can be regarded as a higher-level metric; a metric applied to the detection strategy. For the computation of the quality score, at least the following computation methods can be used:

- *Number of Suspects:*

$$S^{(\sigma)} = n^{(\sigma)}$$

  based on the interpretation that the more suspect design entities are reported by a detection strategy the lower the quality of the design is.

- *Relative Number of Suspects:*

$$S^{(\sigma)} = \frac{n^{(\sigma)}}{n}$$

  where $n$ represents the total number of measured entities.
  This computation method has the advantage over the previous method that it takes into account the size of the analyzed system, which is an advantage if we want to compare quality among systems of different sizes.

- *Weighted Number of Suspects:*

$$S^{(\sigma)} = \sum_{i=1}^{n^{(\sigma)}} w_i^{(\sigma)}$$

where $w_i^{(\sigma)}$ is the weight of each suspect entity. The weight could be computed for each suspect entity based on the vector of measurement results. This would allow a differentiation, for example, between a class that was considered a suspect because of a single measurement value slightly beyond the threshold limits and another class where all the measurement values in the metrics vector are far beyond the admissible limits.

After computing the raw quality score, the score is mapped to a *ranked score* that is a normalized value taken from a *matrix of ranks*. In the matrix of ranks, the first and the second elements designate an interval of (raw) scores that will be mapped to the ranked score, given in the last (third) column. Thus, having a score $S^{(\sigma)}$ and a rank matrix:

$$\Pi^{(\sigma)} = \begin{pmatrix} S_1^{min} & S_1^{max} & R_1 \\ \dots & \dots & \dots \\ S_k^{min} & S_k^{max} & R_k \\ \dots & \dots & \dots \\ S_m^{min} & S_m^{max} & R_m \end{pmatrix}$$

the retrieved ranked score will be $R_k$, where $k$ is the first line in the ranking matrix that satisfies the condition:

$$S^{(\sigma)} \in [S_k^{min}, S_k^{max}]$$

### Ranking of Quality Factors

In a Factor-Strategy quality model, a quality factor ($\Phi$) is modelled as a vector of pairs, where each pair consists of a detection-strategy ($\sigma_k$) and its ranked-score ($R_k$) that was computed in the previous step. Thus, a quality factor looks as follows

$$\Phi(< \sigma_1, R_1 >, \dots, < \sigma_k, R_k >)$$

This vector is the starting point for the computation of a (ranked) quality score for that quality factor. The ranking process is similar to the one previously presented for ranking strategy results. The process is depicted in Figure 6.5.

Compared to the ranking of strategies, there are only two minor differences: the *formula* for the computing of the raw quality score ($S^{(\Phi)}$) and the *content* of the matrix of ranks ($\Pi^{\Phi}$). This time the formula for computing ($S^{(\Phi)}$) is based on the set of ranked scores ($R_1 \dots R_k$), each score having a weight that represents the relevance of that particular detection strategy in the context of the quantified quality factor. In other words, a ranked score $R_1$ may be weighted differently for different quality factors. If we want to treat equally all the strategies that are associated with a factor, $S^{(\Phi)}$ is simply the *average* of the ranked score.

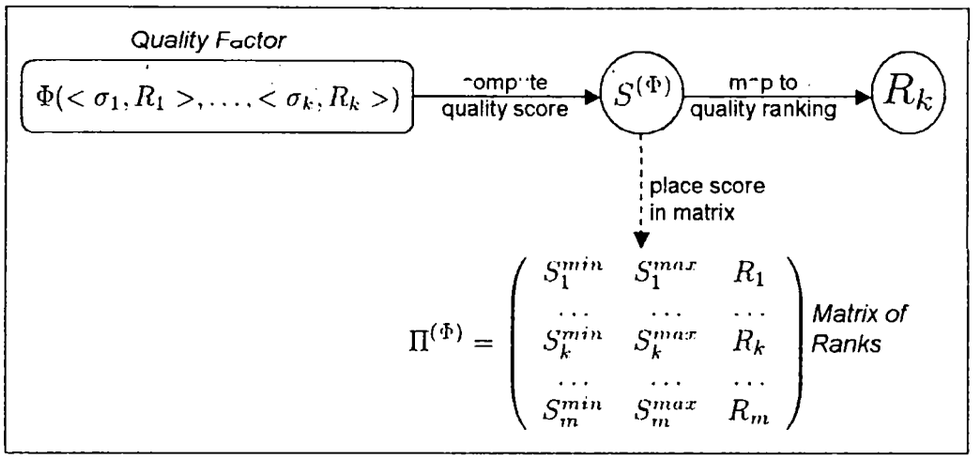



Figure 6.5: The mechanism for computing a score for a quality factor from the rankings of the associated strategies.

### 6.3.2 Identification of Causes for Poor Quality

While the previous section deals with the issue of *how good the design is*, this section is concerned with the following consequent issue: if the quality of the system is not satisfactory, *which parts of the design need improvement* and *what is the concrete improvement that is needed* in order to increase the overall quality of the system.

As we already discussed in Section 6.1.3, the Factor-Criteria-Metric can also identify design fragments that are potential causes for poor quality based on the abnormal values for the different metrics associated with a quality factor. However the major limitation of the FCM approach is its intrinsic incapacity to support the engineer in finding the information that would allow him or her to find an adequate remedy. In other words, we identify the flawed entities, but we don't know the exact flaw; we just know a set of symptoms.

Factor-Strategy quality models solve this problem by associating quality characteristics with detection strategies, instead of metrics. Therefore, because detection strategies quantify violations of concrete design rules and guidelines, the design entities reported as flawed by a strategy can be associated with the concrete design disease captured by that strategy.

As we have seen earlier, one result of a strategy is the set of suspect design entities:

$$\{\varphi_k^{(\sigma)} | k = \overline{1, n^{(\sigma)}}\}$$

Based on these sets of suspects for each detection strategy we can make two statements concerning the identification of design flaws and flawed entities in a Factor-Strategy model:

- The set of design flaws that affect the quality of a system modelled using a FS model, is given by the subset of detection strategies associated with the model, that return a non-empty set of suspects.
- The set of design entities that are responsible for a poor quality in a system modelled using a FS model, is given by the reunion of the suspect sets of all detection strategies associated with the model.

**Remark.** A special case occurs when a design entity (e.g. a class) appears in the suspect sets of more than a single detection strategy. In this case, the conclusion is that the suspect entity is indeed flawed, but the flaw is not uniquely classifiable using the given set of detection strategies. In this case, the manual inspection phase is especially important, in order to identify the real flaw and eventually refine the quality model.

## 6.4 Building Factor-Strategy Quality Models

After we have defined the Factor-Strategy quality model and its evaluation mechanisms, now we move our attention to the issue of building concrete FS models. In this context, first we summarize the potential quality goals and then propose a methodology for constructing the model. The final part of the section will provide a concrete example of a FS quality model for maintainability. This quality model will then be used in Chapter 7 (see Section 7.4) to evaluate the usability and efficiency of this new approach to quality models.

### 6.4.1 Quality Goals

The quality goal is the driving force for defining and using a particular quality model. McCall [MRW77] defined a *triangle of quality* (see Figure 6.6), as he identified three different perspectives on the quality of software products: operation, revision and transition. For each of these three views he identifies a set of high-level, desirable properties – i.e. a set of quality goals.

The scope of this thesis is limited to the assessment of product quality as reflected by the quality of code and design, using static analysis techniques. Because of that, although the Factor-Strategy principle can be used for each and every quality goal, we cannot build FS models for all quality goals using the detection strategies defined in Chapter 5, as these are entirely based on design metrics. As a consequence, we want to identify a subset of quality goals for which FS models are entirely definable, based on the approach found in this dissertation. We place the entire discussion in the context of reengineering as we show that this approach will help us identify the quality goals.

This work emerged from the context of reengineering research [BBC+99b], [Cas98], [BBC+99a]. A re-engineering operation will be driven by one of several goals (Figure 6.7); but above all of these goals, there is one abstract goal, i.e. to

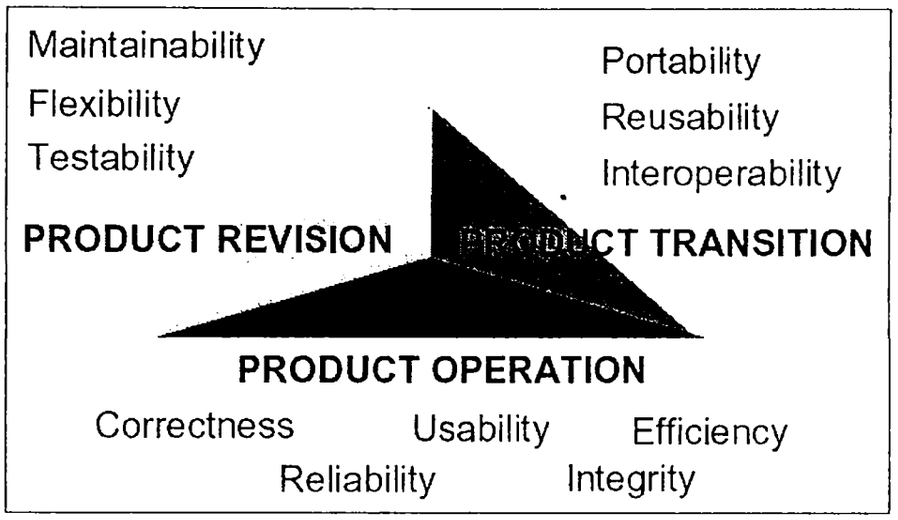



Figure 6.6: McCall's Triangle of Quality

improve different aspects of quality, by renovating the structure of an existing system. Most of the works on software analysis have their roots in this context [CC90] [CC92] [MWT94]. Furthermore, in the recent years it was observed [Ciu01] that reengineering activities are not only relevant for *legacy systems*, but they can be efficiently applied in each iterative and incremental development process, especially in lightweight development processes [Bec99] where the design structure is often subject to change.

Therefore, the quality goals that are assessable based on design information can be identified by using the reengineering perspective and analyzing the major *reengineering goals*. The association of quality goals with reengineering goals is both relevant and accurate.

An existing system can evolve in one of two ways depending on its business-values: *enhancement*, if the business-value of the entire system is high, or *reuse* if parts of the system are of a particularly high value.

**Enhancement.** If the business-value of the system as a whole is high then the high-level goal is to enhance the system. Going into more detail we identify three types of enhancement: *functional*, *non-functional* and *environmental*.

Improving a system from the *functional* point of view requires that the structure of the existing system would support further functional extensions. In terms of quality goals this means that the system must be **flexible**.

The *non-functional* enhancements include those changes that improve the quality of the system without the addition of any functional extensions. Improve-

| High-Level Goal | Type of Enhancement | Reengineering Goal | Quality Goals |
|---|---|---|---|
| Enhancement | Functional | Make the system easier to extend (to add new functionality) | **Flexibility** |
| | Non-Functional | Reduce the effort of migrating to a new technology | **Portability** |
| | | Make the system easier to maintain | **Maintainability Testability** |
| | Environmental | Reduce the effort of bringing the system to a new platform | **Portability** |
| Reuse | Component / Framework Extraction | Increase the modularity of the system | **Reusability Flexibility** |

Figure 6.7: A synthesis of the main quality goals directly related to the structure of a software system

ments that fall into this category are those that increase either the **maintainability** of the system or its **testability**. Another non-functional enhancement is to increase the performance of the system by migrating it to a new technology. This is reflected again at the structural level by an increase in **flexibility**.

A third type of enhancements is concerned with the *environment* of the system. The quality goal that relates to this issue is to increase **portability**, i.e. to reduce the effort of bringing the system to a new platform.

**Reuse.** Sometimes, the system as a whole has lost most of its business-values, yet some of its parts are still highly valuable. Those parts of the system should be extracted so as to be reused. Therefore, the high-level goal is to increase the **reusability** of the initial system. In order to do that, the modularity (see Section 2.2) of the system must be increased.

Using the reengineering perspective as a starting point we identified a subset of quality goals that are directly related to the structural (design and implementation) aspects of a software system. In Figure 6.7 we summarized the previous discussion in the form of a table. The central point of this table is the association of reengineering goals with quality attributes that are directly related with the structure of the software system. Taking now another look at McCall's triangle of quality (Figure 6.6) we observe that none of the quality goals related to the structure belongs to the *product operation* category; they all belong to the product *revision* or *transition* category. The explanation is simple: the quality goals related to product operation are mainly pointing to the *dynamic and semantic* aspects of the system.

Based on the previous observations we may state that the special target of FS quality models based on structural detection strategies is the revision and

transition of software systems.

### 6.4.2 Stepwise Construction of Factor-Strategy Models

The main issue in constructing any type of quality model is how to build the *association* between the higher and the lower levels of the model; for example, in building a Factor-Criteria-Metric model we would be concerned with how to associate the quality factors with the proper criterion, or how to choose the metrics for a given criterion. As the Factor-Strategy models are also based on a decompositional approach the association is still the relevant issue. Based on the previous considerations from this chapter, we identified two distinct aspects on this matter of association: a *semantical* and a *computational* aspect.

- The *semantical aspect* must sustain the validity of choosing a particular decomposition for a higher-level element into lower-level ones. In other words, it must explain the rationale behind the association; it must answer the question: **why** *and* **how** *do we choose a particular decomposition for a higher-level element of the model?*

- The *computational aspect* must tackle the issue of quantifying the association, i.e. it must answer the following question: *How is the quality score for the higher-level element to be computed from the quality scores of the lower-level elements associated with it?* Obviously, the *computational association* follows the *semantical* one. In other words, we must first define an association that "stands" semantically, and only then the focus must be set to finding the *association formula* that quantifies it. The association formula must reflect the "participation level" of each lower-level element within the higher-level aspect of quality.

In constructing a Factor-Strategy model there are two association that must be done: the decomposition of the quality *goal* in quality *factors* and the association between quality factors and detection strategies that can express and detect design flaws that affect the given quality factor. For each of these associations, next we will discuss both the semantical and the computational aspects.

#### Decomposition of the Quality Goal in Quality Factors

There are two possible approaches to address the semantical aspect of the association between a quality goal and a set of quality factors: we can either rely on a *predefined decomposition* found in the literature or go for a *user-defined one*. The former option has the advantage of a wider acceptance and validation, while the latter is more flexible and adaptable to the particular investigation needs. In this dissertation we decided to rely on an existing and widely accepted decomposition, i.e. the one found in the ISO9126 standard [ISO91]. For the general case we recommend using a hybrid solution: start from a predefined decomposition found in the literature that comes closest to your ideal model and then slightly customize it until it matches your perspective on quality.

This association is orthogonal to the programming paradigm used for the design and implementation of the system. The decompositions found in literature, in spite of many differences, keep the higher level of quality decomposition [1], abstract enough to make it independent of the development paradigm. As a consequence, in a FS model the decomposition of a quality goal in factors is not different in any aspect to that found in the FCM approach. Therefore, the computational aspect of this association does not raise additional discussions at the conceptual level.

#### Association of the Quality Factors with Detection Strategies

Detection strategies used in FS models capture deviations of a design from design rules and guidelines. The authors of such good-design rules implicitly or explicitly relate them to quality factors, or to abstract design principles (e.g. abstraction, modularity, simplicity) that can be easily mapped to quality factors (see Section 2.2). Therefore, the *semantical aspect* of the association between quality factors and detection strategies becomes self-evident in the FS approach. This is one of the main advantages of the FS models over the FCM approach, where the correspondent association is subject to severe drawbacks, as we have already pointed out in Section 6.1.3.

There are two actions related to the *computational aspect* of the association between factors and their set of strategies:

1. *Compute a quality score for each detection strategy.* In Section 6.3.1 we have defined two mechanisms for computing a quality score from the results of a detection strategy: first, a *formula* to compute the raw-score, $S^{(\sigma)}$, must be established mainly based on the number of suspects; second, a *matrix of ranks* based on which the raw-score $S^{(\sigma)}$ is transformed into a quality scores $R_k^{(\sigma)}$.

2. *Compute a quality score for the quality factor.* This score is also computed based on an algorithm (Section 6.3.1), that involves again two mechanisms: first, an *association formula* in which the operands are the quality scores ($R_k^{\sigma}$) computed for the strategies; second, a *matrix of ranks* that transforms the raw-score ($S^{(\Phi)}$) for the quality factor into a quality score $R_k^{(\Phi)}$.

### 6.4.3 A Factor-Strategy Model for Maintainability

We want to open this section with a disclaimer: the quality model that we are going to present below raises no claim of completeness. Moreover, we believe that a complete and universally acceptable quality model is impossible to define at least because of the following reasons:

[1] By intention, we didn't name this level the *factors level* because some authors refer to it as the criteria level. In order to avoid confusion in terminology we referred to it as the "higher-level of quality decomposition".

- There is no objective argument for adding or removing a component from the model [KW86].
- The "knowledge-box" used in the model – i.e. the detection strategies defined for the model – is limited and we see no possibility of claiming completeness in this aspect.

In the context of the previous disclaimer, why do we still propose this concrete model? The motivation is twofold:

1. First, we want to illustrate the steps and mechanisms involved in the construction of a FS quality model. Thus, we want to illustrate how metrics are encapsulated in detection strategies and how quality factors are associated with these strategies that quantify deviations from good design rules.
2. The second reason is that we want to illustrate how the drawbacks of the FCM approach are eliminated, by comparing the newly proposed model with the one defined in Section 6.1.2.

For describing FS quality models, during our research we defined a simple description language, called QMDL[2]. The language is inspired from the description language used in Telelogic [Tel00] to describe FCM quality models (see Section 6.1.2). The decision to define QMDL as a variant of a description language used in connection with FCM quality models was deliberate, as we believe that this would simplify the understanding of both the commonalities and the differences between the FCM and the FS approach.

Next, we will sketch out a step-by-step building of a quality model for maintainability. In this section we just illustrate the construction steps and mechanisms. The full annotated description of the proposed quality model for maintainability (in QMDL) is provided in Appendix A.

### Decomposition of Maintainability in Quality Factors

In conformity with the ISO-9126 standard [ISO91] maintainability is decomposed in the following four factors: analysability, changeability, stability and testability. Because we want to weight equally the four factors in the evaluation of maintainability, we will use the average value of the scores computed for each quality factor. Using (QMDL) the association formula for maintainability is expressed as follows:

```
Maintainability := avg( Changeability, Analysability,
                        Testability, Stability)
```

Obviously, any other mathematical formula might have been used depending on the special emphasis of the quality evaluation. For example, if the emphasis would have been on the *analysability* aspect of maintainability, the previous formula could have been replaced by an weighted average:

[2]QMDL is the abbreviation from Quality Models Description Language

```
Maintainability := (Changeability + 3*Analysability +
                    Testability + Stability) / 6
```

### Associating Quality Factors with Detection Strategies

We briefly illustrate the process of association between a factor and a set of strategies using the *Stability* factor, which is in this model an aspect of maintainability. The definition of stability was provided in Section 6.1.2.

Analyzing the definitions of the design flaws that are detectable using the current set of detection strategies [3] we have selected the strategies associated with six of these flaws, i.e. those that affect directly stability. These are: `ShotgunSurgery`, `GodClasses`, `DataClasses`, `GodMethod`, `LackOfState` and `LackOfSingleton`.

After the semantical association between the *Stability* factor and the six strategies, we focus on the computational aspect, using the following sequence of steps:

**Step 1:** *Choose a formula for computing the raw-score $S^{(\sigma)}$ for each strategy.* In our case we have used for all the strategies the simplest formula, i.e. the raw score is the *number of suspects* (see Section 6.3.1) detected by that strategy.

**Step 2:** *Choose an adequate matrix of ranks for each strategy.* Using this matrix we place the raw-score in the context of quality, i.e. the matrix of ranks tells us how good or bad a raw-score is with respect to the quality factor. We used three levels of tolerance in ranking the raw-scores for the strategies, and consequently we defined three matrices: a severe one (`SevereScoring`), a permissive one (`TolerantScoring`) and one in between the two (`MediumScoring`). For the design flaws that in our view had the highest impact on stability we applied the `SevereScoring` matrix.

For example, we believe the design flaws that affect stability in the highest degree are `ShotgunSurgery` (see Section 5.3.2) and `GodClasses` (see Section 5.3.1). Therefore, we computed their quality score ($R^{(\sigma)}$) using the `SevereScoring` matrix, which is defined as follows:

```
SevereScoring {
  0   0   10,    /* EXCELLENT  */
  1   1   9,     /* VERY GOOD  */
  2   4   7,     /* GOOD  */
  5   7   5,     /* ACCEPTABLE */
  8   +oo 3      /* POOR */
},
```

For the rest of the design flaws we used the `MediumScoring` matrix.

[3] Please refer to Appendix B for definitions and further details on the complete set of detection strategies defined during this research.

**Step 3:** *Define a formula for computing the raw score for the factor.* Because we intended to weight equally the six strategies when computing a score for stability, we used the *average* function (avg). Throughout the model we applied the same function for computing the raw-scores for quality factors.

**Step 4:** *Choose the matrix of ranks for computing the quality score for the factor.* The raw-score computed during the previous step must also be placed in a matrix of ranks in order to retrieve a normalized quality score. As we reached the last step, we can now "reveal" the quantified definition of *Stability* in QMDL:

```
Stability := avg(ShotgunSurgery(SevereScoring),
                 GodClasses(SevereScoring),
                 DataClasses(MediumScoring),
                 GodMethod(MediumScoring),
                 LackOfState(MediumScoring),
                 LackOfSingleton(MediumScoring))
    {
          9   10  10,  /* EXCELLENT  */
          7   9   8,   /* GOOD  */
          5   7   6,   /* ACCEPTABLE */
          0   5   4    /* POOR */
    };
```

## 6.5 Conclusion

In this chapter we presented a new approach to quality models, the *Factor-Strategy* models. By defining this approach we put together all the elements defined in the previous chapters in a coherent framework, that allowed us to prove the thesis stated at the beginning, i.e. that the gap between qualitative and quantitative statements, concerning object-oriented software design can be bridged. While detection strategies represent the higher-level mechanism for measurement interpretation, the Factor-Strategy quality model provides a goal-driven approach for applying the detection strategies for quality assessment.

The Factor-Strategy approach presented in this chapter has two major improvements over the classical approaches:

1. First, the *construction* of the quality model is far easier because the quality of the design is naturally and explicitly linked to the principles and good-style rules of object-oriented design. Our approach is in contrast with the classical Factor-Criteria-Metric approach, where in spite of the decomposition of external quality factors into measurable criteria, quality is eventually linked to metrics in a way that is far less intuitive and natural.

2. Second, the *interpretation* of the strategy-driven quality model occurs at a higher abstraction level, i.e. the level of design principles, and therefore it leads to a direct identification of the real causes of quality flaws, as they are reflected in flaws at the design level. As we pointed out earlier, in this new approach quality is expressed and evaluated in terms of an explicit knowledge-box of object-oriented design.

Besides the improvement of the quality assessment process, the detection strategies used in the context of a Factor-Strategy quality model proved to have a further applicability, at the conceptual level: for the first time a quality factor could be described in a concrete and sharp manner with respect to a given programming paradigm. This is achieved by describing the quality factors in terms of the detection strategies that capture design problems that affect the quality factor, within the given paradigm.

# Chapter 7

# Evaluation

Through the last three chapters we described a complete approach for bridging the gap between qualitative statements and measurements. We started by showing that one of the main causes for this gap is the abstraction level of measurements interpretation, which is too low to capture the design aspects that are relevant for evaluating and improving the quality of object-oriented design. Consequently, we introduced *detection strategies* as a mechanism for defining and using higher-level interpretation rules, in order to increase the relevance of measurement results. Additionally, we provided a methodology for building concrete detection strategies, according to a particular investigation goal. The next step was to define proper detection strategies for the identification and location of structural design problems. For this purpose we defined a coherent suite of detection strategies that address a set of well-known design problems informally described in the literature. But we observed that in order to give their highest benefit, the detection strategies that build the suite should be correlated and explicitly related to quality, in an adequate quality framework. Therefore, the final step of the approach was to define a new type of quality model, based on detection strategies, that improves conceptually and methodologically previously existing approaches. By taking this final step we reached the initial goal, i.e. to bridge the gap between qualitative and quantitative statements concerning object-oriented design.

In this chapter we will evaluate the practical applicability of the entire approach. For this purpose we designed a real-world experiment, implemented an adequate toolkit and applied the previously described methods and techniques.

The chapter begins with a presentation of the entire experimental setup (Section 7.1). In this context, we define a set of evaluation criteria, describe the industrial case-study to be analyzed, design the experiment and defend its relevance for the approach described by this work. The second part of this chapter describes the toolkit that was developed in order to support the approach (Section 7.2). We emphasize the necessity of the toolkit in order to ensure the

scalability of the methods and techniques defined by this thesis. The third part of the chapter presents and discusses the results of the experiment. It is organized in two sections: first, we evaluate the usability of defining and applying detection strategies for the concrete investigation goal defined in Chapter 5, i.e. the detection of design problems in object-oriented systems (Section 7.3); second, we validate the accuracy and efficiency of the new *Factor-Strategy* quality model defined in Chapter 6.

## 7.1 The Experimental Setup

In this section we describe the evaluation approach. First, we define a set of criteria that represents the evaluation goal. Next we design the experiment and defend its relevance to the approach to be evaluated. Finally, we describe the industrial case-study used as the basis for our evaluation.

### 7.1.1 Evaluation Goals and Criteria

Before describing the evaluation approach we want to define the goal for the entire evaluation process, by identifying the criteria that are relevant for this experiment. Thus, *the goal of this experiment is to validate the approach by evaluating the* **scalability** *and the* **accuracy** *of its key mechanisms.* The approach defined by this dissertation contains two key mechanisms:

- **Detection Strategy** – is the higher-level interpretation mechanism used for quantifying informal design-related rules related to a given investigation goal. The methodological applicability of this mechanism has been already proved in Chapter 5, where we successfully defined more than 15 concrete detection strategies for a particular investigation goal, i.e. the quantification and detection of several design flaws, placed on different granularity and abstraction levels. Therefore, in this chapter (Section 7.3), we accomplish the validation of the detection strategy mechanism, by also evaluating its *practical* usability. In order to do this, we evaluate the ability of the before-mentioned suite of detection strategies to correctly identify and localize design problems in a "real-world" object-oriented system.

- **Factor-Strategy Quality Model** – is the new type of quality model, defined in Chapter 6. in which every quality attribute is associated with a set of quantified expression of rules of "good design". We claimed that this new type of quality model would support not only the assessment of quality at the design level, but also its improvement, by identifying and locating the real quality problems of the system. Therefore, in this chapter (Section 7.4) we want to evaluate the validity of the previous statement, by analyzing the results of applying the FS model for maintainability (Section 6.4.3) on a large-scale legacy system.

In the following, we briefly explain the meaning of the two evaluation criteria, i.e. scalability and accuracy, in the context of this chapter.

### Scalability

An approach is scalable if it continues to be efficiently usable when applied to large-scale systems. This work was mainly focused on the identification of design problems that affect the quality of an object-oriented system. Design problems in a system with a size up to a few hundred LOC can be easily found by reading through the code; if the analyzed project is in the size of a few thousand LOC, design flaws can still be localized, using a minimal tool support of general use (e.g. using facilities of an IDE). But when we come to large-scale industrial systems in the range of hundreds or millions of LOC, and most object-oriented software systems are in this range, rudimentary approaches don't scale up. Therefore, our approach must address the issue of problem detection in *large-scale* systems, in order to be useful. In conclusion, scalability is extremely important.

Analyzing the two aforementioned mechanisms of our approach, it is obvious that the scalability of the entire approach depends on the scalability of *detection strategies*. For a detection strategy that identifies design problems, there are two aspects related to scalability:

1. A detection strategy is scalable if the number of suspect entities detected by the strategy is much smaller than the total number of design entities of that type that exists in the analyzed project. This kind of scalability is influenced by the *definition* of the strategy, more precisely to the proper parametrization of the detection strategy (see Section 4.4).

2. A detection strategy is scalable if the execution time needed to retrieve the list of suspects stays in generally acceptable ranges. This aspect of scalability is identified with the scalability of tool support (Section 7.2).

### Accuracy

An approach is accurate if the obtained results are in conformance with the right or agreed results. This definition may be applied to the concrete context of problem detection and quality assurance as follows:

- *Accuracy of Detection Strategies.* The accuracy of the strategies defined for the detection of design flaws, presents two distinct aspects:

    1. A detection strategy is accurate if it exclusively and correctly localizes flawed design entities or fragments. From this point of view the key questions are: How many design flaws were overseen (*false negatives*)? How many false problems did it report (*false positives*)?

    2. A detection strategy is accurate if the identified suspect entities are indeed affected by the respective design flaw.

- *Accuracy of the Factor-Strategy quality model.* The accuracy of FS models is given by its ability to precisely reflect the quality level of the analyzed

system with respect to the specified quality goal. Additionally, a FS model is accurate if it is able to correctly identify and localize structural problems that reduce the quality level of the system.

### 7.1.2 Evaluation Approach

For the evaluation, we are going to use two successive versions of a large-scale, industrial object-oriented system (Section 7.1.3), for which we know that the second version is a reengineered and enhanced version of the first system. We also know that the design was substantially improved in order to increase its maintainability.

The evaluation approach is based on two basic assumptions:

1. **Assumption 1**: All major design problems that troubled the developers in the first version, were eliminated during the reengineering process, and consequently will not be found anymore in the second version. Thus, we do not assume that all the design problems have been eliminated, as we know that the reengineering operation was driven by a set of priorities – in this case related to the improvement of maintenance. But we assume that most relevant problems have been deal with.

2. **Assumption 2**: The level of maintainability for the reengineered version of the system is higher (better) than the one of the initial version of the system, as the reengineering goal was to increase maintainability.

#### Relevance of the Approach

The experimental approach that we are going to use is relevant for the evaluation of the approach defined by this dissertation because of a number of reasons, enumerated in the following:

1. Analyzing a large-scale industrial case-study gives us the opportunity to evaluate the *scalability* of the approach.

2. By analyzing two successive versions of a system which are also part of a "before-and-after reengineering" scenario, allows us to set up an evaluation methodology that assesses the *accuracy* of the detection strategies. Thus, based on **Assumption1** we can automatically identify the *false positive* suspects, i.e. entities that were erroneously reported as flaws by a given detection strategy. This evaluation methodology is described in detail in Section 7.3.

3. Taking advantage of the supplementary information that the goal of the original reengineering process was to improve maintainability, we can evaluate the *accuracy* and *relevance* of the information provided by the *Factor-Strategy* quality model for maintainability, defined in Section 6.4.3. In conclusion, based on **Assumption2** we can define a simple method to

evaluate the FS quality model. This method is discussed in detail in Section 7.4.

### 7.1.3 The Case-Study

As already mentioned, we are going to a make a comparative analysis based on two consecutive versions of a real-world system. The analyzed software is a large-sized business application related to computer-aided route planning, which covers everything from identification of ideal city routes to planning travels through Europe[1]. The application is developed in C++. The application is composed of a number of hierarchically organized subsystems.

The size characteristics of the first version of the system are summarized in Table 7.1. Through the rest of this chapter, we will designate this version of the system as SV1.[2]. SV1 was developed between 1997-1998 by designers and

| KLOC / KBytes | Packages | Classes | Methods |
|---|---|---|---|
| 93 / 3.000 | 18 | 152 | 1284 |

Table 7.1: Size of the first version of the system (SV1)

programmers quite familiar to object-orientation. This case-study is considered a legacy system not because of its age, but because one year after the first version was launched, the developers requested a reengineering of the system. This proves that there were some design weaknesses that needed to be detected.

As it can be observed in Table 7.2, the second version of the system significantly increased in size as more functionality has been added, but for us the most interesting aspect is the comparison of the two systems from the reengineering point of view.

| KLOC / KBytes | Packages | Classes | Methods |
|---|---|---|---|
| 115.6 / 5.300 | 29 | 387 | 3446 |

Table 7.2: Size of the second version of the system (SV2)

[1]The system that constitutes our case-study is subject to a non-disclosure agreement, that is why we cannot offer more information about it.

[2]standing for *System Version 1*. Similarly SV2 stands for System Version 2.

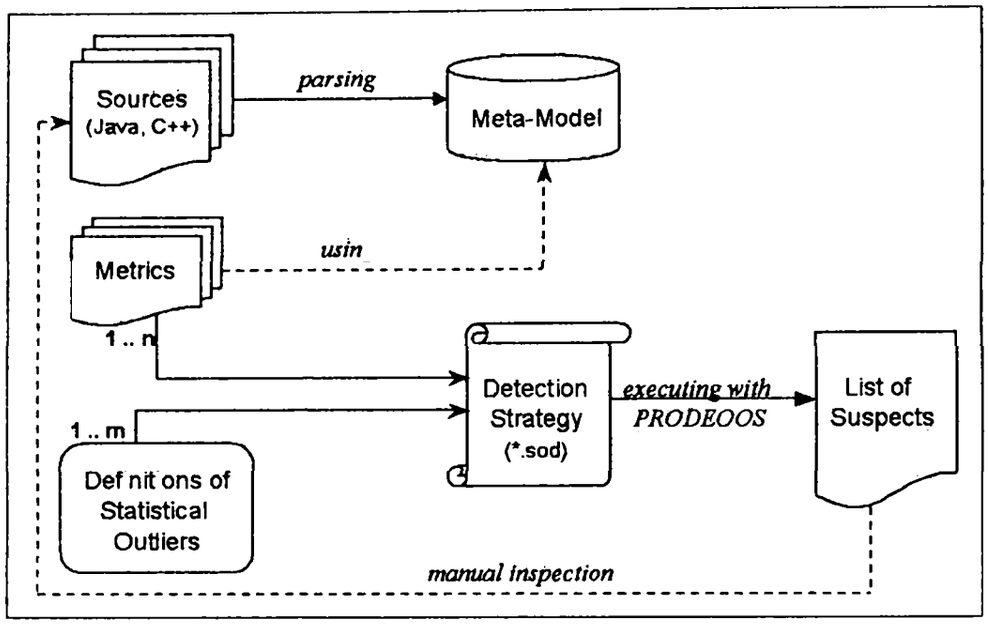



Figure 7.1: An overview of the inspection process. The focus of the figure is set on the artifacts involved in the process.

## 7.2 Tool Support

The following discussion concerning the tool support is structured as follows: first we describe a design-inspection process based on detection strategies, an by describing the phases of this process we identify three key elements. Consequently, each of these elements, are discussed in a separate subsection. The elements that are essential for the tool support are: the meta-model (Section 7.2.2), the tools that extract a design model from the source-code (Section 7.2.3), and the tool that automatizes the execution of detection strategies and FS quality models (Section 7.2.4).

### 7.2.1 Phases of the Inspection Process

The detection process associated with our approach consists of the following steps (see Figure 7.1):

1. **Construction of the System Model.** In order to apply the metrics on a given project, we need to have all necessary design information. This is stored in a *meta-model* (see Section 4.2). The meta-model consists of information about the design entities of the system and about the existing relations among these entities. The extraction of this meta-model is a step that must be done only one time per project. The concrete meta-model implemented in our toolkit is described in Section 7.2.2.

2. **Definition of Detection Strategy.** In the previous chapters we have defined detection strategies and how they can be expressed using the SOD language. In brief, this step mainly consists in selecting a set of metrics which are appropriate for detecting a particular design characteristic (e.g. a design flaw). Metrics are defined as queries on the meta-model built in the previous step. For each metric we associate one or more statistical operators, based on which we can specify which values are the outliers of the metric. Finally, metrics are combined in a strategy using logical connectors. This step must be taken only when a new strategy is defined, independent of the number of analyzed projects.

3. **Running the Detection Strategy.** After a detection strategy is defined, we run it automatically using the PRODEOOS tool in order to get the detection results for the analyzed project. The result is a set of design entities together with the values for the different metrics that were involved in the detection strategy. This step is fully automatized. In Section 7.2.4 we discuss in detail the PRODEOOS application.

4. **Verifying the Results.** The results obtained in the previous step must be manually inspected in order to decide if the suspects conform to the rule quantified in the detection strategy.

The last two steps of the detection process are run for every project and every detection strategy that must be checked for the analyzed project.

### 7.2.2 The Unified Meta-Model

The meta-model used in our approach is a structured collection of design information extracted from the source code, on which the whole analysis takes place. In order to assure that the detection process is language independent as much as possible, we defined a *unified format* for the data tables containing the design information from Java and C++. In order to interrogate the model in terms of metrics, we use a relational-database server. Therefore, the *Unified Meta-Model* is implemented as a set of 6 related database tables:

VARIABLES **Table.** The purpose of this table is to store the information concerning variable declarations, i.e. global variables, class attributes, function/method parameters and local variables (including block variables). The information extracted and stored in the table can be grouped as follows:

- *Context Information.* This includes the name of the class and the signature of the function where the variable is declared.

- *Information about the Variable.* This includes the name, the base type and the complete type (which differs from the base type for pointers, references and arrays) of the variable.

- *Attributes of the Variable.* From these fields we can learn what kind of variable (e.g. global, parameter, etc.) that is and if the type is an abstract data-type (ADT).

METHODS **Table.** The purpose of this table is to store any types of methods defined in the project, i.e. single functions, methods, operators, constructors and destructors. For each method, the following information is extracted:

- *Context Information.* The context consists of the name of the class where the function is defined.

- *Information about the Function.* This includes the name, the list of parameter types and the return type of the function. If a parameter has a "const" specifier this is also stored in the parameter list. If the function returns a constant value the return type is prefixed by the "const" specifier.

- *Attributes of the Function.* There are two fields in which the type of function and the special storage specifiers (virtual, static) are kept. A function can be "static", and methods can also be "virtual".

CLASSES **Table.** This table contains essential information about classes. The information included in this table can be grouped in two categories:

- *General Information.* In this category we have the name of the class and the visibility scope of the class. The visibility scope for a class is different from "global" only for *nested classes* and for classes defined within a particular *namespace.*

- *Attributes of the Class.* There are two special characteristics a class can have: it might be *abstract* or/and *generic* (template). These two boolean fields also belong to the information contained in the *Classes* table.

INHERITANCE RELATIONS **Table.** This table contains the information on *direct* inheritance relationships. This information is structured in three fields: the derived class, the parent-class and the inheritance attribute ("public", "private" or "protected").

ACCESSES OF VARIABLES **Table.** The role of this table is to store all the information about the accesses (uses) of variables. This table contains cross-referencing information, and it relies on the information stored in the VARIABLES and INHERITANCE RELATIONS tables, described before. We can group the information stored in this table in three categories:

- *Location of Access,* i.e. the name of the function where the access takes place, its parameters list and the class that the function belongs to;

- *Accessed Variable,* i.e. the name of the variable, its base-type, the kind of variable (e.g. local, parameter), the name of the class where the variable was declared, and two boolean-fields: one indicating if the variable has a predefined or a user-defined type and the other indicating if the variables has class-scope (is declared static).

INVOCATIONS OF METHODS **Table.** This table stores all the information involved in the invocation (call) of a function or a method. Similarly to the previous table, this one also contains cross-referencing design information, and it relies on the information stored in the FUNCTIONS and INHERITANCE RELATIONS tables. The data contained in the table can be grouped in two categories:

- *Location of Invocation,* i.e. the name of the function where the invocation occurred, its parameters list and the class that the function belongs to.
- *Invoked Method,* i.e. the name and the parameter list of the invoked function, the class in which the invoked method is defined, and the kind of function (e.g. "single-function", "public-method").

### 7.2.3 Meta-Model Extractors

#### TableGen

In order to analyze C++ projects we developed TABLEGEN [Mar97a] that scans a C++ project and extracts the essential static design information from the source code, storing it into ASCII tables. The tool is intended to be part of larger CASE and/or reengineering tools. The place of TABLEGEN in the architecture of such a tool is at the lower level, whereby its role would be to stand between the source-code and the higher levels of the tool, offering to the latter all the necessary information upon which the design or reengineering methods and techniques can be applied.

#### MeMoJ

The relational form of the unified meta-model initially started from the analysis of C++ source-code. Later we have defined MEMOJ, in form of a Java library, as an object-oriented meta-model mainly for Java sources. This meta-model also stores the design information needed for code analysis, especially the one which is necessary to compute the metrics involved in our detection strategies. A brief summary of the MEMOJ architecture is depicted in Figure 7.2. MEMOJ is based on the Java parsing technology of COMPOST [ALN00][3]. In order to assure that we have indeed a unified expression of the meta-model for both C++ and Java, we implemented the Java correspondent of TABLEGEN, called JTABLES. As a consequence, we are now able to analyze both *Java* and *C++* projects on the same meta-model.

[3]Beginning with 2001 it became Sourceforge project, called RECODER. Further information is available on http://recoder.sourceforge.net/

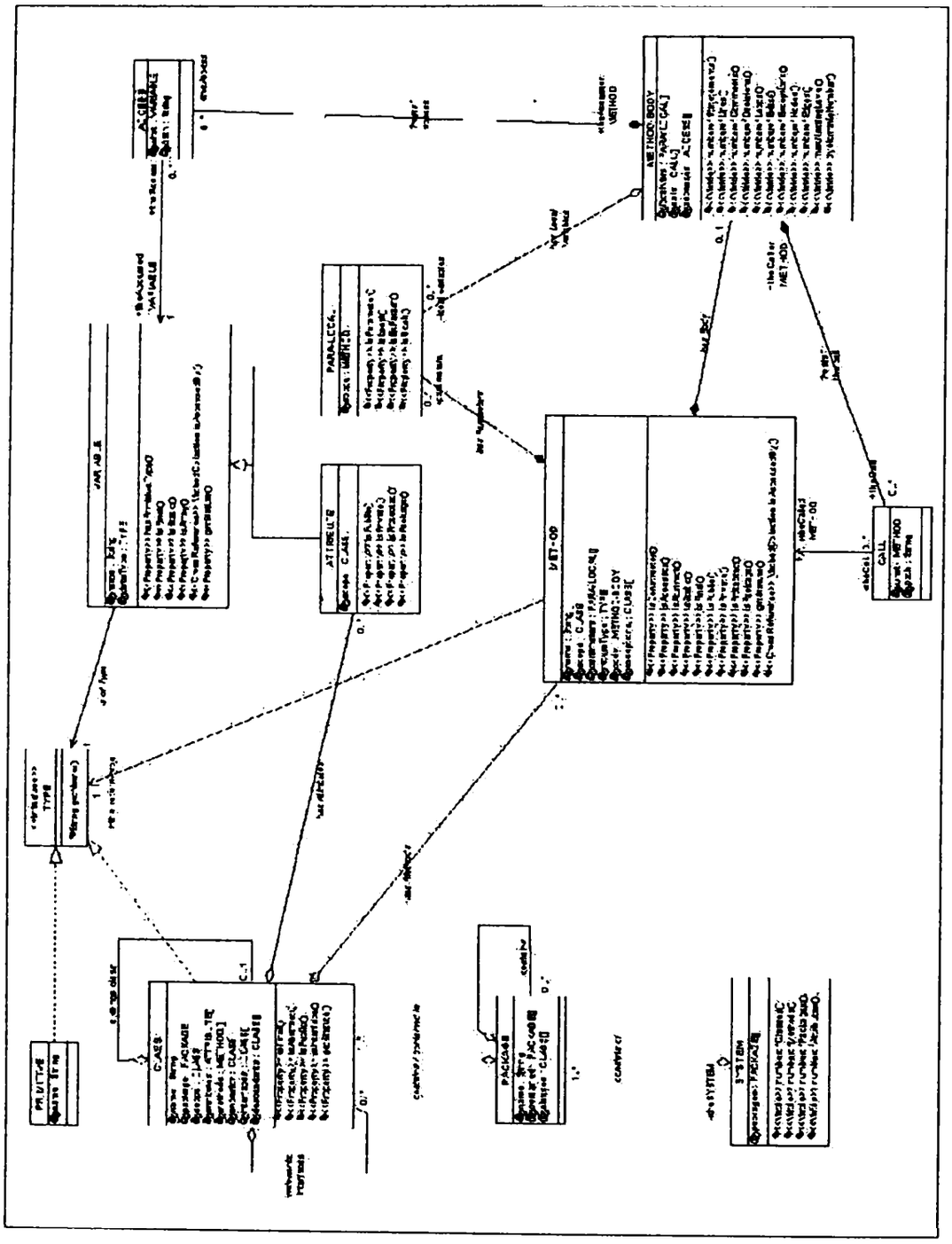


Figure 7.2: The MEMOJ Meta-model

**Remark** The major advantage of using a meta-model is that it assures the language-independency of the detection process. After the model for a given project was built the rest of detection process is language independent (of course only if the language stays in the realm of object-orientation). Currently we have support for the C++ and the JAVA programming languages and we envision future support for a third object-oriented language, probably ADA95.

### 7.2.4 ProDeOOS

PRODEOOS[4] is a tool that we designed and implemented [Chi01] in order to support detection strategies based on the SOD language. It allows us to create, modify and execute detection strategies. The tool uses the SOD language for defining strategies, while all the metrics are implemented as SQL queries based on the information found in the *Unified Meta-Model Repository*.

PRODEOOS has a modular structure consisting of three main components: the *Design Inspector*, the *Quality Studio*, and the *Report Generator*. We will briefly describe these modules next.

#### The Design Inspector

This module operates on a tree structure that models the detection strategy. Thus, in order to use this module the user must first load a SOD strategy file. After the strategy file was loaded, at the user's click on the detection button (see Figure 7.3) the results for each node are computed by executing the queries that implement the metrics. Results are displayed in the right pane of the window, as a table. The overall results of the strategy are displayed if the root node of the tree is selected. Partial results are also important for the analysis. They are available by selecting a node from the detection tree.

The results consist of the measured entities and the values for the different metrics that are involved in a detection strategy. If the value for a metric is not displayed in the result table, this means that the value for that metric is not critical, i.e. it is not an outlier value.

#### The Quality Studio

This module is intended to support the automatic evaluation of the design against a given quality model based on detection strategies. The quality models are described using the SDQT language. As a result of this evaluation, the engineer first gets a score for the quality of the evaluated attribute. But the tool also allows the engineer to navigate through the quality tree and inspect all the intermediary scores down to the detection strategies. This way it becomes easy to trace back how the score was computed and thus quickly identify the causes of a possible low score.

[4] **PRO**blem **DE**tector for Object-Oriented Systems

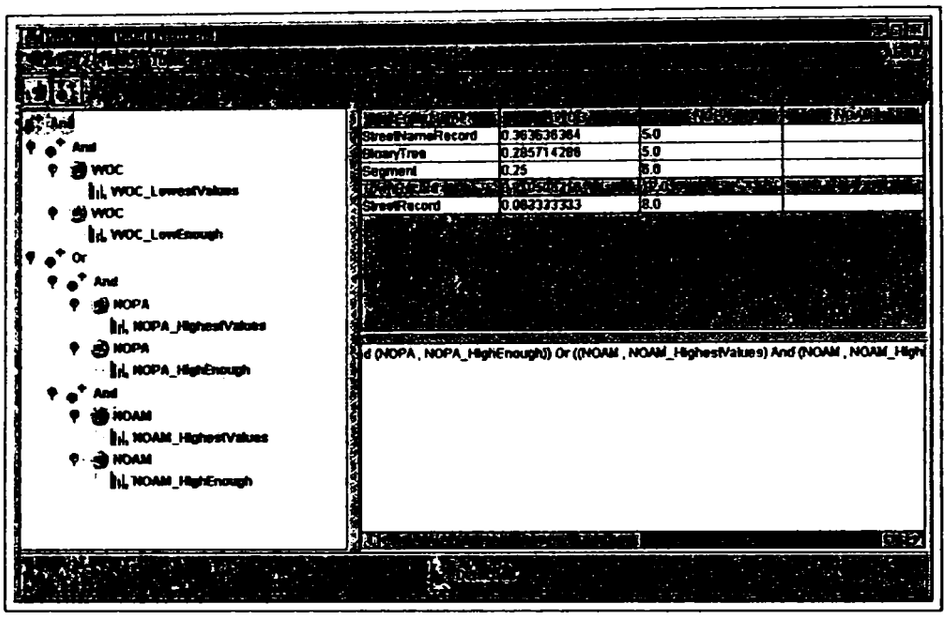



Figure 7.3: The Design Inspector module

### The Report Generator

PRODEOOS as a tool is needed to *run* the strategies and the quality models and retrieve the results. For the *analysis* of the results we don't need the tool anymore, but just the results. So, we would like to have the "navigability" offered by PRODEOOS even after closing the tool. For this purpose, we defined a *Report Generator* in PRODEOOS that "freezes" the results in form of a multi-frame HTML report. Saving the results in form of a navigable report has the major advantage that the results can be analyzed by anyone and anywhere a HTML browser is available. We designed the look and feel of the report to be very similar to the main window of PRODEOOS (see Figure 7.4)

## 7.3 Evaluation of Detection Strategies

The evaluation was focused on the following desirable characteristics of a detection strategy used for the identification of design flaws:

1. Detection Strategies should be an *accurate mapping mechanism* between measurement results and design problems. Therefore, during evaluation, we want to prove if abnormal numbers are indeed pointing out to real design problems

2. Detection Strategies should properly *identify the problems*. Thus, a strategy is good if based on the symptoms (i.e. the measurement results) it can detect the disease (i.e. the design flaw) that it claims to detect.

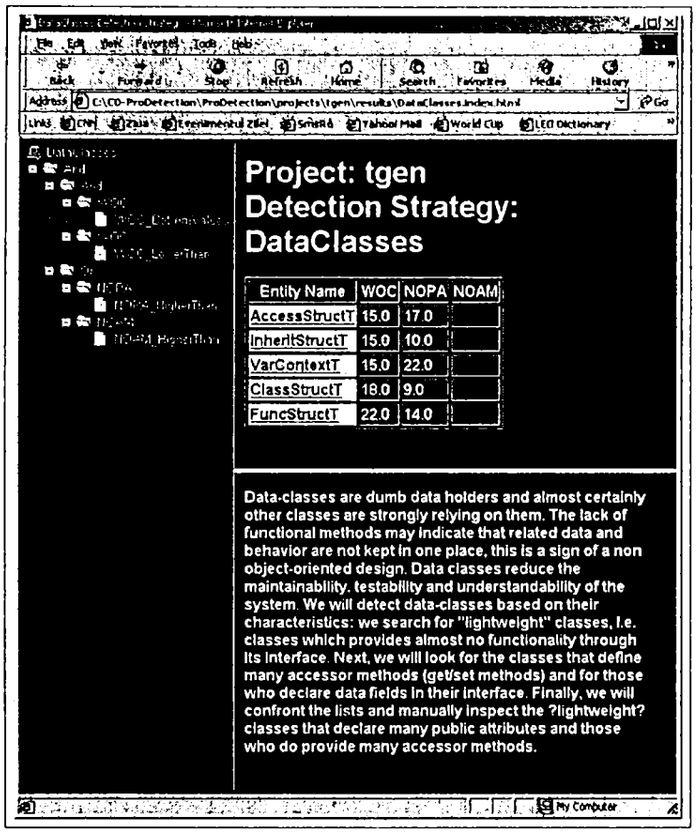



Figure 7.4: Report generated with PRODEOOS

3. Detection Strategies should be able *to localize the problem.* Therefore, a strategy is good if it able to identify those design fragments that are affected by a particular design flaw

### Remarks

Although not obvious at first sight, there is a clear difference between the first mentioned characteristic and this last one; while the first characteristic is limited to the ability of detection strategies to be an efficient higher-level interpretation mechanism for metric results that leads us to flawed design fragments, the last characteristic is more challenging, as it requires from a detection strategy to specifically identify *the* design flaw that it claims to detect.

The reader should also note that we do not suggest that there is a one-to-one mapping between the design problem that affects a design entity and a detection strategy. In other words, it is possible for an ill-designed entity to be reported by two or more related strategies. This case will be illustrated later in

| Suspect Before? (in SV1) | Suspect After? (in SV2) | Conclusion |
|---|---|---|
| YES | YES | False Positive<br>Two possible conclusions:<br>• the strategy is not precise enough<br>• because of the specific reengineering goal the suspect design fragment was not refactored |
| YES | NO | Real Flaw<br>Three possible situations:<br>• in the same package<br>• in a different package<br>• not found at all |
| NO | YES | Special Case<br>Needs further investigation |

Figure 7.5: The decision table used to classify the suspects from the two versions of the analyzed system, in conformity with *Assumption 1.*

this chapter (Section 7.3.4), as we will make an in-depth analysis on the results of the *God Class* detection strategy.

### 7.3.1 Two Evaluation Methods

In the following we present two methods that we used to evaluate the *scalability* and the *accuracy* of detection strategies.

#### Automatic Classification

In Figure 7.5 we synthesized an approach that allows us to evaluate if the detection strategies detect *real* design problems. The main idea is that we split the suspects (indicated by a detection strategy) in the first version of a system (SV1) in two categories: real flaws and false positives. We differentiate between the two categories based on a corollary of *Assumption 1*, namely: a suspect is affected by a *false positive* if it reoccurs in the list of suspects in SV2, for the same detection strategy; otherwise a *real flaw* was detected. By dividing the number of real flaws by the total number of suspects we get the *accuracy rate* for the given detection strategy. The main advantages of this approach are that it is highly *objective* and it can be easily *automatized.*

#### Manual Investigation

Because the *Automatic Classification* approach is "blind" based on a reasonable yet fallible assumption, it doesn't give us the certainty that the classification of the suspects, and consequently the computed accuracy factor are correct. Therefore, we decided to use in addition to the described approach a further manual investigation, which involves the manual inspection of all the suspects.

The manual investigation allows a proper evaluation of both accuracy aspects that are relevant for a detection strategy, i.e. it allows us to identify those

suspect design fragments, which although ill-designed, are not affected by the design-flaw quantified by the detection strategy (see Section 7.1.1).

### 7.3.2 Results Summary

In this section we will just summarize the results for the two evaluation approaches; then, in the next section, we interpret these results in respect to the evaluation criteria defined in Section 7.1.1. Figure 7.6 presents the results of applying a suite of detection strategies on the two versions of the analyzed system. The table also contains the partitioning of the suspects in SV1 in conformity with the rules of the *Automatic Classification* approach (see Figure 7.5). In the last column of the table we see the *accuracy rate* for each evaluated detection strategy. The accuracy rate is computed as the number of real flaws divided by the total number of suspects in SV1.

In Figure 7.7 we summarized the results of the *Manual Investigation* approach

| Abstraction Level | Design Flaw | Suspects in SV1 | Suspects in SV2 | False Positives | Special Cases | Real Flaws | Accuracy Rate |
|---|---|---|---|---|---|---|---|
| Methods | *Feature Envy* | 40 | 15 | 11 | 4 | 25 | 63% |
| | *God Method* | 4 | 4 | 1 | 3 | 3 | 75% |
| Classes | *Shotgun Surgery* | 15 | 7 | 6 | 1 | 9 | 60% |
| | *Refused Bequest* | 22 | 6 | 4 | 2 | 18 | 81% |
| | *God Class* | 5 | 2 | 2 | 0 | 3 | 60% |
| | *Data Class* | 3 | 2 | 1 | 1 | 2 | 66% |
| Packages | *God Package* | 2 | 1 | 1 | 0 | 1 | 50% |
| | *Misplaced Class* | 4 | 2 | 1 | 1 | 3 | 75% |
| | *Wide Subsystem Interface* | 5 | 1 | 1 | 0 | 4 | 80% |
| Micro-Architecture | *Lack of Bridge* | 0 | 0 | 0 | 0 | 0 | – |
| | *Lack of Strategy* | 4 | 2 | 2 | 0 | 2 | 50% |
| | *Lack of Singleton* | 4 | 5 | 2 | 3 | 2 | 50% |

Figure 7.6: Accuracy scores for the suite of detection strategies based on the *Automatic Classification* approach

for a subset of the previously analyzed strategies. In this case, we only analyzed the suspects from the first version of the system (SV1). The total number of suspects for a detection strategy is classified in three categories, after a manual inspection: *Correct Detection* contains those suspects that proved to be affected by the design flaws that the detection strategy claims to find; the *Other Flaw* category contains the suspects that proved to be ill-designed but are affected by another design problem than the one supposed to be captured by the strategy; the last category, *False Positive* includes all the suspects that are not at all affected by a design problem.

Using the manual investigation approach we can compute two types of accuracy rates: a *strict* one that only counts the cases in which the "diagnosis" was correct and *looser* one that counts all the suspects that were indeed ill-

| Abstraction Level | Design Flaw | Total Suspects | Correct Detection | Other Flaw | False Positive | Strict Accuracy | Loose Accuracy |
|---|---|---|---|---|---|---|---|
| Methods | *God Method* | 4 | 2 | 1 | 1 | 50% | 75% |
| Classes | *Shotgun Surgery* | 15 | 10 | 2 | 3 | 66% | 80% |
| | *God Class* | 5 | 3 | 1 | 0 | 80% | 100% |
| | *Data Class* | 3 | 3 | 0 | 0 | 100% | 100% |
| Packages | *Wide Subsystem Interface* | 5 | 3 | 1 | 1 | 60% | 80% |
| Micro-Architecture | *Lack of Strategy* | 4 | 3 | 1 | 0 | 75% | 100% |
| | *Lack of Singleton* | 4 | 1 | 1 | 2 | 25% | 50% |

Figure 7.7: Accuracy scores for the suite of detection strategies based on the *Manual Investigation* approach, applied to SV1

designed. Thus, the accuracy rates are computed as follows:

$$StrictAccuracy = \frac{CorrectDetection}{TotalSuspects}$$

$$LooseAccuracy = \frac{CorrectDetection + OtherFlaw}{TotalSuspects}$$

### 7.3.3 Assessment of the Evaluation Criteria

#### Scalability

The first aspect of scalability, mentioned in Section 7.1.1, is concerned with the relative number of design entities detected by a strategy. Analyzing the results in Table 7.6 in connection with the size information about the SV1 system (Table 7.1) we see that at the class level the percentage of suspects is not higher than 14.4% (for *Refused Bequest*), at the method level it reaches a maximum of 3.1% (for *Feature Envy*), and at the subsystem level the percentage is also not higher than 27.7%. All of these values are a concrete proof that the way strategies are defined reduces the analysis effort considerably Therefore, from this point of view, the detection strategies defined in Chapter 5 proved to scale for large systems.

The other aspect of the scalability issue is related to execution time of detection strategies. The time needed for the execution of detection strategies on the latter version of the system (SV2)[5] varied between a couple of seconds (e.g. *Wide Subsystem Interface*, *Data Classes*) and several (3-5) minutes (e.g. *Lack of Strategy*, *Shotgun Surgery*). The only outlier from this point of view proved to be the *Feature Envy* strategy, which needed 1.5 hours to deliver the results[6]. Comparing these times with the dimensions of the system we may conclude that all the detection strategies proved to be scalable with respect to execution times.

[5] At this we use SV2 as a "benchmark" because it is considerably larger than the first version (SV1) and therefore the results are more relevant from the point of view of scalability

[6] All the before-mentioned times were measured with using a machine based on an AMD Duron 800 CPU and 512 MB of RAM, running under Windows 2000

### Accuracy

In Section 7.1.1 we distinguished between two aspects of the accuracy criterion: one that evaluates the ability of a detection strategy to find ill-designed code fragments (loose accuracy) and another one that deals with the ability of a strategy to find only those design entities that are affected by the particular flaws that the strategy claims to detect (strict accuracy). Using the *Automatic Classification* approach we could only measure the strict accuracy of the evaluated strategies. For all the strategies the accuracy rate was over 50%, while the average accuracy is 64.5%. The strategies with the lowest accuracy rates (50%) are: *God Package*, *Lack of Strategy* and *Lack of Singleton*. While for the *God Package* we believe that the low rate is due to the low absolute number of suspects (2), for the other two cases we decided to include the strategies in the subset of strategies used for the *Manual Investigation* approach in order to get a clearer picture on the accuracy. Based on the results of the manual approach we conclude that the *Lack of Strategy* detection strategy proved to be extremely accurate in the end, while the *Lack of Singleton* strategy proved to be a quite imprecise instrument for detecting the flaw it was designed for.

Using the *Manual Investigation* evaluation method we could compute both the *loose* and the *strict* accuracy rate. Because the manual investigation is a time consuming operation we limited it to only a subset of detection strategies. The average *loose accuracy rate* was in this case 83.57%, substantially higher than the one obtained using the automatic approach. Thus, a first conclusion is that the strategies that we defined so far have a high accuracy in detecting ill-designed entities.

As we compute the average of the *strict accuracy rate* the result is 65.1%. Thus, in almost two out of three cases the suspect entity was indeed affected by the expected design flaw, which we consider to be an acceptable accuracy rate.

## 7.3.4 In-Depth Analysis: Detection of God Classes

### Applying the Detection Strategy.

First, we applied the ATFD metric and found 5 outlier classes, with values between 7 and 4. Next, we computed the value for the other two metrics, WMC[7]. With two exceptions, the ATFD outliers were also among the WMC outliers. On the other hand, surprisingly, only two ATFD outlier classes proved to be non-cohesive. The results are summarized in Table 7.3.

Based on these results and on our selection criteria, we decided to pick up the (`CParser`) class for an in-depth analysis, as it was among the outliers for all the three metrics that compose the strategy. We inspected this class, by analyzing its relations to the data-classes that it was using and the conclusion

[7]For the WMC metric we used two definitions: the one based on unitary complexities (NOM) and the other based on McCabe's cyclomatic complexity

was that the CParser definitely centralizes almost all the functionality and uses the other classes only to get or to set some data.

| Class | ATFD | WMC | TCC |
|---|---|---|---|
| CParser | 7 | 75 | 0.28 |
| SuperPolygonArray | 5 | 54 | — |
| StreetNameTable | 5 | 44 | — |
| CTownNearestSearch | 4 | — | 0.25 |
| TownView | 4 | 43 | — |

Table 7.3: The suspect classes for the *God Classes* detection strategy. A dash (—) indicates that the class is not among the outliers for that metric.

### Findings

**Relation between class CParser and the predicate classes (CPredicate).** Class CParser parses a string representing a selection query (SELECT) and keeps two lists: one containing the selected columns and one containing the predicates that compose the conditional clause (WHERE) of the selection. The predicates can be of different types, – four are identified in the current version but the number may grow – each one being parsed differently. In the current implementation, for each predicate type a class is defined. These classes are in fact "dumb" records (see Figure 7.8) that contain only the fields that keep the specific data for each type of predicate. In the CParser class, for each predicate type a method is defined that parses the predicate and sets the fields.

*Improvement Proposal:* The design can be improved by applying the *Strategy* pattern [GHJV94] delegating the responsibility of parsing a particular predicate to the class defined for that predicate (see Figure 7.9); consequently, the setting of the predicate fields will also move inside the predicate classes. This way, the CParser class is simplified, the predicate classes become "responsible" classes, and the coupling of the parser class to the particular predicate classes is reduced.

**Remark.** As the improvement solution was a pattern we naturally analyzed if the CParser class was also among the suspects of the *Lack of Strategy* detection strategy. And indeed it was! This is a concrete illustration of the remark from the beginning of this section where we emphasized the fact that there is not always a one-to-one mapping between a design problem and a strategy.

**Relation between classes CParser and CScanner.** Class CScanner is in fact a string tokenizer used by the CParser class to parse a SQL selection statement. The CScanner class has a field that indicates the type of the current token. If the token is a number or an identifier the scanner must also provide the value of the number or the string. In the current design CScanner is a structure that exposes all the implementation details, and CParser directly reads the

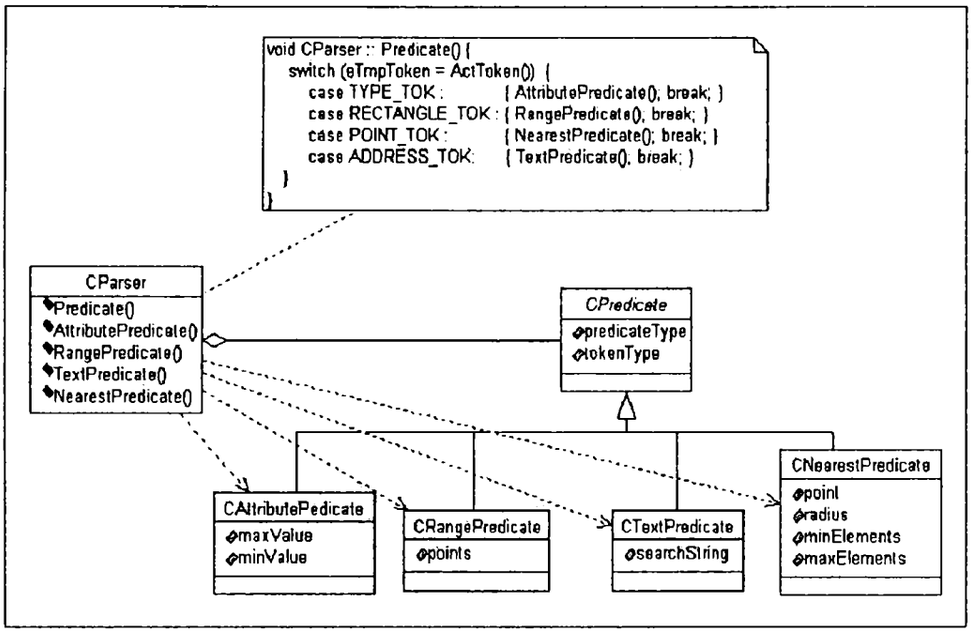



Figure 7.8: The CParser "God Class"

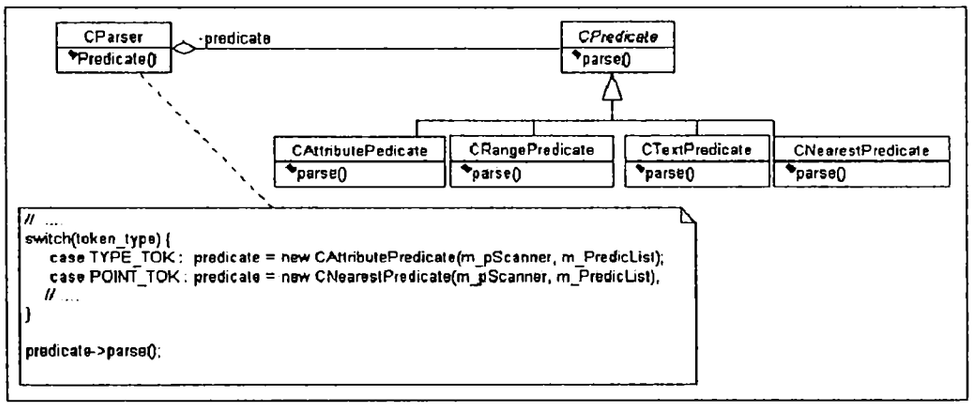



Figure 7.9: The Reengineered version of CParser

fields from the tokenizer.

*Improvement Proposal:* The design could be improved by setting the data fields from CScanner as private and introducing a method (e.g. scan) in the CScanner class. This method should be overloaded so that it can accept three types of parameters: a token-type identifier, a number or a string. If scan is called from CParser with a string parameter, the method checks if the token is an identifier and if so it copies the value to the parameter. If the token is not an identifier, the method will throw an exception that will be caught in the CParser class.

## 7.4 Evaluation of Factor-Strategy Models

In this section, we evaluate the efficiency of Factor-Strategy quality models, based on the quality model for maintainability defined in the previous chapter (Section 6.4.3). The evaluation will be based on the comparison between the two versions of the case-study, and especially on the knowledge that the re-engineering goal that drove the structural changes in the second version was the *enhancement of maintainability*. Thus, the assumption that the level of maintainability for the second version is higher than the one in the first version is a plausible assumption that can be used to evaluate the proposed quality model.

### 7.4.1 The Evaluation Methodology

The evaluation approach for the Factor-Strategy approach to quality models is very simple. We take the quality model built in the previous chapter for assessing maintainability and apply it on the two versions of the system. As the second version was reengineered with the goal of eliminating maintenance difficulties, we want to see if our Factor-Strategy quality model properly reflects the improvements.

A second evaluation criterion is to see if the model can help us identify the major design flaws that made the difference in maintainability between the two versions.

### 7.4.2 Results Summary

The results of applying our FS quality model on the two versions of the system are synthesized in Figure 7.10. All the scores were computed based on the evaluation mechanisms for the quality model described in Section 6.3. A complete description of the quality model can be found in Appendix A. First of all, note the difference of 1.12 points (on a 1 to 10 scale) of the maintainability score, which indicates that the quality model captured a sensible improvement of the level of maintainability. More than that, for each of the four quality criteria, we can see that the model shows an improvement, varying from 0.75 points (for Changeability) up to 1.4 (for Testability) and 1.33 (for Analysability).

### 7.4.3 Assessment of the Accuracy Criterion

The first observation concerning the accuracy of the FS model on maintainability is that an increase of quality was "sensed" for all four quality factors. In addition to this, the fact that the highest improvements were registered for *Testability* and *Analysability* is an extremely positive signal concerning the accuracy of the model, as these two quality aspects are the most important ones, for a system that is going to be extended with new functionality, like the analyzed

| | | | SV1 | | SV2 | |
|---|---|---|---|---|---|---|
| Quality Goal | Factor | Strategy | Score | Qualifier | Score | Qualifier |
| Maintainability | | | 6,95 | | 8,07 | |
| | Changeability | | 7,58 | 8 | 8,33 | 8 |
| | | LackOfBridgeDeep | 0 | 10 | 0 | 10 |
| | | LackOfBridgeShallow | 0 | 10 | 0 | 10 |
| | | LackOfStrategyGodLike | 0 | 10 | 0 | 10 |
| | | LackOfStrategyHierarch | 4 | 8 | 2 | 10 |
| | | LackOfState | 0 | 10 | 0 | 10 |
| | | LackOfFacade | 5 | 5 | 1 | 9 |
| | | GodMethod | 4 | 8 | 5 | 8 |
| | | DataClasses | 3 | 7 | 2 | 9 |
| | | GodClasses | 5 | 5 | 2 | 9 |
| | | ShotgunSurgery | 15 | 3 | 7 | 3 |
| | | GodPackage | 1 | 9 | 2 | 9 |
| | | ISPViolation | 10 | 6 | 11 | 3 |
| | Analyzability | | 6,67 | 6 | 8 | 8 |
| | | LackOfStrategyGodLike | 0 | 10 | 0 | 10 |
| | | GodMethod | 4 | 8 | 5 | 8 |
| | | DataClasses | 3 | 8 | 2 | 10 |
| | | GodClasses | 5 | 5 | 2 | 9 |
| | | RefusedBequest | 22 | 3 | 6 | 6 |
| | | MisplacedClass | 0 | 10 | 0 | 10 |
| | | GodPackage | 1 | 9 | 2 | 9 |
| | | TemporaryField | 31 | 4 | 20 | 5 |
| | | ShotgunSurgeryWeighted | 33 | 3 | 18 | 5 |
| | Testability | | 6,2 | 6 | 7,6 | 8 |
| | | GodMethod | 4 | 7 | 5 | 5 |
| | | DataClasses | 3 | 8 | 2 | 10 |
| | | GodClasses | 5 | 5 | 2 | 9 |
| | | ShotgunSurgery | 15 | 3 | 7 | 6 |
| | | LackOfSingleton | 4 | 8 | 5 | 8 |
| | Stability | | 7,33 | 8 | 8,33 | 8 |
| | | LackOfState | 0 | 10 | 0 | 10 |
| | | GodMethod | 4 | 8 | 5 | 8 |
| | | DataClasses | 3 | 8 | 2 | 10 |
| | | GodClasses | 5 | 5 | 2 | 9 |
| | | ShotgunSurgeryWeighted | 33 | 5 | 20 | 5 |
| | | LackOfSingleton | 4 | 8 | 5 | 8 |

Figure 7.10: Results Summary. The numbers in the two "Score" columns are given by the number of suspects reported by each detection strategy, while the "Qualifier" is the corresponding score taken from the scoring table.

system is. Thus, these results tend to confirm our first hypothesis, i.e. that FS quality models are efficient means for defining associations between external quality goals and the quality of the design structure.

Next, we wanted to identify what caused the difference between the maintainability scores of the two versions. For this, we went through the strategy-level of the quality-model and selected those with the most important score increases. The results are synthesized in Figure 7.11 The answer to the question above illustrates best the advantages of the Factor-Strategy approach: we do not receive – like in a usual FCM approach – a bunch of numbers (i.e. the compared

| Strategy | Initial System (SV1) | | Re-engineered System (SV2) | | Improvement | |
|---|---|---|---|---|---|---|
| | Score | Qualifier | Score | Qualifier | Score | Qualifier |
| Wide Subsystem Interface | 5 | 5 | 1 | 9 | 4 | 4 |
| God Class | 5 | 5 | 2 | 9 | 3 | 4 |
| Refused Bequest | 22 | 3 | 6 | 6 | 16 | 3 |
| Data Class | 3 | 7 | 2 | 9 | 1 | 2 |
| Shotgun Surgery | 15 | 3 | 7 | 3 | 8 | 0 |

Figure 7.11: The top of the most spectacular score improvements between SV1 and SV2. These results help us drive immediate conclusions on the major design problems in SV1 that hindered its maintenance.

measurement results) together with statements like "the results for metric X and Y improved in the second version compared to the first one", which require further effort to be converted into useful information for an engineer. Instead, we can now reason and pick up conclusions *directly* from our quality model!

In this concrete case, when we analyze Figure 7.11 we can immediately drive the following conclusions:

- The main problems are related to the class and package design rather than methods implementation.
- The problems at the package level were strongly related to the interface of the packages, i.e. the number of classes from outside the package directly using it (*Wide Subsystem Interface*)
- The problems at the class level were basically related to the tendency to centralize the intelligence of the system in a small number of classes (*God Class*) that used several "dumb" data-holders (*Data Class*), an improper class hierarchy (*Refused Bequest*) and an increased coupling level between the classes (*Shotgun Surgery*)

The manual analysis of the SV1 system, together with further information concerning the initial weaknesses of the system fully confirmed the conclusions above.

# Chapter 8

# Conclusions and Perspectives

## 8.1 Summary

The thesis of this dissertation, as stated at the beginning of this work, is: the gap between qualitative and quantitative statements concerning object-oriented software design can be bridged using a higher-level, goal-driven approach for measurements interpretation. In this context, the goal of this work was to develop methods and techniques that provide such a relevant, goal-driven interpretation of measurements applied to the investigation of object-oriented software design. The idea of this approach is to use a decompositional approach that formulates (expresses) the investigation goal in terms of metrics-based design-related rules, called detection strategies.

In the last decade software metrics became more and more an important means for assessing and controlling the quality of software systems in general and for object-oriented systems in particular. We researched the former approaches, covering the main contributions on measurement definition and interpretation, the methods and techniques of design inspection and the various ways of defining quality models. During this research we discovered that there is still a large gap between the use of individual measurements and the principles that rule the construction of object-oriented software. Current approaches do not provide a way to quantify such design principles and rules. In addition to that, as we analyzed the existing approaches on quality models we discovered that a relevant feedback link from the results of individual measurements to the quality factors was missing. A metric in itself does not provide enough information for making a decision for a code transformation that would improve quality. Thus, the developer is provided only with the problem, while he or she must still empirically find the real cause and eventually look for a way to improve the design.

In this context, our work introduced a new mechanism, named *detection strategy*, for increasing the relevance and usability of metrics in object-oriented design by providing a higher-level (more abstract level) means for interpreting measurement results. A detection strategy was defined in this work as the quantifiable expression of a rule by which design fragments that are conforming to that rule can be identified in the source-code. The main goal of a strategy is to provide the engineer with a mechanism that will allow him or her to work with metrics on a more abstract level, which is conceptually much closer to his real intentions in using metrics. Such rules may refer both to design recovery (*understanding of design*) and the identification of design flaws (*evaluation of design*).

After introducing detection strategies, together with a methodology for transforming an informal design rules into a quantifiable expression, we moved on to apply the described mechanism to a concrete investigation goal strongly related to design quality: the identification and localization of design flaws in object-oriented systems, which are seen as violations of design principles, rules and heuristics. Thus, we defined a suite of strategies for detecting design problems at four levels of abstraction, starting with the lowest level of design problems at the method level, going through the class and package and ending with strategies for the detection of flaws at the micro-architectural level where we proposed strategies for identifying design fragments where particular design patterns should have been applied. Our initial assumption that design rules and heuristics can be captured in form of metrics-based rules proved to be true as we ended with a suite of over twenty detection strategies.

As the ultimate goal of this dissertation was to demonstrate that the gap between qualitative and quantitative statements can be bridged, we focused next our attention on the issue of quality models. As we did so, we discovered a general difficulty of classical approaches, which can be synthesized as follows: the possibility to identify the real causes of quality weaknesses as perceived from the outside (e.g. poor maintainability, low reuse level) is strongly hampered by the fact that the metrics level in quality models is too fine-grained to allow an understanding of the real design problems, which consequently hinders the proper redesign decisions, for a long-term increase of quality. Therefore, we defined a new quality model in which quality factors are expressed (in the sense of decomposition) in a set of quantifiable rules that identify violations of design principles, rules and heuristics. Of course, these quantifiable rules are given by the suite of detection strategies defined earlier in this work. Thus, instead of communicating directly with a "bunch" of numbers, which has a low relevance level, in our approach quality is expressed and evaluated in terms of an explicit *"knowledge-box of object-oriented design"* containing the quantified expressions of the good-style design rules for the object-oriented paradigm.

The approach was evaluated based on a case-study, containing two versions of an industrial system, whereby between the first and the second version a reengineering process took place. The evaluation consisted of two parts: firstly

we assessed our suite of detection strategies for identifying (mining) design flaws by comparing the results obtained on the first version of the case-study with its second, reengineered version. The second part was focused on the assessment of the quality model for maintenance that we built in the previous chapters. The results from this case-study and from several other ones, proved on the one hand that detection strategies are a means usable in practice for quantifying principles, rules and heuristics related to design. On the other hand the results proved that the relevance of the information provided by the quality model based on detection strategies is higher and more useful than the previous approaches.

## 8.2 Evaluation of Contributions

The approach presented in this work brings a number of essential contributions to the field of quality assessment in object-oriented design based on software metrics. These contributions are summarized below as follows:

- **Definition of the *detection strategy* concept** as a proper means for a higher-level measurement interpretation. Detection strategies are a means for defining metrics-based rules. They offer an encapsulation of metrics in a higher-level construct that permits a more meaningful interpretation and usage. For the first time, detection strategies become a mechanism for "articulating" (expressing, quantifying) rules related to the structure of code and design in terms of metrics.

- **A method usable in practice for quantifying informal code- and design-related rules**. We defined not only the mechanism to objectify and quantify such rules, but we also provided a methodology for moving from informal descriptions of rules to detection strategies.

- **A suite of detection strategies for the identification of well-known design flaws**. For the first time descriptions of design problems like "Feature Envy" [FBB+99] or "God Classes" [Rie96] are quantifiable and detectable. In addition to that, the suite also contains detection strategies for identifying places in the source code where a particular design pattern should have been applied.

- **Definition of a quality model based on detection strategies**. Although the proposed model is also decompositional one, yet for the first time in this model quality communicates with the principles, rules and heuristics of good object-oriented design – objectified by the different detection strategies – instead of being mapped to "pure numbers" (measurement results). The main advantage of this new quality model is that it leads to a direct identification of the real causes of quality flaws, as they are reflected in flaws at the design level.

- **Strong tool support for the high level of automatization and scalability** of the approach. The tool support is covering all the methods

and techniques presented in this work from the definition and execution of new metrics and detection strategies, up to the definition of complex quality models based on detection strategies. The tool support has a high-level of language-independency, as it works on a unified meta-model for object-oriented languages. The provided tool support makes the approach scalable, and we proved that by analyzing real-world systems containing up to 1 MLOC.

- **Concrete evaluation of the concepts, methods and techniques** defined and presented in this thesis, based on two versions of an industrial case-study and a suite of smaller research and university projects.

## 8.3 Future Work

During our research we encountered a number of questions that we believe that are worth of further investigation in the future. We classified the possible continuations of this work in three categories: refinement, migration and integration.

### 8.3.1 Refinement

- **The issue of threshold values.** Although the work provides some answers on the question of how to define and set the threshold values for individual measurements some improvements are visible, as for example the usage of fuzzification techniques [SB01], [Kos96] or the building of neural networks that would learn the proper values for the thresholds [Mih02].

- **Refinement of a new strategy-based quality model.** In this work we proposed a concrete quality model for the assessment of maintainability, based on detection strategies. The question that we would like to answer in the future is how can further quality factors be expressed in a similar manner? A further question is: is it necessary to build *domain-specific* quality models?

- **Suite of detection strategies for design recovery**. We have described in this work how detection strategies can be used taking the detection of design problems as an investigation goal. Further investigation goals could be considered in order to define similar suites. One of these goals could be to have detection strategies for design recovery, which would be of high interest if we consider the impressive number of legacy systems to be reengineered.

### 8.3.2 Migration

- **Migration to emerging programming paradigms**. The question here is: How can the method and the strategies presented in this work be used beyond the limits of object-orientation? Can we, for example, define

detection strategies for adaptive (AP) or aspect-oriented programming (AOP)? Which would be the invariants of the approach? Which are the parts that are going to change?

- **Migration to new technologies.** We evaluated our methods and techniques mainly on "classical" object-oriented systems. In the recent years a lot of technologies were developed for web-based applications (e.g. Enterprise Java Beans); therefore we are asking how can our approach support these technologies?

- **Detection strategies for component identification.** Recent efforts [Tri01] focused on developing methods and techniques for component identification in legacy systems. We believe that detection strategies are a good means for expressing and composing the rules the identification of "componentifiable" design rules.

### 8.3.3 Integration

The whole approach presented in this thesis, rather than being theoretical is very close to the world of practical software engineering. Therefore, we assumed from the beginning that the approach will become in the near future very interesting for CASE tool providers. Some preliminary discussions with several important companies justified our initial assumptions. This raises the question of integration, i.e. how can the techniques and methods developed during this dissertation be integrated in a commercial development environment?

# Bibliography


[ALN00] U. Assmann, A. Ludwig, and R. Neumann. Compost home page. *http://i44www.info.uni-karlsruhe.de/compost*, March 2000.

[AM96] F.B. Abreu and W. Melo. Evaluating the Impact of Object-Oriented Design on Software quality. *Proc. of Symposium on Software Metrics METRICS '96*, mar 1996.

[Bär98] H. Bär. Automatische Suche von Designproblemen in Objektorientierten Systemen. Diplomarbeit, Universität Karlsruhe, März 1998.

[BBC+99a] H. Bär, 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.

[BBC+99b] H. Bär, M. Bauer, O. Ciupke, T. Genssler, R. Marinescu, B. Schulz, and J. Weisbrod. Sanierung objektorientierter Systeme. In *Proceedings of the First German Workshop on Software-Reengineering*, 1999.

[BBD01] L.C. Briand, C. Bunse, and J.W. Daly. A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. *IEEE Transactions on Software Engineering*, 27(6), jun 2001.

[BBDD97] L.C. Briand, C. Bunse, J.W. Daly, and C. Differding. An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. *Empirical Software Engineering. An International Journal*, 2(3), 1997.

[BBK78] B.W. Boehm, J.R. Brown, and J.R. Kaspar. *Characteristics of Software Quality.* TRW Series of Software Technology, Amsterdam, North Holland, 1978.

[BDW99] L. Briand, J. Daly, and J. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Softw. Engineering*, 25(1), jan/feb 1999.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[Bin99] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[BK95] J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proc. ACM Symposium on Software Reusability*, apr 1995.

[BMB+98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons; ISBN-0471197130, 1998.

[BMP87] D. Bell, I. Morrey, and J. Pugh. *Software Engineering - A Programming Approach.* Prentice-Hall, NJ, 1987.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 2 edition, 1994.

[BPP81] K.H. Britton, R.A. Parker, and D.L. Parnas. A Procedure for Designing Abstract Interfaces for Device Interface Modules. In *International Conference on Software Engineering (ICSE)*, 1981.

[BR88] V. Basili and D. Rombach. The TAME project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Softw. Engineering*, 14(6), jun 1988.

[BSB96] L.C. Briand, S.Morasca, and V.R. Basili. Property-Based Software Engineering Measurement. *IEEE Transactions on Softw. Engineering*, 22(1), jan 1996.

[Bud91] T. Budd. *Object-Oriented Programming.* Addison-Wesley, april 1991.

[Cas96] E. Casais. State-of-the-art in Re-engineering Methods. achievement report SOAMET-A1.3.1, FAMOOS, October 1996.

[Cas98] E. Casais. Re-Engineering Object-Oriented Legacy Systems. *Journal of Object-Oriented Programming*, pages 45–52, January 1998.

[CB88] G. Caldiera and V.R. Basili. Indetifying and Qualifying Reusable Software Components. *IEEE Computer*, 23(5), may 1988.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CC92] E.J. Chikofsky and J.H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[CCKT83] J. Chambers, W. Cleveland, B. Kleiner, and P. Tukey. *Graphical Methods for Data Analysis.* Wadsworth, 1983.

[CG90] D. Card and R. Glass. *Measure Software Design Quality.* Prentice-Hall, NJ, 1990.

[Chi01] C. Chirilă. Instrument Software pentru Detecţia Carenţelor de Proiectare în Sisteme Orientate-Obiect. Diploma Thesis (Advisor: R.Marinescu), "Politehnica" University Timişoara, 2001.

[Ciu99] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems - TOOLS 30, IEEE Computer Press*, 1999.

[Ciu01] O. Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen.* PhD thesis, Universität Karlsruhe, 2001.

[CK94] S.R. Chidamber and C.F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, London, 2 edition, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, London, 2 edition, 1991.

[CZ00] H. Cleve and A. Zeller. Finding Failure Causes through Automated Testing. In *Fourth International Workshop on Automated Debugging.* Muenich (Germany), 28-30 August 2000.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding Refactorings via Change Metrics. In *OOPSLA 2000 proceedings*, 2000.

[DeM82] T. DeMarco. *Controlling Software Projects; Management, Measurement and Estimation.* Yourdan Press, New Jersey, 1982.

[EL96] K. Erni and C. Lewerentz. Applying Design-Metrics to Object-Oriented Frameworks. In *Proceedings 3rd International Software Metrics Symposium*, pages 64–74, Los Alamitos, 1996. IEEE Computer Science Press.

[Ern96] K. Erni. *Anwendung multipler Metriken bei der Entwicklung objektorientierter Frameworks.* PhD thesis, Universität Karlsruhe, Mnster, 1996.

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, Massachusetts, 1990.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Fen94] N. Fenton. Software Measurement; A Necessary Scientific Base. *IEEE Transactions on Softw. Engineering*, 20(3), mar 1994.

[FN00] N.E. Fenton and M. Neil. Software metrics: A Roadmap. In *ICSE - Future of SE Track*, pages 357–370, 2000.

[FP97] N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thomson Computer Press, London, UK, second edition, 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[HCN98] R. Harrison, S.J. Counsell, and R.V. Nithi. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Softw. Engineering*, 24(6), jun 1998.

[Hoe54] P.G. Hoel. *Introduction to Mathematical Statistics.* Wiley, 1954.

[HS96] B. Henderson-Sellers. *Object-Oriented Metrics – Measures of Complexity.* Prentice-Hall,Sydney, 1996.

[ISO91] International Standard Organization ISO. ISO-9126 – Information Technology–Software Evaluation–Quality Characteristics and Guidelines for Their Use. *International Standard Organization, Brussels*, 1991.

[JF88] R.E. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[JR00] D. Jackson and M. Rinard. Software analysis: A roadmap. In *International Conference on Software Engineering*, 2000.

[Kos96] B. Kosko. *Fuzzy Engineering.* Prentice Hall, 1996.

[KPF95] B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*, 21(12), dec 1995.

[KW86] B.A. Kitchenham and J.G. Walker. The meaning of quality. In *Conference on Software Engineering, 1986*, 1986.

[Lad02] R. Laddad. I Want My AOP! *Java World*, (1), January 2002.

[Lak96] J. Lakos. *Large-Scale C++ Software Design.* Addison-Wesley, 1996.

[LB98] J. Wüst L. Briand, J. Daly. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(2), 1998.

[LD01] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *OOPSLA 2001 proceedings*, 2001.

[LH89] K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programming. *IEEE Software*, pages 38–48, September 1989.

[LH93] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. *IEEE Proc. First International Software Metrics Symp.*, pages 52–60, may 1993.

[Lis88] B. Liskov. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices*, 23(5), may 1988.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics.* Prentice-Hall Object-Oriented Series, Englewood Cliffs, NY, 1994.

[Lor93] M. Lorenz. *Object-Oriented Software Development: A Practical Guide.* Prentice-Hall, NJ, 1993.

[Mar96a] R.C. Martin. Interface Segregation Principle. *C++ Report*, 1996.

[Mar96b] R.C. Martin. Open-Closed Principle. *C++ Report*, 1996.

[Mar96c] R.C. Martin. The Liskov Substitution Principle. *C++ Report*, 1996.

[Mar97a] R. Marinescu. The Use of Software Metrics in the Design of Object-Oriented Systems. Diploma Thesis, "Politehnica" University Timişoara, 1997.

[Mar97b] R.C. Martin. Granularity. *C++ Report*, 1997.

[Mar97c] R.C. Martin. Stability. *C++ Report*, February 1997. An article about the interrelationships between large scale modules.

[Mar98] R. Marinescu. An Object Oriented Metrics Suite on Coupling. Master's thesis, "Politehnica" University Timisoara, 1998.

[Mar99] R. Marinescu. A Multi-Layered System of Metrics for the Measurement of Reuse by Inheritance. In *Proceedings of TOOLS Asia 1999*, pages 142–153. IEEE Computer Society, 1999.

[Mar00] R.C. Martin. Design Principles and Patterns. *Object Mentor, http://www.objectmentor.com*, 2000.

[Mar01] R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In *Proceedings of TOOLS USA 2001*, pages 103–116. IEEE Computer Society, 2001.

[McC76] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, dec 1976.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[Mey91] B. Meyer. Tools for a new culture – Lessons from the design of the Eiffel libraries. *Communications of the ACM*, (2), february 1991.

[Mih02] P. Mihancea. *The Meta-Architecture of the Detection Strategies Tuning Machine.* Technical Report at "Politehnica" University Timişoara, 2002.

[MRW77] J.A. McCall, P.G. Richards, and G.F. Walters. *Factors in Software Quality, Volume I.* NTIS AD/A-049 014, NTIS Springfield, VA, 1977.

[MWT94] H.A. Müller, K. Wong, and S.R. Tilley. Understanding Software Systems using Reverse Engineering Technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

[Opd92] W. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Par72] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), dec 1972.

[Pfl98] S.L. Pfleeger. *Software Engineering - Theory and Practice.* Prentice-Hall, NJ, 1998.

[Pid02] W. Pidcock. What is Meta-Modelling. *Metamodel.com, http://www.metamodel.com/metamodeling/*, 2002.

[PJ88] M. Page-Jones. *Practical Guide to Structured Systems Design (2nd Edition).* Prentice-Hall PTR, Englewood Cliffs, NY, 1988.

[PL99] K. Periyasamy and X. Liu. A New Metrics Set for Evaluating Testing Efforts for Object-Oriented Programs. In *TOOLS 30, IEEE Computer Society*, 1999.

[Rie96] A.J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[RL92] C. Rajaraman and M.R. Lyu. Some Coupling Measures for C++ Programs. In *Proceedings of TOOLS USA'92, Prentice-Hall, Englewood Cliffs, NJ*, 1992.

[Rus01] A. Rusu. Strategii de Detecţie a Carenţelor de Proiectare în Sisteme Orientate-Obiect. Diploma Thesis (Advisor: R.Marinescu), "Politehnica" University Timişoara, 2001.

[SB01] H.A. Sahraoui and M. Boukadoum. Extending Software Quality Predictive Models Using Domain Knowledge. In *5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2001.

[SBL01] H.A. Sahraoui, M. Boukadoum, and H. Lounis. Building Quality Estimation models with Fuzzy Threshold Values. *lObjet*, 17(4), 2001.

[Sch97] H. Schildt. *C++. Manual complet.* Editura TEORA, 1997.

[She90] M. Shepperd. Early Life-cycle Metrics and Software Quality Modules. *Inf. Software Technologies*, 32(4), 1990.

[Som95] I. Sommerville. *Software Engineering. Sixth Edition.* Addison-Wesley, 1995.

[Tel00] Telelogic. *Telelogic Tau Logiscope 5.1. - Audit Basic Concepts.* Telelogic AB, Malmoe, Sweden, 2000.

[Tri01] A. Trifu. Using Cluster Analysis in the Architecture Recovery of OO Legacy Systems. Diploma Thesis, Karlsruhe and the "Politehnica" University Timişoara, 2001.

[TS92] D.P. Tegarden and S.D. Sheetz. Object-oriented system complexity: an integrated model of structure and perceptions. In *OOPSLA'92 Workshop on Metrics for Object-Oriented Software Development (Washington DC)*, 1992.

[Wir95] N. Wirth. A Plea for Lean Software. *IEEE Computer*, 28(2), jan/feb 1995.

[Zel99] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of ESEC/FSE 99*, pages 253–267. Springer Verlag – Lecture Notes on Computer Science, 1999.

[Zus92] H. Zuse. Properties of Software Measures. *Software Quality Journal*, 1:225–260, 1992.

# Appendix A

# Factor-Strategy Quality Model for Maintainability

This appendix contains the full description of the Factor-Strategy quality model for maintainability proposed in Section 6.4.3. The language used to describe the model is very similar to the one used in [Tel00]. Factor-Strategy quality models described using this language can be evaluated using the PRODEOOS toolkit (Section 7.2.4).

First we see that the maintainability score is computed as an average between the scores of the four criteria that compose it in conformity with the definition of maintainability found in ISO9126[ISO91] (lines 01 and 02).

In the following lines (04 to 15) we see how the score for the Changeability criteria is computed: first, a raw score is computed as an average from the scores of the detection strategies that are related with the quality criteria (lines 04 to 09).After that, the raw score is mapped to a quality ranking, i.e. a normalized score obtained from a *matrix of ranks* (see lines 10 to 15). The score for the other three factor criteria (Analysability, Testability, Stability) are then computed in a similar manner (lines 17 to 50).

The score for each detection strategy is computed in a similar manner: first a raw score is computed mainly based on the number of suspects for the strategy. That score is than mapped to a normalized score based on one of the *matrix of ranks*(lines 54 to 74). Note that the same detection strategy can return a different score for different criteria based on the scoring mapper that was used. For example, the `DataClasses` detection strategy is considered to affect more severely the Changeability criteria and therefore its score is computed with a different score mapper (compare line 07 with 19, 31 and 42).

```
01    Maintainability := avg(Changeability, Analysability,
02                           Testability, Stability)
03
04    Changeability := avg (LackOfBridgeDeep(MediumScoring),
05        LackOfStrategy(SevereScoring), LackOfState(MediumScoring),
06        LackOfFacade(SevereScoring), GodMethod(MediumScoring),
07        DataClasses(SevereScoring), GodClasses(SevereScoring),
08        ShotgunSurgery(SevereScoring), GodPackage(SevereScoring),
09        ISPViolation(MediumScoring))
10    {
11           9   10  10,  /* EXCELLENT  */
12           7   9   8,   /* GOOD  */
13           5   7   6,   /* ACCEPTABLE */
14           0   5   4    /* POOR */
15    },
16
17    Analyzability := avg(LackOfStrategyGodLike(MediumScoring),
18        FeatureEnvy(TolerantScoring), GodMethod(MediumScoring),
19        DataClasses(MediumScoring), GodClasses(SevereScoring),
20        RefusedBequest(MediumScoring), MisplacedClass(TolerantScoring),
21        GodPackage(SevereScoring), TemporaryField(TolerantScoring),
22        ShotgunSurgeryWeighted(MediumScoring))
23    {
24           9   10  10,  /* EXCELLENT  */
25           7   9   8,   /* GOOD  */
26           5   7   6,   /* ACCEPTABLE */
27           0   5   4    /* POOR */
28    },
29
30    Testability := avg(FeatureEnvy(MediumScoring),
31        GodMethod(SevereScoring), DataClasses(MediumScoring),
32        GodClasses(SevereScoring), ShotgunSurgery(MediumScoring),
33        LackOfSingleton(MediumScoring))
34    {
35           9   10  10,  /* EXCELLENT  */
36           7   9   8,   /* GOOD  */
37           5   7   6,   /* ACCEPTABLE */
38           0   5   4    /* POOR */
39    },
40
41    Stability := avg (LackOfState(MediumScoring),
42        GodMethod(MediumScoring), DataClasses(MediumScoring),
43        GodClasses(SevereScoring), LackOfSingleton(MediumScoring),
44        ShotgunSurgeryWeighted(SevereScoring))
```

```
45    {
46            9    10   10,  /* EXCELLENT  */
47            7    9    8,   /* GOOD  */
48            5    7    6,   /* ACCEPTABLE */
49            0    5    4    /* POOR */
50    };
51
52    /* Matrix of Ranks for the Strategies*/
53
54    SevereScoring {
55      0    0    10,   /* EXCELLENT  */
56      1    1    9,    /* VERY GOOD  */
57      2    4    7,    /* GOOD  */
58      5    7    5,    /* ACCEPTABLE */
59      8    +oo  3     /* POOR */
60    },
61
62   MediumScoring {
63      0    1    10,   /* EXCELLENT  */
64      2    4    8,    /* GOOD  */
65      5    10   6,    /* ACCEPTABLE */
66      11   +oo  3     /* POOR */ `
67    },
68
69    TolerantScoring {
70      0    3    10,   /* EXCELLENT  */
71      4    8    8,    /* GOOD  */
72      9    12   6,    /* ACCEPTABLE */
73      13   +oo  4     /* POOR */
74    };
```

# Appendix B

# List of Further Detection Strategies

The purpose of this appendix is to supplement the collection of detection strategies defined in Chapter 5. For each class of design problems we briefly described several concrete flaws, but we defined detection strategies only for two flaws in each category. Therefore, in this appendix we want to provide the reader with the definitions for the other detection strategies. We will use a simplified (and slightly modified) form of the template defined in Section 4.4.1. Thus, for each detection strategy we provide the *literature source* for the design flaw that is being quantified by the strategy, the *detection rule*, and the *metrics* that are part of the detection rule.

## B.1 Data Classes

### Source

A. J. Riel. – *Object-Oriented Design Heuristics* [Rie96]

### Rule

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

### Metrics

1. **Weight of a Class (WOC)** [Mar01]
   - *Definition:* WOC is the number of non-accessor methods in the interface of the class divided by the total number of interface members.
   - *Implementation Details:* Inherited members are not counted. The members that belong to the interface of the class are the public, non-inherited methods and data members of a class.

2. **Number Of Public Attributes (NOPA)** [Mar01]
   - *Definition:* NOPA is defined as the number of non-inherited attributes that belong to the interface of a class.
3. **Number Of Accessor Methods (NOAM)** [Mar01]
   - *Definition:* NOAM is defined as the number of the non-inherited accessor-methods declared in the interface of a class.
   - *Implementation Details:* The big problem is how to identify the accessor-methods? We used the following "pattern": accessor-methods are small-methods, with a unitary cyclomatic complexity, and we rely on the name convention, stating that the names of accesor methods are prefixed with the `get` (or `Get`) and `set` (or `Set`) prefix.

## B.2 ISP Violation

### Source

R. Martin – *Interface Segregation Principle* [Mar96a]

### Rule

```
ISPViolation := ((CIW, TopValues(20%) butnotin (CIW, LowerThan(10)))
                and (AUF, LowerThan(0.5)) and (COC, HigherThan(3))
```

### Metrics

1. **Class Interface Width (CIW)**
   - *Definition:* CIW is defined as the number of members of a class that belong to the interface of the class.
   - *Implementation Details:* The members that belong to the interface of the class are the public, non-inherited methods and data members of a class.
2. **Clients Of Class (COC)**
   - *Definition:* The COC is defined as the number of classes that use the interface of the measured class.
   - *Implementation Details:* Inner classes are not counted. In the context of this metric a class A uses the interface of a class C if (at least) it calls a public method or accesses a public attribute of that class.
3. **Average Use of Interface (AUF)**
   - *Definition:* The AUF metric is defined as the average number of interface members of a class that are used by other classes.

- *Implementation Details:* AUF is computed by suming up the number of used members for each of the client-classes and divide it by the number of client classes (COC).

## B.3 Lack of State

### Source

E. Gamma, R. Helm, R. Johnson, J. Vlissides. – *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV94]. The *State* pattern.

### Rule

```
LackOfState := (AMW, HigherThan(4)) and (NOA, HigherThan(3))
               and ((WMC, HigherThan(10)) or (NPubM, HigherThan(3)))
```

### Metrics

1. **Average Method Weight (AMW)**

   - *Definition:* AMW is computed as the average cyclomatic complexity for the class. This is measured by dividing the *Weighted Method Count*(WMC) – computed based on McCabe cyclomatic complexity measure – value for the class, by the number of methods in the class. Thus,
   $$AMW = \frac{WMC}{NOM}$$
   - *Implementation Details:* An alternative for this metric would be to use another measure of method-complexity instead of McCabe's cyclomatic. The proposed metric is *Maximum Number Of Branches* (MNOB)[1]. This measure would be even closer to our detection goal.

2. **Number Of Attributes (NOA)**

   - *Definition:* NOA is the total number of attributes defined in a class.
   - *Implementation Details:* Inherited attributes should not be counted.

3. **Number Of Public Methods (NPubM)**

   - *Definition:* NPubM is the number of public methods defined in the measured class.
   - *Implementation Details:* Constructors and the destructor of the class should not be counted.

[1] For a definition of this metric please refer to Section 2.4.4

4. **Weighted Method Count (WMC)**[CK94]

   - *Definition:* WMC 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).
   - *Implementation Details:* We recommend the use of McCabe's cyclomatic number[McC76] for the quantification of method complexity.

## B.4 Lack of Visitor

### Source

E. Gamma, R. Helm, R. Johnson, J. Vlissides. – *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV94]. The *Visitor* pattern.

### Rule

```
LackOfVisitor := (AOR, HigherThan(0.5) and (NOD, HigherThan(3))
                 and (NPubM, HigherThan(5))
```

### Metrics

1. **Override Ratio (OR)**

   - *Definition:* The OR metric is computed between a base class and a child class, as the relative numbers of methods from the interface of the base class that are overriden in the derived class.
   - *Implementation Details:* Only the interface methods are counted (i.e. the public methods). Constructors and destructor of the base class are excluded.

2. **Average Override Ratio (AOR)**

   - *Definition:* The AOR metric is computed from the OR metric defined before as the average OR value of the base class
   - *Implementation Details:* For a base class C, let the set of the classes directly derived from it be $\{D_i | i = \overline{1, NOC}\}$, where $NOC$ is the number of child classes [CK94] for class C. Then, AOR is:

$$AOR = \frac{\sum_{i=1}^{NOC} OR(C, D_i)}{NOC}$$

3. **Number Of Descendants (NOD)**

   - *Definition:* NOD is the number of classes directly or indirectly derived from the measured class.

4. **Number Of Public Methods (NPubM)**
   - *Definition:* NPubM is the number of public methods defined in the measured class.
   - *Implementation Details:* Constructors and the destructor of the class should not be counted.

## B.5 Misplaced Class

### Source

This detection strategy is *home-grown* based on the principles of package cohesion found in: R. Martin – *Design Principles and Design Patterns* [Mar00]

### Rule

```
MisplacedClass := (CL, LowerThan(0.33) and ((NOED, TopValues(25%))
                  and (NOED, HigherThan(6))) and (DD, LowerThan(3))
```

### Metrics

1. **Number Of External Dependencies (NOED)**
   - *Definition:* NOED is the number of classes from other packages on which the measured class depends on.
   - *Implementation Details:* A class A depends on another class B, if class A calls methods and/or accesses attributes and/or extends class B.
2. **Class Locality (CL)**
   - *Definition:* CL is computed as the relative number of dependencies that a class has on its own package.
   - *Implementation Details:* In order to compute the metric we will divide the NOED value by the total number of classes on which the measured class depends on. Inner classes should not be counted.
3. **Dependency Dispersion (DD)**
   - *Definition:* DD is the number of other packages on which a class depends.
   - *Implementation Details:* The class depends on an package if it depends on of the classes from that package.

## B.6 Refused Bequest

### Source

M. Fowler – *Refactoring: Improving the Design of Existing Code* [FBB+99].

### Rule

```
RefusedBequest := ((AIUR, BottomValues(25%)) butnotin (DIT, LowerThan(1)))
                  and (AIUR, LowerThan(0.33)))
```

### Metrics

1. **Inheritance Usage Ratio (IUR)**

   - *Definition:* The IUR metric is a metric defined between a subclass and one its ancestor classes, as the relative number of inheritance-specific members from the ancestor class used in the derived class.
   - *Implementation Details:* A member of an ancestor class is an *inheritance-specific member* if its usage is related to inheritance. We identify following inheritance-specific members:
     - protected data members and methods[2]
     - non-private virtual methods

     The IUR metric is computed by counting the number of inheritance-specific members of the ancestor-class that are used in the subclass and divide it by the total number of inheritance-specific members from the ancestor. The *only* usages that are counted are: access of a protected data member, call of protected method and redefinition of a virtual method.

2. **Average Inheritance Usage Ratio (AIUR)**

   - *Definition:* We define AIUR for a derived class as the average value of the IUR metric computed between that class and all its ancestor classes.
   - *Implementation Details:* For a class C, let the set of its ancestor be $\{A_i | i = \overline{1, NOA}\}$, where $NOA$ is the number of ancestor classes for class C. Then, AIUR is:

     $$AIUR = \frac{\sum_{i=1}^{NOA} IUR(C, A_i)}{NOA}$$

     An alternative would be to consider only the direct inheritance relation.

[2] In JAVA protected members can be accessed not only from the subclasses, but also from any other class belonging to the same package. Thus, because there is no "inheritance-specific" access-specifier, counting the protected members might theoretically introduce some errors to the detection strategy. Yet, we believe that in practice the `protected` access-specifier is used mostly to provide an exclusive access on those members for the subclasses.

3. **Depth of Inheritance Tree (DIT)**[CK94]

   - *Definition:* The length of the inheritance chain from the root of inheritance tree to the measured class is the DIT metric for the class.
   - *Implementation Details:* In case involving multiple inheritance, the DIT will be the maximum length from measured class to the root of the tree.