Michele Lanza
Radu Marinescu

Object-Oriented Metrics in Practice

Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems

Foreword by Stéphane Ducasse

Springer

Object-Oriented Metrics in Practice

Michele Lanza · Radu Marinescu

Object-Oriented Metrics in Practice

Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems

Foreword by Stéphane Ducasse

With 80 Figures and 8 Tables

Springer

Authors

Michele Lanza

Faculty of Informatics
University of Lugano
Via G. Buffi 6
6900 Lugano
Switzerland
michele.lanza@unisi.ch

Radu Marinescu

"Politehnica" University of Timişoara
Department of Computer Science
LOOSE Research Group
Bvd. Vasile Pârvan 2
300223 Timişoara
Romania
radu.marinescu@cs.upt.ro

Library of Congress Control Number: 2006928322

ACM Computing Classification (1998): D.2.7, D.2.8, D.2.9

ISBN-10 3-540-24429-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-24429-5 Springer Berlin Heidelberg New York

Typeset by the authors
Production: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig
Cover design: KünkelLopka Werbeagentur, Heidelberg

Printed on acid-free paper 45/3100/YL - 5 4 3 2 1 0

For Marisa and Cristina

Acknowledgments

We would like to thank the following people.

Stéphane Ducasse, who was a challenging reviewer and gave many inspirations which helped making this book possible. Also, many thanks for providing a striking foreword. Oscar Nierstrasz, for the formidable and helpful review he provided. Thanks for being though as a rock, Oscar! Tudor Gîrba, for providing many, many insightful comments in a nice and constructive way. Thanks for being around, and spending so much time on this little tome. We would also like to thank Richard Gronback at Borland for providing positive comments on the value of our book.

We would like to thank Ralf Gerstner, our editor, and his team at Springer for following this book project with dedication and patience.

Our special thanks go to Prof. Gerhard Goos for his strong encouragement to start writing this book, and for remembering us that "every man must plant a tree, build a house and write a book".

We are grateful to Daniel Raţiu, Danny Dig, Petru Florin Mihancea, Adrian Trifu and Richard Wettel for revising with intelligence, enthusiasm and patience the various versions of our manuscript.

Finally we would like to thank our wives, Marisa and Cristina, for being so understanding — patience is truly a female trait!

Michele Lanza & Radu Marinescu

April, 2006

Foreword

Some Context

It is a great pleasure and difficult task to write the foreword of this book. I would like to start by setting out some context.

Everything started back in 1996 in the context of the IST project FAMOOS (Framework-Based Approach for Mastering Object-Oriented Software Evolution). At that time we started to think about patterns to help approach and maintain large and complex industrial applications. Some years later, in 2002, after a lot of rewriting these patterns ended up in our book *"Object-Oriented Reengineering Patterns"*. Back in 1999, Radu Marinescu was a young researcher on object-oriented metrics and Michele Lanza was starting to work on program visualization. At that time, object-oriented reengineering was nearly a new field that we explored with imagination and fun. While writing the *"Object-Oriented Reengineering Patterns"* book, we (Oscar Nierstrasz, Serge Demeyer and I) felt the need to have some metric-based patterns that would help us apply metrics to understand or spot problems in large applications, but we could not find the right form for doing it, so we dropped this important topics from our book.

A few years later, in the context of RELEASE Network, a European Science Foundation network, I remember talking with Radu, who was working on detection strategies, about a book that would have pattern metrics at its center. Such a book was then still missing. Now you can read about years of concrete experience in this book.

A Word About Design

Programming, and object-oriented programming in particular, is about defining an adequate vocabulary that will help express a complex problem in a much simpler way. While object-oriented design provides a good way to express new vocabularies, object-oriented design is *difficult*. Difficult because different concerns have to be taken into account: Is the vocabulary good enough? How will the terms interact with each other? Will the domain be extended? Can it be extended? Will the operations change? Can we know this upfront in our nice crystal ball? Are the entities representing the domain important enough to be first class entities? And many other concerns. We have some important conceptual tools for assessing the design of an application — experience, code heuristics, and design patterns are some of them — still Object-Oriented Design (in capitals) is difficult.

Over the years, I have programmed a lot and taught a lot of object-oriented design. Of course, not simply UML, which is a notation, but the identification of objects and their responsibilities, how these entities interact to gracefully achieve our complex tasks. Note that often people confuse the format with the contents, as XML marketing tends to demonstrate it.

The goal of my lectures is not that students learn some design patterns, but that the students train and educate their design taste. Maybe because

I'm French, I often use the metaphor of teaching cooking where, besides the technical aspects of slicing and cooking the elements, creativity comes into play because the cook knows tastes and spices and how they interact. To learn we should get in touch with varieties of spices, aromas and textures: we do not teach cooks by only feeding them with fast food, but by exposing them to varieties and subtle flavors. I always remember when I was a kid the first time I went to sleep in a friend's place. There things were the same but also different. I realized that we understand the world also by stressing and tasting differences. After being exposed to change, we can decide to explore or not, but at least this helps us to understand our own world. This is why I expose students to the beauty of Smalltalk. My goal is to destabilize them, so that they realize that "0.7 sin" (i.e., sin is just a message sent to a number) can be more natural than "Math.sin(0,7)", or that late binding is a big case statement at the virtual machine level. A nice example is to understand how Boolean behavior (NOT, AND, OR) is defined when we have *only* objects and not primitive types.

Recently I have been more and more involved in the maintenance and evolution of Squeak, this great open-source multimedia Smalltalk. I decided that I should help make this gem shine. And this has been rewarding since I have learned a lot. Squeak has given me many ideas about my own practices and has sharpened my taste and views about design, and often even changed my mind. Here are some of the thoughts I want to share with you:

(1) Reducing coupling is difficult. Often we would like to be able to load one package independently of others. But there is this one reference to that class that does not make it possible. Easy you think. Just move the class to another package. But you simply move the dependency around! If you are lucky you have dead code. If you can attach the changes as a class extension to another package you can fix it, but in Java and C++ you do not have that possibility, while the next version of C# is taking a step in that direction. In all the other non-trivial cases you have to understand the context and see if a registration mechanism or any other design change can solve the problem.

(2) It is really fun to see that the old procedural way of thinking is still with us. People still believe that a package should be cohesive and that it should be loosely coupled to the rest of the system. Of course strong coupling is a problem. But what is cohesion in the presence of late binding and frameworks? Maybe the packages I'm writing are transitively cohesive because the classes they contained extend framework classes defined in cohesive packages? Therefore naive assessments may be wrong.

(3) Evolution in general is difficult. Not really because of the technical difficulty of the changes but because of the users. The most difficult things I learned with Squeak is that on the one hand all the system and the world urge you to fix that specific behavior, it is easy to fix and the system and your ego would be better after. But the key questions are: How are the clients impacted? Is the change worth it? May be the design is good enough finally?

But what is "good enough"? On the other side, not changing is *not* the solution. Not changing is not really satisfactory because maybe with a slightly different vocabulary our problem would be so simple to express. In addition a used system *must* change. Therefore the next challenge is then how can we escape code sclerosis. How can we create a context in which changes are acceptable and possible and not a huge pain? The only way is to build a change-friendly context. One path to follow is investing in automated tests.

A Word About Metrics

Funny enough, I never believed that metrics could help in assessing design. Indeed, what metric can tell me when we should introduce a Visitor pattern. We could get an indication, for example, when the domain objects do not change over the years and when we want to plug in different algorithms acting on the domains. But, is it worth it? Is it worth it when you are using a language that supports class extension such as Objective-C or Smalltalk[1].

However on the other hand, when I was writing the object-oriented reengineering patterns, I was dreaming about small metric-based patterns that would help the reengineers to identify some structural problems, maybe not Design problems but still important problems and bad smells. Indeed, it would be wonderful to be able to use simple metrics and to know how to use them to identify code problems. And this is what Michele and Radu have succeeded in presenting in this book. So, after all, I have changed my mind regarding metrics.

I think that the contribution of this book is quite rich. Indeed what is fascinating is to see the amount of Java code tool analysis. The major problem with these tools is that of course they compute metrics — or what I would humbly call measurements with respect to metric experts. And we have tons and tons of metrics! We are overwhelmed by numbers and acronyms! But nearly none of the tools puts the metrics in context, or simply makes them confront each other. Of course this is difficult, but this is where the information or semantics is revealed. By putting metrics in context we pass from a quantitative and boring approach to a qualitative understanding. The great value of this book is to put metrics in perspective; it does this using two conceptual tools: the overview pyramid and the polymetric view.

The Overview Pyramid is really a simple and powerful tool to introduce some way to understand the metrics, to correlate them, and, by this simple fact, generate a deeper knowledge. It is well known that by mixing metrics we obtain meaningless results. Still the overview pyramid avoids this problem and uses ratios at the right level. The overview pyramid produces new insights about the code. It makes a big difference whether a package contains 1000 lines of code for 100 or 10 classes.

[1] In Smalltalk or Objective-C, a method does not have to be in the file or package of the class to which the method is attached. A package can define a method that will extend a class defined in another package.

I'm a bit biased when I talk about polymetric views since I love them. Polymetric views display structural entities and their relationships using some trivial algorithms. Then the entities are enriched with metrics. Once again, the metrics are put into a context. And from this perspective new knowledge emerges. It is worth mentioning that one of the powers of polymetric views is their simplicity. Indeed, researchers tend to focus on solving difficult problems, and some people confuse the complexity of problems with that of the solutions. I have always favored simple solutions when possible since they may have a chance to get through. Polymetric views have been designed to be simple so that engineers using different environments can implement them in one or two days. As an anecdote, an Apple engineer to whom we showed the polymetric views one evening showed us the next morning that he had introduced some of them in his environment. This was delightful.

I hope that in the future metrics tools will introduce the overview pyramid and that reengineers will use the power of polymetric views.

This book goes a step further: It also introduces a systematic way of detecting bad smells by defining detection strategies. Basically a detection strategy is a query on code entities that identifies potential bad smells and *structural* design problems. Now there are two dangers: first there is the danger of thinking that because your code does not exhibit some of these bad smells you are safe; and second there is the danger of thinking the inverse. Indeed, the authors measure and reveal *structural* aspects of the program and not its Design[2]. While this may be true that if the structure of an application is bad, its design can have problems — there is no systematic way of measuring the design of an application. Of course, in trivial cases (i.e., when a system is distorted according to bad practices) structural measurements will reveal flaws; but in the case of well-designed systems that have evolved over time, this is another story.

Therefore it is important to see the suggested refactorings as the preliminary step to further and more consequent analysis and action. But this is an important step. This is like removing the furniture of a room before renovating it — once you removed it you can see the wall that you should fix. Thus, just because the suggested refactorings are applied and the proposed detection strategies do not detect anything does not mean that the problem is not there, but you are in a much better position moving forward.

So, for all the reasons I've mentioned, I'm convinced — and I guess that you see that I'm not an easy guy to convince — that this book will really help you to deal with your large applications.

Université de Savoie, April 2006 Stéphane Ducasse

[2] You remember, with a capital "D".

Contents

1

Introduction

This book is not about metrics *per se*. It is about the way metrics can be used *in practice* to aid us in characterizing software systems, to evaluate their design and when we detect design problems to provide the appropriate refactorings.

The goal of this book is to help you characterize, evaluate and improve the design of the large applications that you have to maintain and enhance, by using metrics and visualization techniques to localize potential structural design problems and identify context-dependent recovery means.

Why are these relevant problems? Well, if a straightforward and simple software engineering solution to build perfect and extensible applications existed, any software engineer would know it and you would not be reading this book. Designing large applications is difficult because of the intrinsic complexity of the modelled domains. To this intrinsic complexity, another incidental factor is added, which comes from business processes, organizational issues, human and other external factors.

Based on centuries of development, engineers have built – with success – extremely complex artifacts, such as bridges, buildings, satellites, space shuttles, etc. In software engineering, many development methodologies and design heuristics [Rie96] have been proposed in recent decades to help software engineers to produce robust and extensible software. Some methodologies promoted up-front design such as the now obsolete waterfall model. The spiral model acknowledged this fact and proposes a more flexible model of development adapted to changes [Boe88]. More recently, agile methodologies acknowledged the fact that there is no way to predict future require-

ments and that the only way to survive is to embrace change as a fact of the software industry [Bec00, Kru04].

Over the years a common understanding of basic software engineering principles emerged. In the case of object-oriented programming and design good examples are the Open-Closed Principle [Mey88a], the Law of Demeter [Lie96], the Substitution Principle [MGM02, LW93b], Responsibility-Driven Design [WBW89], etc. Moreover, nowadays most software engineers understand and use design patterns [GHJV95, ABW98], write unit tests and use refactorings [FBB+99].

There are several factors that make designing and developing large software systems a difficult task:

- Designing software is about abstraction manipulation, i.e., a software system has no tangible, physical form. Therefore it is more complex to assess whether the outcome is the desired one.
- Writing software involves human communication that becomes more and more complex the more people are involved, and even has an effect on the structure of the system itself, as stated in Conway's law: "Organizations that design systems are constrained to produce designs that are copies of the communication structures of these organizations".
- Software *must evolve and change to be successful*. Indeed, due to the inherent complexity of design, the necessity of meeting new client requirements adds stress to software systems over their life-time. A design often degrades and gets more and more complex and thus harder to evolve [DDN02]. Lehman and Belady established laws that describe the inevitable evolution of successful projects [LB85]. These laws stress the fact that software *must* continuously evolve to stay useful and that this evolution is accompanied by an increase in complexity and that energy will have to be invested to control this growth. This *continuous evolution* implies that a design is not written in stone but must be revised and improved over and over again to fight against the effects of *aging and decay* that software systems inevitably incur.

An important question is then how can we control complexity. There is no definite answer but often a combination of actions such as redesigning, rethinking and adapting the architecture, modularizing the application, etc. [DDN02, Fea05]. What would be ideal would be to have analyses or predictions telling us which parts of the system should be changed to support the system evolution. Many re-

searchers are working on capturing software quality in terms of metrics. Up to now, no magic metric has been found and we can consider the definition of a universal design quality metric as the holy grail of software engineering.

Still, metrics can be a useful tool to understand, steer and control the development of complex software applications. Software metrics can be used to understand applications, to get an overview of a large system and identify *potential* design problems. In this book we propose the use of metrics to get an overview of large applications using source code metrics and to identify potential design problems by combining source code metrics.

Metrics — a Swiss Army knife for software design?

> *Good design quality metrics are not necessarily indicative of good designs. Likewise, bad design quality metrics are not necessarily indicative of bad designs.* [JDepend Docs]

It is important not to be blinded by metrics. Metrics are not a panacea. Metrics are a tool with power but also with limits. There are many aspects of design and its quality that are difficult to measure. We offer an extensive set of ways to measure design, but of course these ways are based on structural element characterization, i.e., source code and design assessment cannot be reduced to measuring the structure of an application. However, metrics help in spotting problems that an expert can use as a starting point for a deeper analysis.

In addition, within your organization you certainly have access to other information that you can measure such as the number of changes, the number of bug fixes associated with a particular artefact, the amount of time spent to develop a part of the system, and so on. Following our approach you can certainly gain insights into your applications by developing your own *detection strategies*.

Why Should You Read This Book?

We wrote this book for all the software engineers concretely facing legacy systems in their daily work. The ideal reader we have in mind is a fluent programmer who has to maintain and evolve code he did not write, or a consultant who has to assess large applications.

While many books exist to help designing existing applications, there is no book that supports the understanding of applications and the identification of potential design problems in a scalable manner.

It is rare to have the possibility to design a new system from scratch. Most of the time engineers are assigned to maintain and evolve existing successful applications. The term used in such a case is *legacy system*. Such systems suffer from typical problems, such as obsolete documentation, convoluted design, intensive patch mechanisms, large size, severe duplication, obsolete parts, long build times, loss of original developers, etc. [Fea05, DDN02].

The key characteristics of legacy systems, besides their important financial value, is their *complexity*, which stems from the fact that they were created and evolved by many developers. For example, the software system which helps a Jumbo Jet fly consists of ca. 8,000,000 (8 million!) lines of code [Coo99].

The central question is how can we deal with the size of legacy systems. Academic researchers has proposed numerous approaches, two of which have proven to scale up and be successful, *metrics* and *visualization*.

Metrics. Metrics are good at summarizing particular aspects of things and detecting outliers in large amounts of data. They scale up and are a good basis for synthesizing the many details of software.

Visualization. Humans are trained to understand signs and pictures, therefore visualization is an excellent tool for understanding and identifying hidden aspects of large software.

In this book we use and combine metrics and visualization to offer efficient approaches to understanding and assessing large applications. This book should help you to develop strategies for approaching, assessing and supporting the evolution and maintenance of large applications.

How Does This Book Fit into Your Library?

Several good books have emerged over recent years, and next we present a short and non-exhaustive list. While none of them covers the objective of this book, they are connected to the exact purpose of this book, i.e., to characterizing, evaluating and improving existing code design.

Software metrics books. There are also many good books on software metrics. Lorenz and Kidd [LK94] present a first attempt to use simple metrics to qualify the design of applications. However, the result is fairly outdated and overly simplistic. There

are other, more technical books, such as the one by Henderson-Sellers [HS96] which offers an extensive survey of metrics, or the one by Fenton [FP96] which lays down the foundation of measurement. These books, while valuable to researchers, are not oriented towards software engineers daily facing large applications. Our book offers directly applicable methods for practitioners.

Object-Oriented Design Heuristics. The book by Riel [Rie96] offers a large repository of concise, yet effective, design heuristics, guidelines and best practices for achieving good object-oriented design. In this book we use many of these heuristics as a starting point for the detection of relevant aspects of good (and bad) object-oriented design, aspects that need to be quantified using metrics.

Reengineering Books. The book "Object-Oriented Reengineering" by Demeyer et al. [DDN02] identifies a large set of patterns that describe best practices to reengineer a legacy system. This book covers the complete life cycle of approaching, understanding, testing, and restructuring legacy systems. However, it does not provide a systematic (quantifiable) way to assess the quality of an application. Another relevant book is "Working Effectively with Legacy Code" by Feathers [Fea05], which offers strategies for working more effectively with large, untested legacy code bases.

Design Patterns. The seminal book on Design Patterns [GHJV95] of the "gang of four" inspired many good books, such as [ABW98]. However, even though they offer solutions to design problems, they do not support the identification of the problems in massive amount of data that large legacy systems consist of.

Refactorings and Code Smells. The book by Fowler [FBB+99] presents refactorings, behavior-preserving code transformations that can help you to improve your design. In addition, this book presents code smells, factors that indicate bad practices or indicate that the design is starting to get rusty. While enlightening and provocative, the book does not offer a systematic way of identifying possible bad smells. Another excellent book is "Refactoring to Patterns" by Kerievsky [Ker04], which suggests that using patterns to improve an existing design is better than using patterns early in a new design.

Object-Oriented Metrics in Practice is the glue between all these books: It supports the identification of bad smells using software metrics, links back to refactorings or to reengineering patterns as possible solutions, and enforces object-oriented design heuristics.

The Book in a Nutshell

The approaches presented in this book are useful in dealing with the problems of existing legacy systems or when you need to ameliorate the design of a part of an existing system.

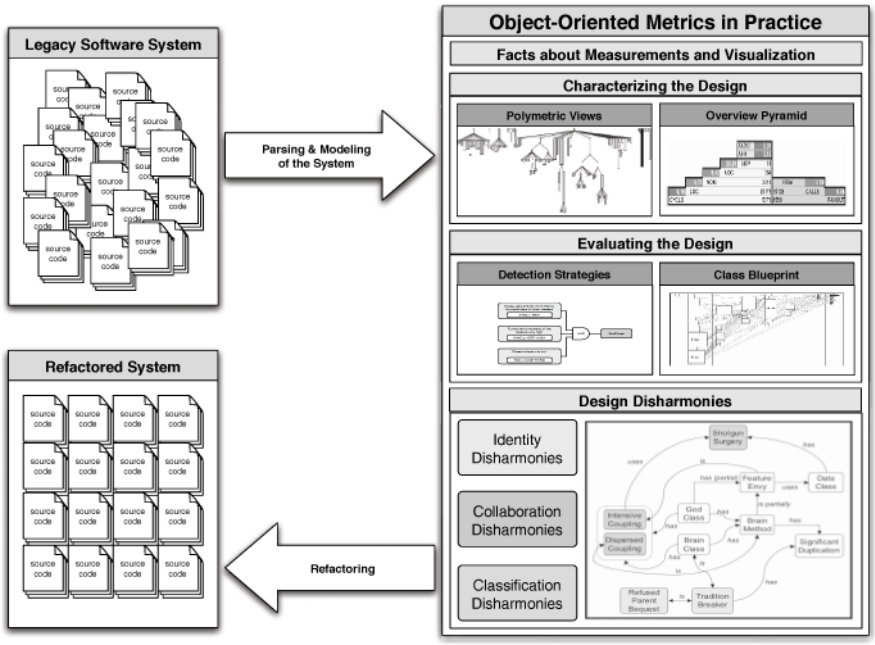


Fig. 1.1. Object-Oriented Metrics in Practice — in a Nutshell.

In Fig. 1.1 we see a depiction of our approach. It starts with a system whose design must first be characterized and then evaluated. This information is necessary to perform refactorings on the system to ameliorate its design.

Chapter 2, *Facts on Measurements and Visualization* , presents a practical view on metrics and the usual pitfalls of their use and how we circumvent them in this book. This chapter puts down the basic principles and vocabulary that is used throughout this book and also introduces the domain of visualization.

Chapter 3, *Characterizing the Design* , presents two metrics-based techniques, the *Overview Pyramid* and *Polymetric Views*, to get an overview of the design of a large software system. The *Overview Pyramid* assembles in one place the most significant measurements about

an object-oriented system, so that an engineer can *see* and *interpret* in one shot everything that is needed to get a first impression about the system. It provides an overview of the application in terms of its complexity, coupling and inheritance. *Polymetric Views* are metrics-enriched visualizations of software entities and their relationships. Their main benefit is that they can visually render numbers in a simple, yet effective and highly condensed way that is directly interpretable by the viewer.

Chapter 4, *Evaluating the Design* , presents two further techniques, i.e., the *Detection Strategy* and the *Class Blueprint* to provide more fine-grained understanding and assessment of the design of an application. *Detection strategies* are queries, expressed as a combination of metrics, identifying design elements in the source code satisfying the properties encoded by the query. They provide us with a means to detect flawed (from a design point of view) entities. A *Class Blueprint* is a semantically enriched and layered visualization of the control-flow and access structure of classes. It provides us with a powerful means to inspect the suspects detected by the *Detection Strategy*.

The following three chapters (*Identity Disharmonies* , *Collaboration Disharmonies* , and *Classification Disharmonies*) present a catalogue of *design disharmonies*. Each chapter presents general design rules for the design of classes, their collaboration and their position in inheritance hierarchy. These rules offer a unified way of approaching design through three general viewpoints: proportion (i.e., the size of the entity), presentation (i.e., how the entity is accessed or the presentation it offers to collaborators), and its implementation (i.e., the relationships between its internal representation). Each chapter then presents in detail *design disharmonies* which can be detected using metrics. For each of them we describe a metrics-based rule, i.e., a *Detection Strategy*, that would discover it in the code, analyze and discuss in detail an example using also the *Class Blueprint* visualization technique, and propose a set of potential cures in terms of refactorings.

About the Metrics

In this book we do not focus on one particular programming language. We exemplify the metrics using one or other language, but the metrics remain as language independent as possible. Therefore the vocabulary used throughout this book may not perfectly match that used by software engineers working with a specific language.

In the context of describing measurements we think of classes and operations as design entities that have properties and are in relation with other entities. This perspective helps us to define almost every measurement for a design entity in the very simple terms of the following three elements:

1. The *Having* Element, i.e., what other entities does the measured entity *have (contain)*, in the sense of being a scope for those entities? This also includes the inverse relation: which entity does the measured entity *belong to*? For example, an operation *has* parameters and local variables, while it *belongs to* a class.
2. The *Using* Element, i.e., what entities does the measured entity *use*; and again the inverse relation: by which entities is the measured one *being used*? For example, an operation is *using* the variables that it accesses, while it *is used by* the other operations that call (invoke) it. A class uses another class by extending it through inheritance, but also uses other classes by communicating with them.
3. The *Being* Element, i.e., what are the properties of the measured entity? For example, a property of a class is that it is abstract, while an attribute can have the property of being "private".

These three elements, as trivial as they may seem, open a huge range of possibilities for measurements, and one should be careful not to fall into the trap of measuring for the sake of measuring. Metrics are only a means, not a goal, and our goal is to evaluate and improve the design of a system using metrics.

A detailed description of this manner of expressing object-oriented metrics is found in Appendix A. There we also describe all the metrics used in this book in terms of the three aforementioned elements.

About the Examples

We applied the presented material on many proprietary projects. However, as we could not use these private industrial applications as examples, we chose to use an open-source project, *ArgoUML* (a well-known UML modeling application[1]) to illustrate our approaches.

ArgoUML consists of more than 200,000 raw lines (including comments, dead pieces, etc.), and slightly less than 100,000 lines of Java source code. We also use some code samples of other open-source projects written in Smalltalk to show that our approach is language independent.

[1] See http://argouml.tigris.org

Tool Support

Our approaches are powerful, but in order to make use of their full potential an adequate tool support is needed. From this point of view there are three possibilities:

1. Our own tool implementations (see Appendix B and Appendix C) which are free and (partially) open-source. Although developed in an academic environment we have oftentimes used them in large-scale industrial applications. We cordially invite you to try them out.
2. We hope that industrial tool builders will be influenced by this book and adapt their own tools.
3. As the ideas behind the tools are not complex, building your own tools based on our ideas is far less complex than you may guess.

The Stage is set

This book presents a hopefully fresh view on software metrics. You will discover that they are indeed useful for characterizing large software systems beyond the usual lines of code. Moreover, with some more sophisticated approaches you will learn how metrics can help to detect design problems in large systems and how a correct identification can also lead to proper refactoring solutions.

Enter metrics...

2

Facts on Measurements and Visualization

In this chapter we briefly introduce you to the good, the bad and the ugly of software metrics. In this context, we also take a short look on why and how visualization can be used in conjunction with metrics to counter-balance several drawbacks of using metrics. By doing this we aim to set a basis for our approach of employing metrics to characterize, evaluate and improve the design of software systems.

What is a metric? It is the mapping of a particular characteristic of a measured entity to a numerical value. An entity can be anything, including yourself; the characteristic can be anything, e.g., your height. The metric *height* in your case, for example, would be 180 cm. The metric could also have been 1.8 m. This seemingly trivial issue actually unravels a space where decisions have to be taken: what is the unit we are using? Is it important? Yes, otherwise you could end up being a giant of 180 meters! Moreover, why do we care at all about your height? Maybe we just wanted to measure your weight – and this leads us to the next issue: we can measure almost everything, but if we do not have a clear goal in mind of what we are actually trying to achieve with these measurements we are wasting our time. Since this is a book about object-oriented construction and design, we are quantifying and qualifying those aspects.

Why is it useful to measure? Engineering artifacts are made according to precise guidelines, i.e., the size, weight, material, etc. of screws, construction elements, etc. must be defined upfront and be respected by those actually creating the artifacts. Metrics in this case are a way to *control* quality. Losing control in such a case may have implications on security and potentially endanger people. In software engineering it is important and useful to measure systems, otherwise we risk losing control because of their complexity. Losing control in

such a case could make us ignore the fact that certain parts of the system grow abnormally or have a bad quality, e.g., cryptic and uncommented code, badly structured code, etc..

In practice there are different types of metrics that quantify various aspects of software development ranging from human resources to bugs and documentation. Consequently, metrics are used by developers, team leaders, and project managers, for many specific purposes, like quantifying and qualifying the code that has been written, or predicting future development efforts that must be invested into a project. Software metrics can be divided into two major groups [LK94]:

1. *Project metrics.* They deal with the dynamics of a project, with what it takes to get to a certain point in the development life cycle and how to know you are there. They can be used in a predictive manner, e.g., to estimate staffing requirements. Being at a higher level of abstraction, they are less prescriptive, but are more important from an overall project perspective.
2. *Design metrics.* These metrics are used to assess the size and in some cases the quality, size and complexity of software. They look at the quality of the project's design at a particular point in the development cycle. Design metrics tend to be more locally focused and more specific, thereby allowing them to be used effectively to directly examine and improve the quality of the product's components.

With respect to this classification, this book is dealing exclusively with design metrics.

The most pragmatic issue is how to use metric values so that they provide real information and not just numbers. In this context, the *Goal–Question–Metric* (GQM) model [BR88] defines the necessary obligations for setting objectives before embarking on any software measurement activity.

1. List the major *Goals* for which metrics are going to be employed.
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 to answer the questions.

The goal indicates the purpose of 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 necessary to answer a single question; likewise, a single measurement may apply to more than one question. In the rest of this book, we implicitly use GQMto efficiently employ metrics for characterizing and evaluating the design structure of object-oriented systems.

2.1 Metrics and Thresholds

With any metric we use we must know what is too high or too low, too much or too little. In other words, we need some *reference points*, some means to link a particular metric value to useful semantics. Therefore we discuss next how to identify threshold values so that metric values can be properly interpreted.

A crucial factor in working with metrics is to be able to interpret values correctly; and for this purpose we need to set *thresholds* for most of the metrics that we use. A threshold divides the space of a metric value into regions; depending on the region a metric value is in, we can make an informed assessment about the measured entity.

For example, if we measure the height of people and we define 2 meters as being the threshold to *very tall* people, then all measured people whose height is above that threshold can be qualified as being *very tall*. This simple example has a few implications: how did we come up with a threshold of 2 meters in the first place? Why not 1.95m? Why not six feet? And, is a person of 2.02 meters not small compared to a person of 2.5 meters? Would such a threshold still be meaningful in a population where the tallest person is 1.8 meters?

The point is that there is no such thing as a *perfect* threshold. However, we can still define *explicable* thresholds, i.e., values that can be chosen based on reasonable arguments. They are not perfect, but they are useful *in practice*, and this makes them good enough for our purposes, i.e., assess software artifacts. How do we find them? In our practical experience in working with metrics, we identified two major sources for threshold values:

1. *Statistical information*, i.e., thresholds based on statistical measurements. They are especially useful for size metrics, where only statistics can tell what usual or unusual values are. For example, if we measure (count) the number of hairs on the head of a person (say 10,000) and we want to assess if the result is low, average or high, we need one or more reference points, i.e., thresholds

which split the space of numbers into meaningful intervals. There is no other way of finding out than using statistical data, which in this case would tell us that the average number of hairs (measured over a statistically relevant population) is between 80,000 and 120,000. These two statistically-determined values help us determine if a person has an excessive pilosity or if it tends to become bald.

2. *Generally accepted semantics*, i.e., thresholds that are based on information which is considered common, widely accepted knowledge. Usually this knowledge is also a result of former statistical observations, but the information is so widely accepted that it implicitly provides the necessary reference points needed to classify measurement results. For example, if we were to measure the number of meals a person consumes per day, then we would use a value of 3 as a "normality" threshold, as usually people eat three times a day.

Statistics-Based Thresholds

What is the average number of operations (methods) per class? Beyond which number of code lines is a method too large? It is difficult to give a correct answer. On the one hand, the answer depends on many factors (i.e., how exactly do I count? what programming language was used? etc.). On the other hand, even after having specified all the measurement conditions we still need statistical data that provide us with proper orientation points (i.e., what is too much? what is too little?).

We come up with statistics-based thresholds by measuring a large number of Java and C++systems with respect to 3 metrics:

1. Average *Number of Methods (NOM)* per class
2. Average *Lines of Code (LOC)* per method (operation)
3. Average *Cyclomatic Number (CYCLO)* per line of code (i.e., density of branching points)

These three metrics have three important characteristics, which makes the gathering of statistical data for them meaningful:

1. they are elementary metrics that address the key issues of a project's size and complexity;
2. they are independent of each other;
3. they are independent of the size of a project.

We collected these metrics from a statistical base of 45 Java projects and 37 C++projects. The projects had been chosen with *diversity* in mind. They have various sizes (from 20,000 up to 2,000,000 lines), they come from various application domains, and we included both open-source and industrial (commercial software) systems.

Having this amount of data, we employed simple statistical techniques in order to determine for each of these metrics:

- the *Typical values*, i.e., the range of values that includes the data from most projects.
- the *Lower* and respectively the *Higher* margins of this interval.
- the *Extreme high values*, i.e., a value beyond which a value can be considered an outlier.

We use two statistical means to find what the typical high and low values are:

1. *Average* (AVG), to determine the most typical value of the data set (i.e., the central tendency).
2. *Standard deviation* (STDEV), to get a measure of how much the values in the data set are *spread*[1].

Knowing the AVG and STDEV values and assuming a *normal distribution* for the collected data (i.e., that most values are concentrated in the middle rather than the margins of the data set), we also know the two margins of the *typical values* interval for a metric[2] and the threshold for very high values. These are:

- *Lower margin*: $AVG - STDEV$.
- *Higher margin*: $AVG + STDEV$.
- *Very high*: $(AVG + STDEV) \cdot 1.5$, i.e., we consider a value to be very high if it is 50% higher than the threshold for a high value.

The computed threshold values are summarized in Table 2.1. These margins tell us now the meaning of *Low*, *High* and *Very High* for a

[1] The standard deviation is defined as the square root of the variance. This means it is the root mean square (RMS) deviation from the average. It is defined this way in order to give us a measure of dispersion that is (1) a non-negative number, and (2) has the same units as the data. For example, if the data are distance measurements in meters, the standard deviation will also be measured in meters.

[2] If the distribution of the data set is normal around 70% of the values will be in this interval.

given metric. Based on the information from Table 2.1 we can state that a Java method is *very long* if it has at least 20 LOC, or that a C++class has *few methods* if it has between 4 and 9 methods.

	Java				C++			
Metric	Low	Average	High	Very High	Low	Average	High	Very High
CYCLO/Line of Code	0.16	0.20	0.24	0.36	0.20	0.25	0.30	0.45
LOC/Method	7	10	13	19.5	5	10	16	24
NOM/Class	4	7	10	15	4	9	15	22.5

Table 2.1. Statistical thresholds of 45 Java and 37 C++systems computed for the size and complexity metrics used in this book.

The thresholds values presented above are relevant for more than the three metrics themselves; they can be used to derive thresholds for any metric that can be expressed in terms of these three metrics.

Example. We want to know what a *high* WMC (Weighted Method Count) value is for a class written in Java. We use the following definition of WMC [CK94]: the sum of the CYCLO metric [McC76] over all methods of a class. Thus, WMC can be expressed in terms of the three metrics as follows:

$$WMC = \frac{CYCLO}{LOC} \cdot \frac{LOC}{Method} \cdot \frac{NOM}{Class}$$

To compute a threshold for *high* WMC means selecting from Table 2.1 the *high* statistical values for the three primary terms from the formula above and multiplying them. In a similar fashion we can compute the *low*, *average*, *high*, and *very high* thresholds for two other size and complexity metrics used in this book, i.e., LOC/Class and AMW (Average Method Weight) *a.k.a.* CYCLO/Method (see Table 2.2).

	Java				C++			
Metric	Low	Average	High	Very High	Low	Average	High	Very High
WMC	5	14	31	47	4	23	72	108
AMW	1.1	2.0	3.1	4.7	1.0	2.5	4.8	7.0
LOC/Class	28	70	130	195	20	90	240	360
NOM/Class	4	7	10	15	4	9	15	23

Table 2.2. Derived thresholds of 45 Java and 37 C++systems computed for the size and complexity metrics used in this book.

Meaningful Thresholds

Statistics-based thresholds are useful for most metrics, but for some others they are implicitly given by observations. In that sense they are *also* based on statistics, but their values have become part of our culture. Therefore we do not need to statistically measure them, but we can infer them from common knowledge.

Example. If we think about the maximum nesting level of statements in a method it is clear that 0 denotes a method without any conditional statements and 1, 2 or 3 would mean that there is some nesting but it is quite shallow; but if the maximum nesting level gets higher than that we know that the method has a deep nesting level and following the control flow is harder.

We identified two cases of thresholds based on meanings that are generally accepted and easy to understand: (1) *commonly-used fraction thresholds* and (2) *thresholds with generally-accepted meaning*.

Common Fraction Thresholds

Quiz. Which of the following (fractional) numbers can you mentally associate with a semantic: 0.07; 0.39; 0.75; 0.33; 0.72?

We guess you picked 0.75 because it means *three quarters*; and you also picked 0.33 because it is *one third*. We guess that while looking at 0.72 you thought: "it is close to three quarters". Normalized metrics thus have thresholds which seem natural to us. We summarized them in Table 2.3.

Numeric Value	Semantic Label
0.25	ONE-QUARTER
0.33	ONE-THIRD
0.5	HALF
0.66	TWO-THIRDS
0.75	THREE-QUARTERS

Table 2.3. Threshold values for normalized metrics and their semantic labels.

We recommend that when you use metrics with normalized values you should choose thresholds from this discrete set of values as they can be easily linked to proper semantics.

Example. The metric TCC [BK95] (Tight Class Cohesion) is defined as "the relative number of method pairs of a class that access at least one common attribute of that class". This is a normalized metric, its

values lie between 0 and 1. The lower the TCC value, the less cohesive is the class: if we want to find non-cohesive classes we pick a value lower than *half*, i.e., either *one-third* or *one-quarter*.

Thresholds with Generally-Accepted Meaning

Not just fractional values can be associated with generally accepted semantics, but also absolute values. We consider integer values between 0 and 7, which was proven to be the upper limit of the human *short-term memory* [Pin97]. We summarized them in Table 2.4.

Numeric Value	Semantic Label
0	NONE
1	ONE/SHALLOW
2 – 5	TWO, THREE/FEW/ SEVERAL
7 – 8	Short Memory Capacity

Table 2.4. Threshold values associable with generally accepted semantics.

Note that these thresholds are used throughout this book, but this does not make them generally applicable to other contexts.

Example. We use such thresholds for the ATFD (Access To Foreign Data) metric, which counts how many attributes from other classes are accessed directly from a measured class. If we wanted to judge "by the book" then every class with $ATFD > NONE$ is problematic. But if we wanted to admit that accessing *accidentally* a couple of attributes of other classes is not *that* much of a problem, we can classify as critical only those classes that have $ATFD > FEW$.

2.2 Visualizing Metrics and Design

Characterizing, evaluating and improving the design of large-scale system is a highly complex enterprise, and while metrics are a highly needed means for this purpose, they must be used in conjunction with further techniques to handle this level of complexity. In our opinion the most adequate means to complement metrics is visualization, as it has long been adopted as a means to break down the complexity of information.

The goal of visualization in general is to *visualize any kind of data*. Applications in visualization are so frequent and common, that most people do not notice them: examples include meteorology (weather

maps), geography (street maps), geology, medicine (computer-aided displays to show the inner of the human body), transportation (train tables and metro maps), etc..

Metrics and Visualization

Quiz. Which of the cylinders in Fig. 2.1 have which letter associated with it?

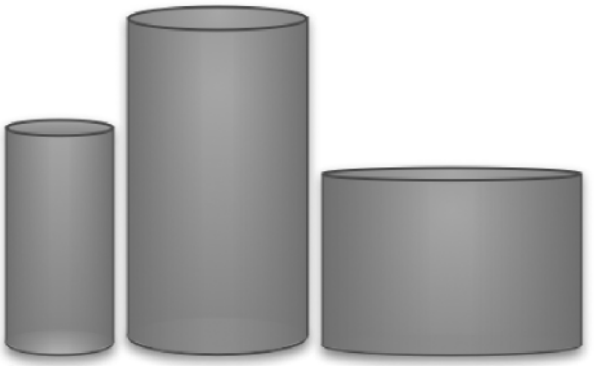


Fig. 2.1. A simple visualization of metrics.

It is easy to assess that the cylinder on the right has the largest diameter, the one in the middle has the greatest height, while the one on the left has the smallest diameter. Why is that? Human perception allows us to perform such non-trivial analysis as an in-grained mechanism, despite the fact that we had no numbers to hand. However, when provided with a table containing metric information (height, diameter, weight) for the cylinders we have no problem assigning those numbers. Do we? There is a problem with the weight metric which confuses us. Why? It does not respect the so-called *representation condition*.

In measurement theory, the procedure of rendering metrics on visual characteristics of representations is called *measurement mapping*, and must fulfill the representation condition, which asserts that "a measurement mapping M must map entities into numbers and

empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations" [FP96]. In other words, if a number a is greater than a number b, the graphical representations of a and b must preserve this fact.

The reader must be aware that visualization does not provide a means to visualize every metric. The provided weight metrics above actually confuse us because we would think that the smallest cylinder would also be the lightest. In that sense, at least regarding the weight, the above visualization does not completely respect the representation condition.

Software Visualization in a Nutshell

Software visualization is defined as "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software" [SDBP98]. It is a specialization of *information visualization*, whose goal is to visualize any kind of abstract data.

The field of software visualization can be divided into two separate areas [SDBP98]: program visualization and algorithm visualization. In this book we focus exclusively on techniques of the first area, as our aim is to improve the efficiency of metrics for assessing and improving the design quality of software systems.

Program visualization techniques provide views of actual program code or data structures in either static or dynamic form. The visualization techniques presented in this book belong to a sub-area of program visualization, namely *static code visualization*, because we visualize (object-oriented) source code by using only information which can be *statically* extracted from the source code without the need to actually run the system.

Static code visualization has been widely used by the reverse engineering research community for the past two decades [SDBP98, SWM97, Sto98, MADSM01]. Many of these visualization techniques provide ways to uncover and utilize information about software systems. When it comes to visualizing code, the preferred way of doing so is by using the notion of a graph, where the vertices represent the entities (such as classes, methods, etc.) and the arcs represent the relationships (such as invocation, inheritance, etc.). In this book we use a special technique called *polymetric view* to visualize the software artifacts that we analyze using metrics.

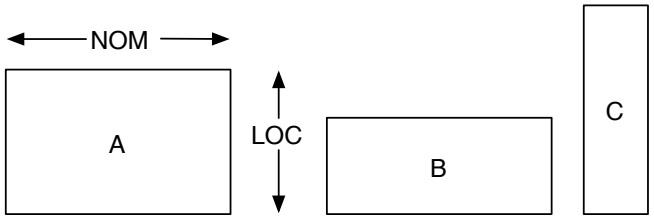


Fig. 2.2. A simple *polymetric* visualization.

Example. In Fig. 2.2 we display three classes A, B, C as rectangles together with the metrics LOC and NOM. We map NOM on the width of each rectangle and we map LOC on the height of each rectangle. The metric information of these three classes which would usually be displayed with a table is thus visually displayed without needing to read through a table, but we can easily see that B has less lines of code, while C has a high LOC count (compared to A and B) and a low NOM count. We derive from this the information that C has rather long methods.

In the next chapter we discuss *Polymetric Views* in more detail.

2.3 Conclusions and Outlook

This book is not about metrics *per se*. It is about the way that metrics can be used to aid us in characterizing software systems, to evaluate them and when we detect design problems to provide the appropriate refactorings.

Characterizing the Design of Object-Oriented Systems

In the next chapter we provide the means to use metrics to characterize the design of systems. We do so by introducing the *Overview Pyramid* and two *Polymetric Views*. We then use these techniques to characterize a real system, namely *ArgoUML* . We chose *ArgoUML* because of its considerable size and complexity, the understandable domain (UML) and the availability of the source code[3].

[3] *ArgoUML* is open source, see `http://argouml.tigris.org` for more information.

Evaluating and Improving the Design of Object-Oriented Systems

In the subsequent chapters we provide the means to use metrics to evaluate the design of systems. We do so by introducing various metrics-based *detection strategies* to uncover *design disharmonies*. We also present the *Class Blueprint*, a visualization technique to display the internal details of classes and class hierarchies without the need to provide endless code listings.

An Integrated Approach Based on Metrics

The *Overview Pyramid*, the *Polymetric Views*, the *detection strategies* and the *Class Blueprint* presented in this book are complementary. While the first two are useful to get an impression of the system, the main purpose of the *detection strategies* is to produce lists of suspects that need to be investigated. Of course such investigations can be performed manually, i.e., by reading the source code in question. However, as the amount of source code that must be read can be several thousands of lines, we use the *class blueprint* to aid us in this task.

3

Characterizing the Design

Independently of our convictions about software metrics, every time we analyze or talk about a software system, we want to obtain an impression of the size and complexity of the software system to be able to *characterize* it. Some people like to express the size of a system in terms of lines of code, others use the number of classes, and even others only measure the amount of source code in megabytes.

These numbers are nothing more than the values of some basic metrics. For object-oriented systems the most common ones are the lines of code, the number of classes, the number of packages or subsystems, the number of operations (methods), etc.

Unfortunately, after getting such numbers in isolation, we still have trouble clearly characterizing the system. How come? There are several causes:

- *Unbalanced characterization.* The characterization is unbalanced because metrics only partially cover the aspects of interest. How faithful to object-oriented principles is the design of a system given the fact that it consists of 500 classes and 25,000 lines of code?
- *Misused metrics.* Knowing that a system has 500 classes does not tell us how large the system is because the classes in question could all be very large or very small. Neither do we know how complex a system is, if we know that it has 25,000 lines of code. Furthermore, how is a *line of code* defined? Do we count comments? Do we count lines containing curly brackets or semi-colons?
- *Uncorrelated metrics.* A lot of information slips through our fingers because it is not revealed by the raw numbers but by the proportions (ratios) between the raw values. Assume again that a system has 500 classes and 50,000 lines of code. Is something striking?

Not really. But what if the same system has 1 million lines of code? Then it becomes quite striking that the 500 classes are overloaded with functionality, since each class has an average of 2,000 lines! Or imagine that we additionally knew the number of methods in the system. What if we find out that there are only 1,000 methods in the system? It would be striking again because having only two methods per class looks suspiciously low.

- *Missing reference points.* Without reference values, any comparison of a qualitative assessment is impossible. A line of C++code is not the same as a line of Java, Smalltalk or Ada code because each language has its own syntax and semantics; and, even more important, each has its own style and best practices.

Providing an overall characterization of a system is a tough job. Consequently, it is deceiving to believe that several "classic" system-level metrics (usually size metrics such as the ones mentioned previously) can characterize a whole system. But characterizing a system is possible if we use the proper measurement means, i.e., if we extend our system-level measurements to other aspects than size and correlate these results in a proper manner.

In the remainder of the chapter we present two techniques to characterize object-oriented systems in terms of size and complexity which solve the above issues:

1. The *Overview Pyramid* is a metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and usage of inheritance.
2. The *Polymetric Views* are a visualization of software entities and their relationships enriched with metrics.

3.1 The Overview Pyramid

The overview of an object-oriented system must necessarily include metrics that reflect three main aspects:

1. *Size and complexity.* We want to understand how big and how complex a system is.
2. *Coupling.* The core of the object-oriented paradigm are objects that encapsulate data and that collaborate at run-time with each other to make the system perform its functionalities. We want to know to which extent classes (the creators of the objects) are coupled with each other.

3. *Inheritance.* A major asset of object-oriented languages is the ease of code reuse that is possible by creating classes that inherit functionality from their superclasses. We want to understand how much the concept of inheritance is used and how well it is used.

To understand these three aspects we introduce the *Overview Pyramid*, which is an integrated, metrics-based means to both describe and characterize the overall structure of an object-oriented system, by quantifying the aspects of complexity, coupling and usage of inheritance.

The basic idea behind the *Overview Pyramid* is to *put together in one place* the most significant measurements about an object-oriented system, so that an engineer can *see* and *interpret* in one shot everything that is needed to get a first impression about the system. The *Overview Pyramid* is a graphical representation of metrics values which can be visually interpreted. Yet, it is *not* a metrics visualization (in the sense of Sect. 2.2). From this point of view the *Overview Pyramid* is a graphical template for presenting (and interpreting) system-level measurements in a unitary manner. In order to visualize the overall structure of a software system, we will use the *Polymetric Views* (see Sect. 3.2).

The Principles of the Overview Pyramid

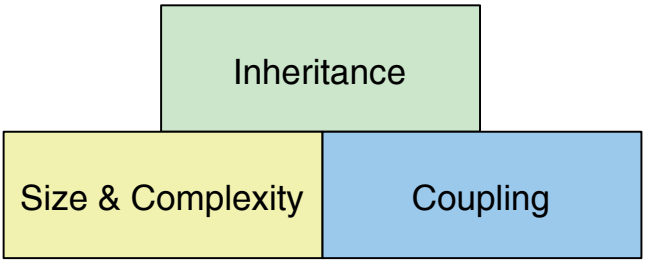


Fig. 3.1. The the three major structural aspects of a system quantified by the *Overview Pyramid*.

The three previously mentioned aspects are closely related and mutually influence each other (Fig. 3.1). While *Size & Complexity* and *Coupling* characterize every software system, the *Inheritance* aspect

is specific for object-oriented software and combines both elements of coupling (e.g., due to inheritance-specific dependencies) and additional size and complexity elements (e.g., due to type-checked up- and down-casts). Measuring these three aspects at the system level provides us with a comprehensive characterization of an entire system. An *Overview Pyramid* is composed of three parts concerning each aspect.

The Left Part: System Size and Complexity

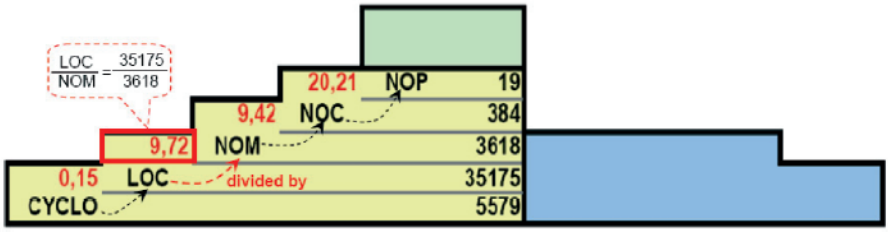


Fig. 3.2. Size and complexity characterization.

The left side of the *Overview Pyramid* (Fig. 3.2) provides information characterizing the *size* and *complexity* of the system.

Size and complexity: direct metrics. We need a set of *direct metrics* (i.e., metrics computed directly from the source code) to describe a system in simple, absolute terms. The metrics describing the size and complexity are probably some of the simplest and widely used metrics. They count the most significant modularity units of an object-oriented system, from the highest level (i.e., packages or namespaces), down to the lowest level units (i.e., code lines and independent functionality blocks). For each unit there is one metric in the *Overview Pyramid* that measures it. The metrics are placed one per line in a top-down manner, from a measure for the highest level unit (i.e., Number of Packages (NOP)) down to a complexity measure counting the number of independent paths in an operation (i.e., the cyclomatic complexity (CYCLO)). We use the following metrics for the *size and complexity* side of the *Overview Pyramid*:

- **NOP — Number of Packages**, i.e., the number of high-level packaging mechanisms, e.g.,*packages* in Java, *namespaces* in C++, etc.

- **NOC — Number of Classes**, i.e., the number of classes defined in the system, not counting library classes.
- **NOM — Number of Operations,**[1] i.e., the total number of user-defined operations within the system, including both methods and global functions (in programming languages that allow such constructs).
- **LOC — Lines of Code**, i.e., the lines of all user-defined operations. In the *Overview Pyramid* only the code lines containing functionality (i.e., lines of code belonging to methods) are counted.
- **CYCLO — Cyclomatic Number**, i.e., the total number of possible program paths summed from all the operations in the system. It is the sum of McCabe's cyclomatic number [McC76] for all operations. We use this metric to quantify the intrinsic functional complexity of the system.

Size and complexity: computed proportions. There is nothing new about the numbers above, but let us have a look at the numbers on the left: there are four computed numbers; we call them *computed proportions* because they are all computed in a "cascading" manner as ratios between the *direct metrics* placed on the right (see Fig. 3.2). All these four proportions have two essential characteristics:

- **Independence.** While the *direct metrics* discussed earlier influence each other (e.g., a system of 100 classes probably has fewer methods than one of 10,000 classes) these proportions are independent of one another. This makes each number a *distinct* characteristic of a specific aspect of code organization at both the procedural and the object-based level.
- **Comparability.** Being computed as ratios between absolute values, these proportions allow for easy comparison with other projects, independent of their size.

How are these proportions computed? As depicted in Fig. 3.2 each proportion metric is computed as a ratio between two consecutive numbers, by dividing the lower number by the next upper one. Thus, for example, the ratio emphasized in the figure (i.e., the one positioned second lowest in the *Overview Pyramid*) is computed as a ratio between the value of LOC (the number on the line below it) and NOM (the number on the same line). The number denotes the average number of code lines per operation in the analyzed system. To

[1] We use the well-known NOM acronym; in the *Overview Pyramidall* user-defined operations are counted, including methods or global functions.

characterize the *size and complexity of a system*, based on the direct metrics used, the following proportions result:

- **High-level Structuring (NOC /Package)**. This proportion provides the reader with a first impression of the packaging level, i.e., the high-level structuring policy of the system. In other words, it indicates if packages tend to be *coarse grained* or *fine grained*.
- **Class structuring (NOM/Class)**. This proportion provides a hint about the *quality of class design*, because it reveals how operations are distributed among classes. Very high values might be a sign of missing classes, i.e., an exaggerated stuffing of operations into classes. In the case of global functions which cannot be attached to any class, we consider them as static methods of a default anonymous class.
- **Operation structuring (LOC/Operation)**. This is an indication of how well the code is distributed among operations. Very high numbers suggest the system's operations are rather "heavy". This can be used as a first sign of the how the system is structured from the point of view of procedural programming.
- **Intrinsic operation complexity (CYCLO/Code Line)**. This last ratio characterizes how much *conditional complexity* we are to expect in operations (e.g., 0.2 means that a new branch is added every five lines).

The Right Part: System Coupling

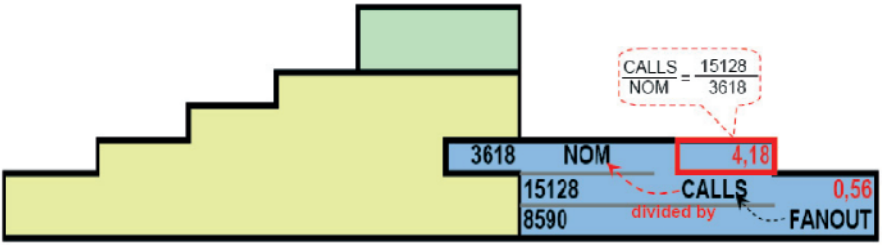


Fig. 3.3. Characterizing a system's coupling.

The second part of the *Overview Pyramid* provides an overview with information about the level of coupling in the system (see Fig. 3.3), by means of operation invocations.

System coupling: direct metrics. The key questions when trying to characterize the level of coupling in a software system are: How *intensive* and how *dispersed* is coupling in the system? The two direct metrics that we use are:

- **CALLS — Number of Operation Calls**, i.e., this metric counts the total number of distinct operation calls (invocations) in the project, by summing the number of operations called by all the user-defined operations. If an operation `foo()` is called three times by a method `f1()` it will be counted only once. If it is called by methods `f1()`, `f2()` and `f3()`, three calls will be counted for this metric.
- **FANOUT — Number of Called Classes**, this is computed as a sum of the FANOUT [LK94] metric (i.e., classes from which operations call methods) for all user-defined operations. This metric provides raw information about how dispersed operation calls are in classes.

System coupling: computed proportions. Again, the numbers above *describe* the total coupling amount of a system, but it is difficult to use those numbers to characterize a system with respect to coupling. We can compute, using the number of operations (NOM), two proportions that better characterize the coupling of a system.

- **Coupling intensity (CALLS/Operation)**. This proportion denotes the level of collaboration (coupling) between the operations, i.e., how many other operations are called on average from each operation. Very high values suggest that there is excessive coupling among operations, i.e., a sign that the calling operation does not "talk" with the right "counterpart".
- **Coupling dispersion (FANOUT /Operation Call)**. This proportion is an indicator of how much the coupling involves many classes (e.g., 0.5 means that every two operation calls involve another class).

Top Part: System Inheritance

The top part of the *Overview Pyramid* is not a ladder as in the previous cases; it is composed of two metrics that provide an overall characterization of *inheritance usage*. These proportion metrics reveal how much inheritance is used in the system, as a first sign of how much "object-orientedness" (i.e., usage of class hierarchies and polymorphism) to expect in the system.

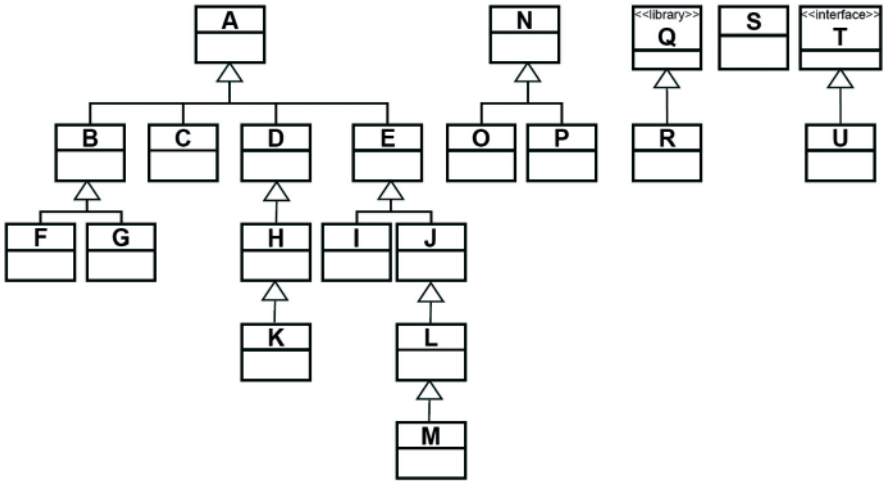


Fig. 3.4. Example to illustrate the computation of inheritance metrics used in the *Overview Pyramid*

The two metrics to characterize the presence and the shape of class hierarchies are:

1. **ANDC — Average Number of Derived Classes**, i.e., the average number of direct subclasses of a class. All classes defined within the measured system (and only those) are considered. Interfaces (in Java or C#) are not counted. If a class has no derived classes, then the class participates with a value of 0 to ANDC . The metric is a first sign of how extensively abstractions are refined by means of inheritance. We illustrate how ANDC is computed based on the example presented in Fig. 3.4. First, we count the classes: there are 19 classes, as we do not count *Q* because it is a library class; we also do not count *T* because it is an interface. Out of the 19 classes, 11 have no subclasses at all, four classes (i.e., *D*, *H*, *J*, *L*) each have one direct subclass, three classes (i.e., *B*, *E*, *N*) have 2 direct subclasses each, and there is one class (i.e., *A*) with four direct descendants. Thus, ANDC is computed as:

$$ANDC = \frac{11 \cdot 0 + 4 \cdot 1 + 3 \cdot 2 + 1 \cdot 4}{19} = 0.73$$

2. **AHH — Average Hierarchy Height.** The metric is computed as an average of the *Height of the Inheritance Tree* (HIT) among the *root classes* defined within the system. AHH is the average of the maximum path length from a root to its deepest subclasses. A

class is a *root* if it is not derived from another class belonging to the analyzed system. Interfaces (in Java or C#) are not counted. Standalone classes (i.e., classes with no base class in the system and no descendants) are considered root classes with a HIT value of 0. The number tells us how deep the class hierarchies are. Low numbers suggests a *flat class hierarchy* structure. In order to illustrate how AHH is computed we revisit the sample system depicted in Fig. 3.4. First, we have to count the *root classes*. Based on the specification of AHH , we identify five root classes: *A*, *N*, *R* (because it is derived from a library class), *S* and *U* (because the implementation of an interface does not make *U* a subclass). For these root classes the values for the HIT metric are: HIT(*A*) = 4, HIT(*N*) = 1, HIT(*R*) = 0, HIT(*S*) = 0, HIT(*U*) = 0. Thus, AHH is computed as:

$$AHH = \frac{4+1+0+0+0}{5} = 1$$

Why did we choose these two proportions and why are they sufficient? They capture two complementary aspects of a class hierarchy: while ANDC provides us with an overview of the *width* of inheritance trees, the ahh metric reveals if class hierarchies tend to be *deep* or shallow. The two metrics provide us with first hints on whether we should expect intensive usage of inheritance relations (ANDC) and, if so, they help us understand how deep these hierarchies are (AHH). Summarizing, Fig. 3.5 shows the entire *Overview Pyramid*.

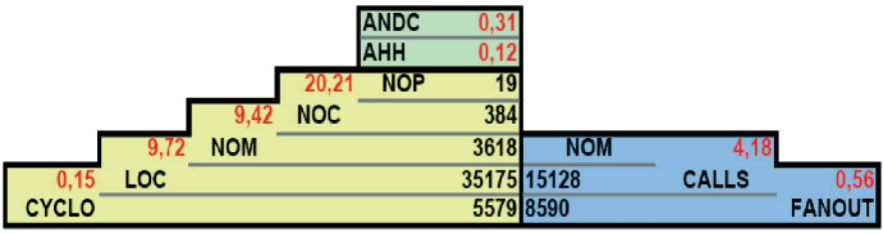


Fig. 3.5. A complete *Overview Pyramid*.

Interpreting the Overview Pyramid

We have seen that the *Overview Pyramid* characterizes a system from three different viewpoints: *size and structural complexity*; *coupling* and the usage of the *inheritance relation*. The characterization

is based on the eight *computed proportions* displayed in the *Overview Pyramid*. All these values have one important property: they are *independent* of the size of the system, allowing for an objective assessment.

Did we say objective? We need a reference point, other than common sense (which is not enough to interpret the numbers). For example, is the 9.42 *NOM/Class* value in Fig. 3.5 normal, too small or too large? We need a reference point.

Based on the statistical thresholds described in the previous chapter (see Sect. 2.1) and using the same statistical base, we computed the *low*, *average* and *high* thresholds for all the proportions.[2] All thresholds are summarized in Table 3.1 [3].

	Java			C++		
Metric	Low	Average	High	Low	Average	High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28	0.37
AHH	0.09	0.21	0.32	0.05	0.13	0.21

Table 3.1. Statistical thresholds of 45 Java and 37 C++systems computed for the proportions (ratios) used in this *Overview Pyramid*.

Based on these thresholds, we refer to the *Overview Pyramid* for the sample system depicted in Fig. 3.5 and knowing that it represents a Java system we interpret the pyramid.

The *Size and Complexity* side can be interpreted as follows: the operations in the system have a rather low intrinsic complexity (as 0.15 is closer to the LOW threshold, which is 0.16), while the size of operations is close to the average value for Java systems. With 9.42

[2] We did not include a *very-high* threshold because we consider that the three *low*, *average* and *high* thresholds are enough for the interpretation of the *Overview Pyramid*.

[3] As already mentioned in the previous chapter, these metrics are collected from a statistical base of 45 Java projects and 37 C++projects. The projects have various sizes (from 20,000 up to 2,000,000 lines), they come from various application domains, and we included both open-source and industrial (commercial software) systems.

operations per class, and 20.21 classes per package the system has rather large classes and packages.

On the *System Coupling* side we learn the following: the system is intensively coupled in terms of operation calls, but these calls tend to be rather localized, i.e., functions tend to call many operations from few classes.

In the *Class Hierarchies* part we read the following: The class hierarchies are frequent in the system (low ANDC value), and very shallow (low AHH value).

To facilitate the visual interpretation of the *Overview Pyramid* we associate the computed proportions with colors that map those numbers to their semantics in terms of the three types of statistical thresholds (i.e., *low*, *average*, *high*) presented in Table 3.1. Thus, we place a computed proportion in a **blue** rectangle to show that the value is closest to the *low* threshold. Similarly, if a value is closest to the *average* threshold it will be placed in a **green** rectangle; eventually, if the computed value is closest to the *high* threshold, the number will be placed in a **red** rectangle.

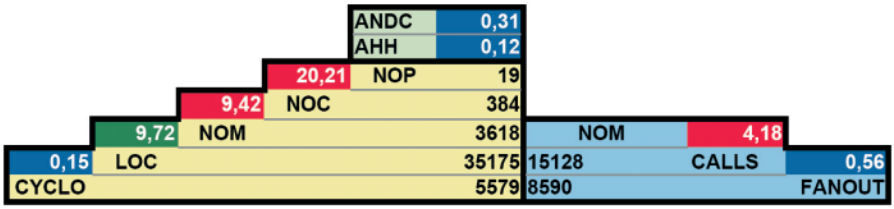


Fig. 3.6. Using colors to interpret the *Overview Pyramid*. BLUE means a *low* value; GREEN means an *average* value; RED stands for a *high* value.

3.2 Polymetric Views

In the context of this book we use so-called *Polymetric Views* to visualize software. A *polymetric view* is a metrics-enriched visualization of software entities and their relationships[LD03]. Their main benefit is that they can visually render numbers in a simple, yet effective and highly condensed way which is directly interpretable by the viewer. It is a lightweight technique that can be easily implemented into any development environments and has already been adopted by some research prototypes [Fav01] because of its simplicity.

The Principles of a Polymetric View

We use rectangles to display software entities or abstractions of them, and we use edges to represent relationships between the entities.[4] This is a widely used practice in information visualization and software visualization tools. Ware claims that "other possible graphical notations for showing connectivity would be far less effective" [War00].

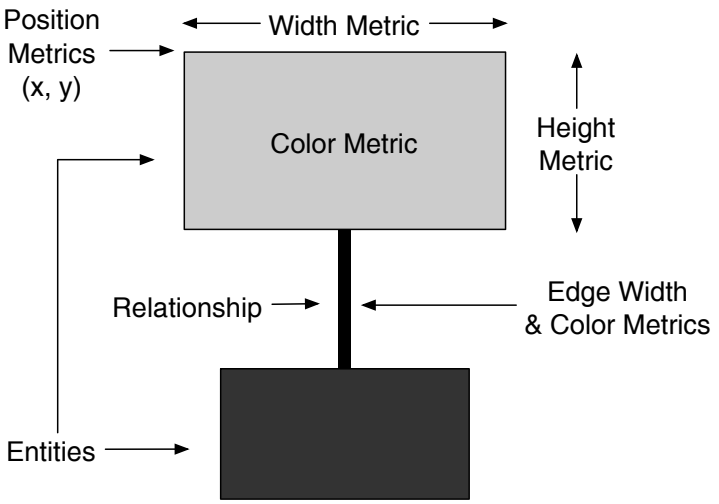


Fig. 3.7. The principles of a *polymetric view*.

We enrich this basic visualization technique by rendering up to five metric measurements on a single node and two metrics on a single edge simultaneously, as we see in Fig. 3.7. We exploit the following visual attributes:

- *Node size.* The width and height of a node can render two measurements. We follow the convention that the wider and the higher the node, the bigger the measurements its size is reflecting.
- *Node color.* The color interval between white and black can display a measurement. Here the convention is that the higher the

[4] The underlying model is thus that a software system can be modelled as a graph where the vertices represent entities, i.e., source code artifacts or abstractions of them, and the arcs represent relationships.

measurement, the darker the node. Thus light gray represents a smaller metric measurement than dark gray. We opted against using different colors, because nominal colors cannot reflect quantities. Tufte [Tuf01] states that "Despite our experiences with the spectrum in science textbooks and rainbows, the mind's eye does not readily give a visual ordering to colors. Because they do have a natural visual hierarchy, varying shades of gray show varying quantities better than color".[5]

- *Node position.* The X and Y coordinates of the position of a node can reflect two other measurements. This requires the presence of an absolute origin within a fixed coordinate system, therefore not all views can exploit such metrics (e.g., in the case of a tree view the position is intrinsically given by the tree layout and cannot reflect a measurement).
- *Edge width and color.* The width and the color interval between white and black of an edge can be used to render two supplemental measurements. If we display an edge between two classes which represents the method invocations that go from one class to the other, the edge width and color can be used to represent the *weight* of the relationships in terms of number of invocations.

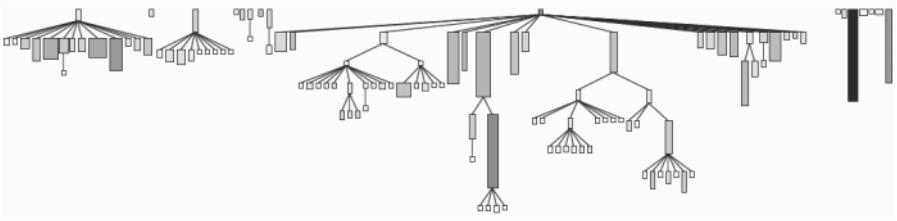


Fig. 3.8. A *System Complexity* view. This view uses the following metrics: width metric = number of attributes, height metric = number of methods, color metric = number of lines of code.

Example. In Fig. 3.8 we see a *polymetric view* called *System Complexity* of the software system *CodeCrawler* . The metrics used in this view are the number of attributes for the width, the number of methods for the height, and the number of lines of code for the color of

[5] There are exceptions to this rule: for example, weather maps use a spectrum which ranges from blue to red to denote cold and warm temperatures.

the displayed class nodes. The edges represent inheritance relationships. Using this view it is easy to spot the large classes (in terms of behavior, i.e., methods, and state, or variables).

polymetric view alone are not enough to tackle the problems of reverse engineering, but they support and complement other techniques to enhance and facilitate the comprehension of software systems. In the context of this book we use the *polymetric views* as a tool to understand a system at a coarse-grained level. Moreover, sometimes — and this is one of the strong points of our views — it is enough to visualize a certain part of a system to get a correct impression. Yet, for the cases where we want to double-check that impression in the source code, the implementation of *polymetric views* (see Appendix C) should allow an easy navigation to the implementation files.

The strongest point of *polymetric views* is that they can combine multiple metrics and produce different colored shapes that can be interpreted by the viewer: The viewer then has to look out for certain visual symptoms (depending on the view) and can thus visually detect interesting and/or disharmonious parts in the system. We also stress that the interesting things are not necessarily visually obvious, but require a trained eye in order to relate and compare things. Next, we are going to present how *polymetric views* support the correlation and comparison of various software entities.

Relating software entities. The eye of the viewer is at first attracted by the larger classes in the system, e.g., the two on the right side. It is true that these classes are the largest in the system, at least as far as the number of methods is concerned. Moreover, the bigger one is also very dark, denoting a high number of lines of code. The advantage here is that we do not have to deal with actual numbers but can put things in relation to each other. These two classes are the largest in *this* system, while in another system they could actually be categorized as being very small or very large. This depends on the system, its domain and the language it is implemented in.

Comparing software entities. The viewer can compare different design fragments: the larger of the two inheritance hierarchies contains two sub-hierarchies. Looking closer at the smaller one of the two large sub-hierarchies we notice that all classes, especially the leaf classes, are more or less of the same small size with *one* exception. This exception is a starting point for an inspection since we expect that sibling classes in inheritance hierarchies will have a similar complexity. The point is that the *polymetric views* allow us to relate things

to each other and compare them not only within a global, but especially within a local context. As another example: in our experience if sibling classes look very similar, i.e., they have a similar number of attributes and methods, this may oftentimes point to duplication.

The research we have performed on the *polymetric views* is extensive and can be found in various publications [Lan03a, Lan03b, LD03, LD05, DL05, LDGP05, GLD05], but since the goal of this book is not to explain everything about the views in detail, we direct the interested reader to those publications. Note also that the views can be easily composed and tweaked, especially regarding the metrics, i.e., the number of potential views is very high, but in the context of this book we limit ourselves to using two simple views, which are presented next.

Polymetrics Views Exemplified

We introduce two *polymetric views* namely the *System Hotspots* and *System Complexity* views. We exemplify their use with a small system of a couple of hundred classes called Duploc, used for duplication detection. Note that Duploc was developed in VisualWorks Smalltalk; we use it to illustrate the language independence of the *polymetric views*.

***System Hotspots* View**							
Nodes	Edges	Layout	Width	Height	Color	X	Y
Classes	–	Checker	NOM	NOM	free	–	–

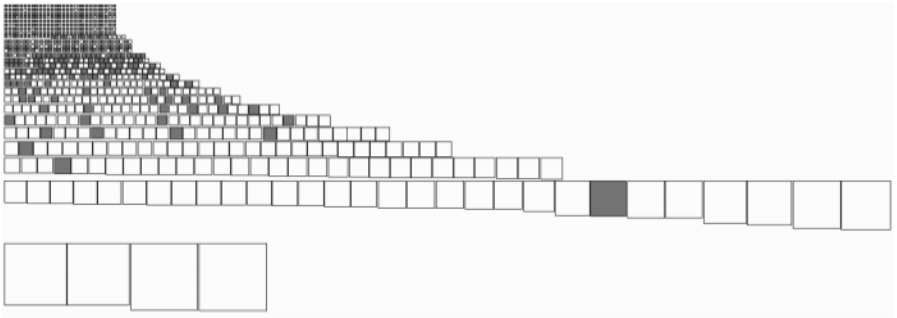


Fig. 3.9. A *System Hotspots* view of Duploc. The nodes represent all the classes, while the size of the nodes represent the number of methods they define. The gray nodes represent metaclasses.

This simple view helps to identify large and small classes and scales up to very large systems. It relates the number of methods to the number of attributes of a class. The nodes are sorted according to the number of methods, which makes the identification of behavioral outliers easy (note that a class that has many methods also tends to consist of many lines of code).

This view gives a general impression of the system in terms of overall size (how many classes are there?) and in terms of size of the classes (are there any really large classes and how many of these giants are there?).

Large nodes represent voluminous classes that define many methods and should be further investigated. Small nodes represent either structs or very small classes. Classes with NOM = 0 should be investigated to see if they are not dead code. Further evidence can be gained from the color, which can be used to reflect the number of lines of code of a class. Should a tall class have a light color it means that the class contains mostly short methods.

Example - *System Hotspots*. In Fig. 3.9 we see a *System Hotspots* view of all the classes of Duploc. The classes in the bottom row contain more than 100 methods and should be further investigated. They are *DuplocPresentationModelController (107 methods)*, *RawMatrix (107)*, *DuplocSmalltalkRepository (116)* and *DuplocApplication (117 methods)*. This view shows that Duploc is a system of more than 300 classes, where the largest classes contain more than 100 methods. It also shows an impressive number of very small classes implementing few methods.

***System Complexity* View**							
Nodes	Edges	Layout	Width	Height	Color	X	Y
Classes	Inheritance	Tree	NOA	NOM	LOC	–	–

This view is based on the inheritance hierarchies of a subject system and gives clues on its complexity and structure. For very large systems it is advisable to apply this view first on subsystems, as it uses a lot of screen space. The goal of this view is to classify inheritance hierarchies in terms of the functionality they represent in a subject system. Hints about the functionality they provide can be inferred by knowing the names of the classes (usually just knowing the name of the root class is telling enough).

The view helps to identify and locate the important inheritance hierarchies, but also shows whether there are large classes not part of

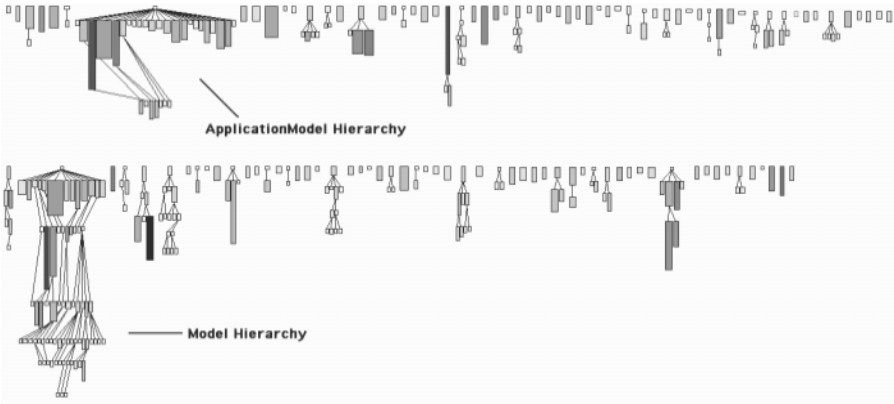


Fig. 3.10. A *System Complexity* view on Duploc. The nodes represent the classes, while the edges represent inheritance relationships. As metrics we use the number of attributes (NOA) for the width, the number of methods (NOM) for the height and the number of lines of code (LOC) for the color.

a hierarchy. It also answers the question about the size of the subject system. Moreover, it helps to detect exceptional classes in terms of number of methods (tall nodes) or number of attributes (wide nodes).

Tall, narrow nodes represent classes with few attributes and many methods. Deep or large hierarchies are definitively subsets of the system on which more specific views should be applied to refine their understanding. Large, standalone nodes represent classes with many attributes and methods without subclasses. It may be worth looking at the internal structure of the class to learn if the class is well structured or if it could be decomposed or reorganized.

Example - *System Complexity.* In Fig. 3.10 we present a *System Complexity* view of all the classes of Duploc. We see that Duploc is composed of many classes not organized in inheritance hierarchies. Indeed, there are some very large classes which do not have subclasses. The largest inheritance hierarchies are five and six levels deep. Noteworthy hierarchies seem to be the ones with the following root classes: *AbstractPresentationModelControllerState*, *AbstractPresentationModelViewState* and *DuplocSourceLocation*. By manually inspecting the first one, with the root class *AbstractPresentationModelControllerState* having 31 descendants, we infer that it seems to be the application of the *state* design pattern [GHJV95, ABW98] for the controller part of an Model–View–Controller pattern. Such a complex hierarchy within Duploc is necessary, since Duploc does not make

any use of advanced graphical frameworks, but uses the standard GUI framework that comes with the development environment it was written with. Following this track of investigation we look for other signs of the MVC pattern and find a hierarchy with *AbstractPresentationModelViewState* as root class with 12 descendants, which seems to constitute the view part of the MVC pattern. This view shows that Duploc consists of several very small hierarchies composed of small classes and two bigger hierarchies, where one represents the domain model of Duploc (the Model hierarchy), and the other one contains all GUI-related classes (the ApplicationModel hierarchy).

3.3 Metrics at Work

In this section we put at work the metrics-based techniques presented so far in this chapter. Our aim is show you how these techniques can be used for describing and characterizing at the system level an object-oriented software system. For this purpose, we will apply them on the *ArgoUML* system[6]. It is an open-source UML modeling tool written in Java and consists of more than 220,000 raw lines (code and comments). Before going into more detail, let us first take a look at some of its size characteristics (see Table 3.2).

Metric	Value	Remarks
No. of Lines of Code	223,068	including comments
No. of Source Files	1,209	*.java files
No. of Packages	99	–
No. of Classes	1,393	including 140 inner classes
No. of Methods	9,561	including accessor methods
No. of Attributes	3,358	all variables including static and local variables

Table 3.2. Size properties of the *ArgoUML* system.

Characterizing *ArgoUML* Using the *Overview Pyramid*

To get a more detailed overview of the system we use the *Overview Pyramid* (24) which for *ArgoUML* is depicted in Fig. 3.11.
To interpret the numbers we compare the values of *ArgoUML* with the statistically computed values summarized in Table 2.2:

[6] We analyzed the end of October 2004 version. See http://argouml.tigris.org/ for more information and the source code itself.

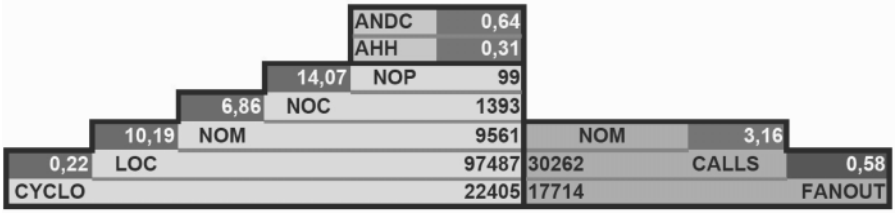


Fig. 3.11. The *Overview Pyramid* applied to *ArgoUML* .

- *Size and complexity*. The system has a complexity which is above average and close to high, while the size of operations, classes and packages are rather balanced, staying close to the average values of other systems.
- *Coupling*. The system is intensively coupled by method calls, but these calls tend to be rather localized, i.e., methods tend to call many other methods from few classes.
- *Inheritance*. *ArgoUML* extensively uses inheritance, as classes tend to have a large number of descendants (compare ANDC values with statistical results), and class hierarchies are also rather "deep" (compare AHH values with statistical results). This interpretation based only on numbers is confirmed by the *System Complexity* view depicted in Fig. 3.13.

Characterizing *ArgoUML* Using *Polymetric Views*

To visually get a first idea of the raw size of the system we display in Fig. 3.12 a *System Hotspots* view. What we see in the figure are all 1,393 model classes of *ArgoUML* . The size of the nodes represents the number of methods (NOM), while the color represents the lines of code (LOC) of each class. The class ModelFacade is striking because of its size (453 methods, 3,507 lines) compared to the other classes in the system. The next three largest classes are CoreFactory (116 NOM, 1,100 LOC), GeneratorCpp (97 NOM, 2,259 LOC) and Project (85 NOM, 690 LOC). Another class which has many lines compared to its number of methods is ParserDisplay ("only" 53 methods but 2517 lines). Moreover, we thickened the border of the abstract classes in the system, and perceive that in *ArgoUML* there are many of them, the two largest being FigNodeModelElement and FigEdgeModelElement which each have 81 and 55 methods.

In Fig. 3.13 we see a *System Complexity* visualization of the complete system in terms of the inheritance hierarchies. The reader

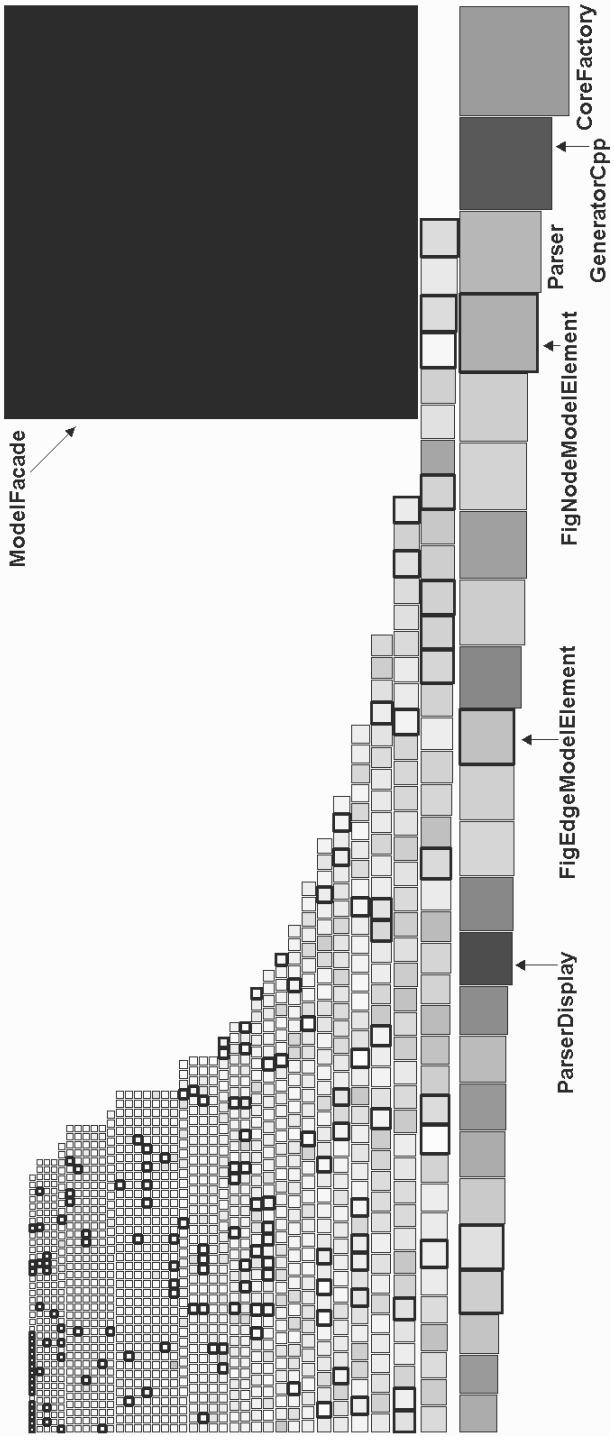


Fig. 3.12. A *System Hotspots* view of *ArgoUML* .

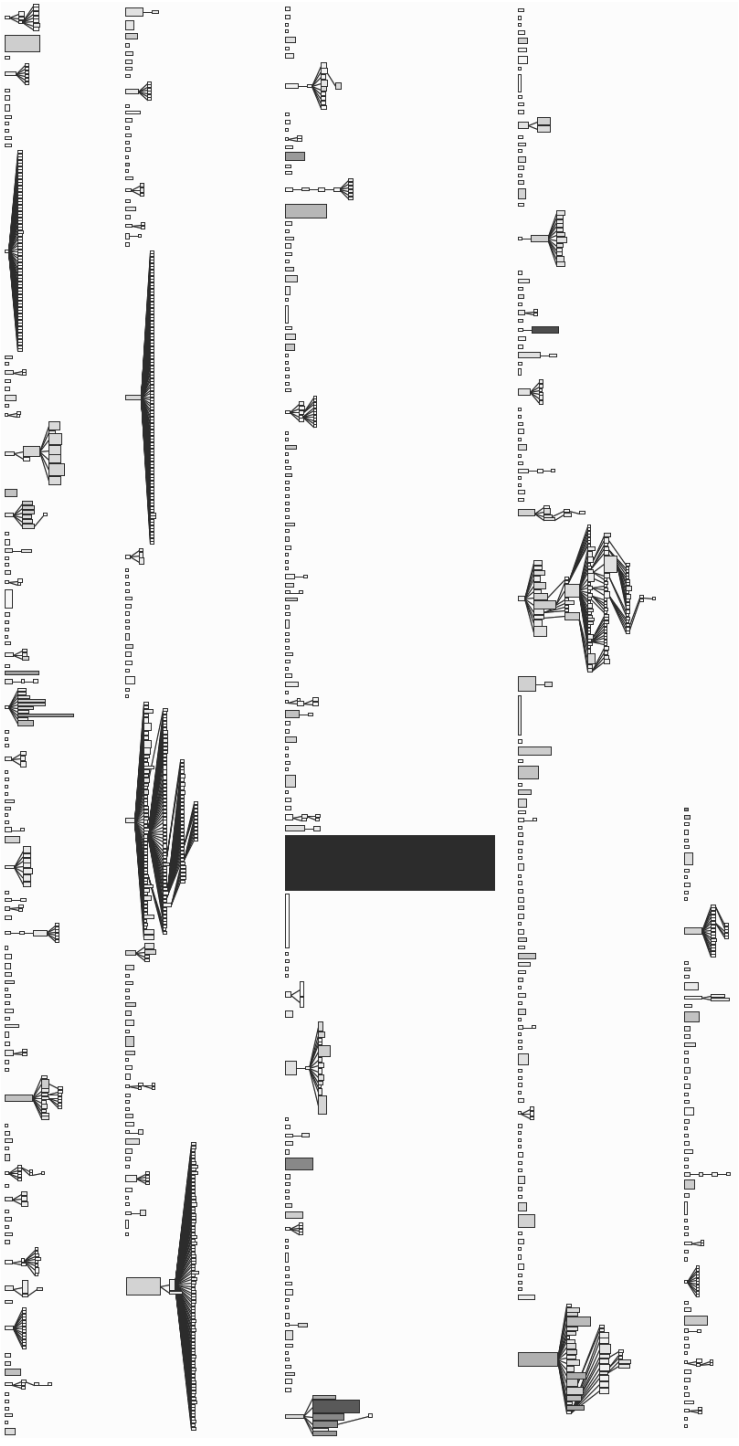


Fig. 3.13. A *System Complexity* view of the complete *ArgoUML* system.

should keep in mind that what is actually displayed in one single figure here resides in more than 1,200 source files distributed in dozens of directories: We are talking about a fairly complex system, although certainly not a *very* complex system. However, we see that *ArgoUML* has been implemented using some complex and deep inheritance hierarchies (some of them have more than seven levels).

3.4 Conclusions and Outlook

In this chapter we presented two approaches which allow us to characterize and assess the design of object-oriented software systems, the *Overview Pyramid* and *Polymetric Views*.

Overview Pyramid— It provides us with a numerical overview of simple and derived system metrics, and characterizes a system in terms of size & complexity, coupling, and inheritance.

Polymetric Views— They provide a simple yet powerful means to *see* a software system in the same terms.

They both serve as starting point to understand the intricacies of systems without the need of reading source code or going through large tables full of numbers. They serve another purpose too: They are the basis for the two approaches that we present in the next chapter, the *Detection Strategy* and the *Class Blueprint*.

4

Evaluating the Design

Object-oriented construction and design are *misleading* words, because they make people think that software can be constructed like a house or designed like a piece of furniture. This is a myth which is hard to kill. The truth is that a software system is at least as complex as any other engineering artifact (such as buildings, if not more, considering the fact that it evolves much faster).

Moreover, a modern software system is written by many people at the same time, leading to (1) communication issues, (2) compatibility issues and above all (3) complexity issues. In addition, a system cannot be written once and for all, put in place and then work forever. It is actually grown like a plant with many interrelated parts that depend on each other, that die, that change, that are bugged and must be fixed (introducing new bugs), etc.

You may well imagine that a plant which is not correctly watered will die. In much the same spirit we can say that a system which is not being cured and maintained will slowly decay and eventually die. But all metaphors, including the one of the plant, do not fit the context of object-oriented software. These systems are much more complex and consist of thousands of artifacts and relationships between the artifacts. A change in one part of the system may break other parts of the system. This is not due to bad programming practice, it is just a matter of complexity: you cannot expect to have a complete picture of a large software system. Moreover, we are speaking about evolving systems which change continuously, leading to more complexity [LB85, DDN02].

Still in software construction finding an appropriate design is important. Indeed it may help people understand the system and ease future changes. For example, it is well-known that using explicit type-

checks goes against the essence of object-oriented programming and creates brittle and hard to change code [DDN02]. However it is important to understand that design decisions such as the impact of applying a given design pattern [GHJV95, ABW98] is difficult to assess — using a design pattern introduces an intrinsic complexity which should be balanced by the benefits of the pattern application. Identifying the exact responsibilities of objects and how they should be distributed among objects is complex. In this book we show you how to use metrics to assess the quality of a design. Metrics measure structural elements and as such they can reveal hidden symptoms. But there will always be a gap between the symptoms and the deep assessment that an expert in object-oriented design can do using these symptoms. Therefore it is important to consider metrics as a tool and as with any tool to know their advantages and disadvantages. This leads us to the crucial questions we answer in this book:

What entities do we measure in object-oriented design?

It depends . . . on the language. In most object-oriented languages we find and can measure *classes*; *operations* (including methods and functions); *variables* (including the whole range from attributes to local variables) etc.

What metrics do we use?

It depends . . . on our measurement goals. We may want to assess the size, the complexity, the quality, etc.

What can we do with the information obtained?

It depends . . . on our objectives. We may want to just assess the status quo to calm down management, we may want to brag with colleagues (“my system is bigger and better than yours”), or we may actually want to ameliorate the quality of parts of the system.

Design Harmony

Simple metrics are not enough to understand and evaluate design, or to put it bluntly: you cannot understand the beauty of a painting by measuring its frame or understand the depth of a poem by counting the lines. Object-oriented systems can be seen as pieces of complex art and the creativity that programming involves backs up this bold statement. Metrics can help to evaluate and improve designs, but

those have to be *meaningful* metrics that are put in a context of *design harmony*.

The reader might be confused to find a word as ambiguous as *harmony* in a book about object-oriented metrics. After all, a major point of this book is that software, objet-oriented and not, can and should be measured. However, metrics have to be put in a context. The aspect of measurability and more specifically about thresholds (such as: When should a class or a method be considered *too large*?) does not make sense if there is no context: A class implementing a parser is never going to be small, the domain is just too complex to be modelled in a concise way. Still, and this is where harmony comes into play, a class can be implemented in several ways, theoretically even in only one huge method. This would however make the class hard to understand.

An application, a class, a method and any other artifact in a software system should be implemented in an harmonious way, e.g., a class has to implement an *appropriate* number of methods of *appropriate* size, complexity, and functionality.

Appropriate to what? This appropriateness is a kind of *harmony* that can indeed be measured and reached. This overall harmony is composed of three distinct harmonies that concern every software artifact:

1. **Identity Harmony** – *"How do I define myself?"* Every entity in a software system must justify its existence: does it implement a specific concept and how does it do that? Is it doing too many things or nothing at all?

2. **Collaboration Harmony** – *"How do I interact with others?"* Every entity collaborates with others to fulfill its tasks. Does it do that all on its own, or does it use other entities. How does it use them? Does it use too many?

3. **Classification Harmony** – *"How do I define myself with respect to my ancestors and descendants?"*. This harmony combines elements of both identity and collaboration harmony in the context of inheritance. For example, does a subclass use all the inherited services, or does it ignore some of them?

Boiling it down: Every artifact in a system needs to be in harmony with itself (not too large, not too small, not too complex, not too simple, etc.), in harmony with its collaborators (do not talk to everybody,

do not talk to nobody, etc.), and finally in harmony with its ancestors and descendants. Every artifact must have its *appropriate* place, size, and complexity to fit the system context.

Detection Strategies and Class Blueprints

In the remainder of this chapter we present two techniques to evaluate the design of object-oriented systems and to detect structural disharmonies:

1. A *detection strategy* is a composed logical condition, based on metrics, that identifies those design fragments that are fulfilling the condition.
2. A *class blueprint* is a semantically rich visualization of the internal structure of classes and class hierarchies. We use a *class blueprint* to inspect source code and to detect visual anomalies which in turn point to design disharmonies.

4.1 Detection Strategies

The Principles of Detection Strategies

A metric alone cannot help to answer all the questions about a system and therefore metrics must be used in combination to provide relevant information. Why?

Using a medical metaphor we might say that the interpretation of abnormal measurements can offer an understanding of *symptoms*, but the measurements cannot provide an understanding of the *disease* that caused those symptoms. The *bottom-up approach*, i.e., going 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 metrics is too fine grained to indicate 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.

How should we combine then metrics in order to make them serve our purposes? The main goal of the mechanism presented below is to provide engineers with a means to work with metrics at a *more abstract level*. The mechanism defined for this purpose is called a *detection strategy*, defined as follows:

A Detection Strategy *is a composed logical condition, based on metrics, by which design fragments with specific properties are detected in the source code.*

The aim with *detection strategies* is to make design rules (and their violations) quantifiable, and thus to be able to detect *design problems* in an object-oriented software system, i.e., to find those design fragments that are affected by a particular design problem.

The use of metrics in the *detection strategies* is based on the mechanisms of *filtering* and *composition*, described next.

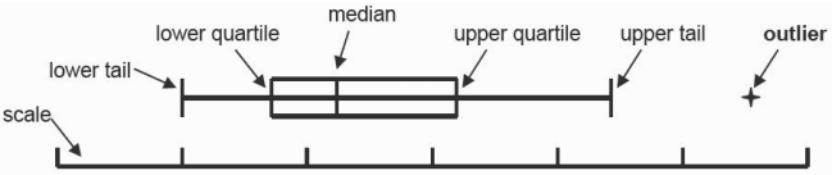
Filtering

The key issue in filtering is to reduce the initial data set so that only those values that present a special characteristic are retained. A *data filter* is a boolean condition by which a *subset* of data is retained from an initial set of measurement results, based on the particular focus of the measurement.

The purpose of filtering is to keep only those design fragments that have special properties captured by the metric. To define a data filter we must define the values for the bottom and upper limits of the filtered subset. Depending on how we specify the limit(s) of the resulting data set, filters can be either *statistical*, based on *absolute thresholds*, or based on *relative thresholds*.

Statistical Filters

A first approach when we seek abnormal values in a data set is to employ statistical means for detecting those values. Thus, the (binary) filtering condition and its semantics are implicitly contained in the statistical rules that we use. The advantage of this approach is that it is not necessary to specify explicitly a threshold value beyond which entities are considered abnormal. One significant example of a statistical filter is the *box-plot technique*, which is a statistical means for detecting the abnormal values (outliers) in a data set [FP96]. In this case, the detection of outliers starts from the *median* value, which can be directly computed from the analyzed data set. Based on this median value, two pairs of thresholds are computed i.e., the *lower/upper quartile* and resp. *lower/upper tail*. These thresholds are again computed *implicitly*, based on the formulas presented in Fig. 4.1. Eventually, in a box-plot an *outlier* is a value from the data set that is either higher than the *upper tail* or lower than the *lower tail* thresholds.



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**

Fig. 4.1. The box-plot technique [FP96].

Threshold-Based Filters

The alternative way of defining filters is to pick-up a *comparator* (e.g.,*lower than* or *highest values*) and specify explicitly a threshold value (e.g., *lower than 10* or *5 highest values*). But, as already discussed in Chapter 2 (see Sect. 2.1), the selection of proper thresholds is one of the hardest issues in using metrics. There are two ways in which these filters can be specified:

1. *Absolute Comparators.* We use the classical comparators for numbers, i.e., $>$ (*greater than*); $\geq$ (*greater than or equal to*); $<$ (*less than*); $\leq$ (*less than or equal to*).
2. *Relative Comparators.* The operators that can be used are *highest values* and *lowest values*. These filters delimit 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., retrieve the 20 entities with the highest LOC values) or *percentile* (e.g., retrieve the 10% of all entities having the lowest LOC values). This kind of filter is useful in contexts where we consider the highest or lowest values from a given data set, rather than indicating precise thresholds.

Composition

In contrast to simple metrics and their interpretation models, a *detection strategy* is intended to quantify more complex design rules,

that involve multiple aspects that needed quantification. 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. It is based on a set of AND and OR operators that compose different metrics together to form a composite rule.

Graphical Notation for Detection Strategies

A *detection strategy* is a composed logical expression by which design entities addressed by the strategy are filtered. Instead of using formulas, we decided to take advantage of a well-known graphical notation used to represent logical circuits. In this representation, the *composition operators* are represented as logical AND and OR gates (see Fig. 4.2). Both the input and the output terms of the gates are *filters*. Inputs can be either *simple* or *composed* filtering conditions.

Representation of Simple Filters

A *simple filter* is represented as a gray rounded rectangle, composed of an *informal description* of the filtering condition and a white compartment (box) where the *filtering formula* is depicted i.e., the metric followed by the filtering operator and the threshold value (see Fig. 4.2).

Representation of Composed Filters

A *composed filter* is represented as a gray rounded rectangle that contains only the *informal description* of the composed condition that it stands for (see Fig. 4.3). Note, that a composed filter is always the result (output) of another gate. Notice, that these intermediary terms are not conceptually necessary. We introduced them, in order to increase increase the understandability of more complex *detection strategies*.

Detection Strategies Exemplified

A *detection strategy* can be used to express in a quantitative manner deviations from a given set of *rules of design harmony*. While it is impossible to establish an objective and general set of such harmony rules that would lead automatically to high-quality design if

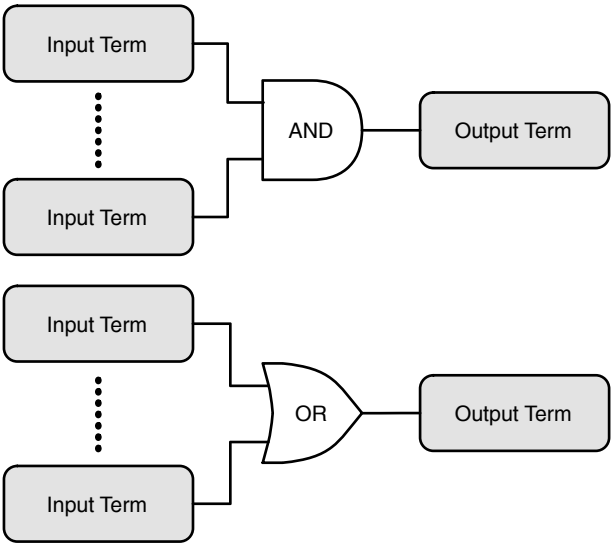


Fig. 4.2. Composition operators used in *detection strategies* represented as logical AND/OR gates.

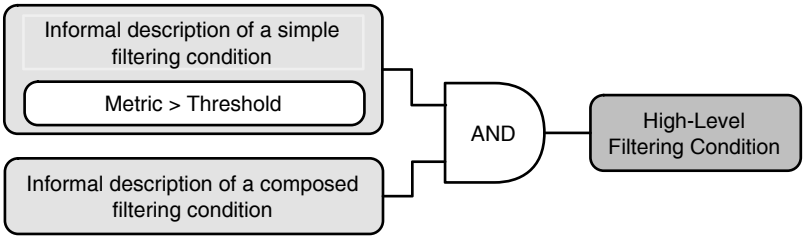


Fig. 4.3. A graphical representation of a *detection strategy*.

they would be applied, yet heuristic knowledge reflects and preserves the experience and quality goals of the developers.

As a consequence, over the last two decades, many authors were concerned with identifying and formulating design principles [Mey88b] [Lis87] [Mar02b], rules [CY91] [Mey88b], and heuristics[Rie96] [JF88] [Lak96] [LR89] that would help developers fulfill those criteria while designing their systems.

An alternative approach to disseminating 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 [BMMM98] 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.

Let us see now, based on the concrete example of the *God Class* [Rie96] design flaw, how *detection strategies* can be defined for a concrete design flaw. The entire process is summarized in Fig. 4.4.

The starting point in defining such a *detection strategy* is given by one (or more) *informal design rules* — like those stated by Riel [Rie96], Martin [Mar02b] or Fowler [FBB+99] — that comprehensively define the design problem, i.e., the disharmony that we want to capture. In this concrete case we start from the three heuristics related to the *God Class* problem, as described by Riel [Rie96]:

> *Top-level classes in a design should share work uniformly.* [...]
> *Beware of classes with much non-communicative behavior.* [...]
> *Beware of classes that access directly data from other classes.*

Step 1: Identify Symptoms

The first step in constructing a *detection strategy* is to break down the informal rules in a correlated set of *symptoms* (e.g., class inflation, excessive method complexity, high coupling) that can be captured by a single metric. In our case the first rule refers to *high class complexity*. The second rule speaks about the level of intra-class communication between the methods of the class; thus it refers to the *low cohesion of classes*. The third heuristic addresses a special type of coupling, i.e., the direct access to instance variables defined in other classes. In this case the symptom is *access of foreign data*.

Step 2: Select Metrics

The second step is to *select proper metrics* that quantify best each of the identified properties. In this context the crucial question is: from where should we take the proper metrics? There are two alternatives:

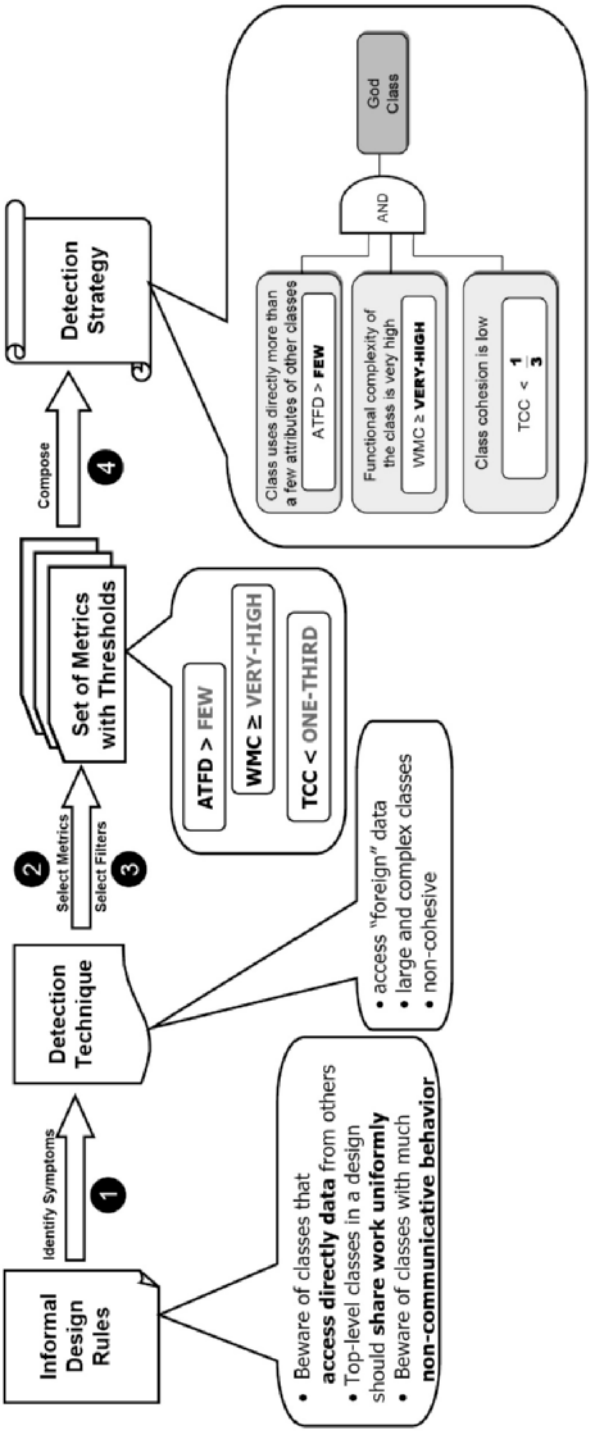


Fig. 4.4. Process of transforming an informal design rule in a *detection strategy*.

1. *Use well-known metrics from the literature*. For example, we could choose a metric from a well-known metrics suite (e.g., the Chidamber&Kemerer [CK94] suite), or from the metrics summarized by various authors (e.g., Lorenz and Kidd [LK94], Henderson-Sellers [HS96], Briand [BDW99, BDW98] etc.)
2. *Define a new metric (or adapt an existing one)*, so that the metric captures exactly one of the symptoms (see previous step) that appears in that design flaw that we intend to quantify.

Our approach is a conservative one, i.e., we try to use as much as possible metrics from the literature, avoiding thus to define new (oftentimes unnecessary) metrics. Yet, in the same time we want to emphasize that, in defining a good *detection strategy*, it is very important not to sacrifice the *exact* quantification of a symptom, just for the sake of using an existing metrics from the literature. In other words, if no adequate metric can be found in the literature, define a new metric that reflects one symptom that needs to be quantified.

For the *God Class* design flaw these properties are class complexity, class cohesion and access of foreign data. Therefore, we choose the following set of metrics[1]:

- *Weighted Method Count* (WMC) is the sum of the statical complexity of all methods in a class [CK94]. We consider McCabe's cyclomatic complexity metric as a complexity measure [McC76, LK94].
- *Tight Class Cohesion* (TCC) is the relative number of methods directly connected via accesses of attributes [BK95, BDW98].
- *Access to Foreign Data* (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods.

Notice that while the first two metrics (i.e., WMC and TCC) are metrics defined in the literature, the last one was defined by us in order to capture a very specific aspect, i.e., the extent to which a class uses attributes of other classes.

Step 3: Select Filters

The next step is to define for each metric the filter that captures best the symptom that the metric is intended to quantify. As mentioned earlier, this implies to (1) pick-up a comparator and (2) to set an

[1] For a precise description of all the metrics used in the book, including the metrics below please refer to Appendix A.

adequate threshold, in conformity with the semantics described in Sect. 2.1.

In our concrete case, the first symptom is referring to *excessively high* class complexity we want to find classes that are complexity outliers. Thus, for the WMC metric we use the $\geq$ (*greater than or equal to*) comparator. How do we find the threshold for extremely high values of the WMC complexity metric? There is no other way than to base it on statistical data related to complexity, as described in Sect. 2.1. Based on the semantic labels described there, we can say now that we will use the *very high* threshold value.

For capturing the aspect of "access to foreign data" we use the $>$ (*greater than*) comparator, whereby the threshold value will be the maximal number of "tolerable" foreign attributes to be used. Thus, the threshold value for ATFD, does not need to be based on statistics, because the metric has a precise semantic: It measures the extent of encapsulation breaking. Based on the rationale presented in Sect. 2.1 "accidental" usage of foreign data, and consequently a *few* such usages are harmless; thus, $ATFD > FEW$.

Eventually, for the *low cohesion* symptom we choose the $<$ (*less than*) comparator. In order to set the proper threshold, we first have to notice that the values of TCC are fractions; thus we can use one of the thresholds with *fraction semantics* summarized in Table 2.3. As this filter must capture non-cohesive classes, we decided to use the *one-third* threshold (see Sect. 2.1), meaning that only one third of the method pairs of the class have in common the usage of the same attribute. If we wanted to capture more extreme cases of non-cohesiveness, we could have used the *one-quarter* threshold.

Step 4: Compose the Detection Strategy

The final step is to correlate these symptoms, using the composition operators described previously. From the context of the informal heuristics as presented by their author in [Rie96], we infer that all these three symptoms should co-exist if a class is to be considered a behavioral *God Class*. Consequently, the final form of the *God Classes detection strategy* is the one depicted in Fig. 4.5.

The Missing Link

Detection strategies are useful to detect problems in object-oriented designs. What they finally produce is a list of *suspects*, i.e., all entities in the system which conform to the applied *detection strategy*.

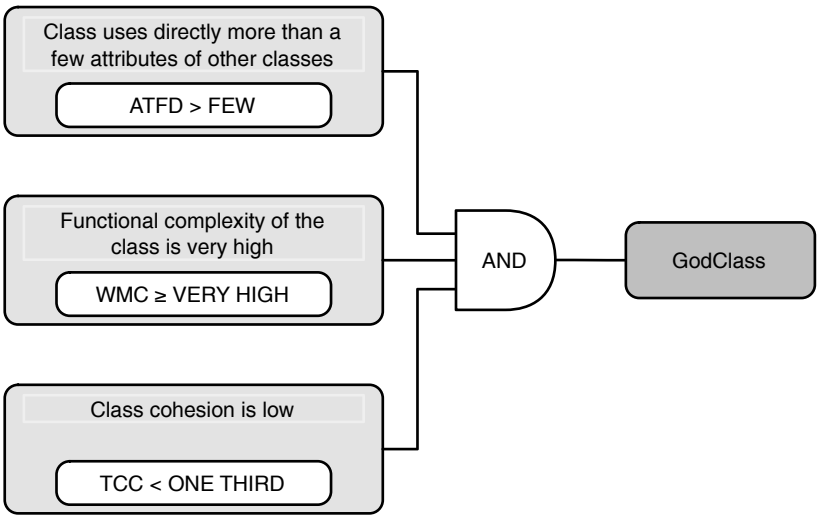


Fig. 4.5. Detection of a *God Class*

These suspects must be manually inspected to find those that cause the most severe problems in the context of the entire system. Applying the numerous *detection strategies* presented in this book (see the next three chapters) would lead to many long code listings that you, the engineer must manually inspect, which is painful and time-consuming process.

Consequently, we need a technique that helps us to (1) *assess quickly* the context of each suspect, (2) decide if the suspect needs to be urgently refactored, and (3) get insights into how this is to be done.

Next, we will introduce a powerful visualization technique called *Class Blueprint* which helps us cover this "missing link" between lists of suspects and the design fragments that need to be improved. *Class Blueprint* is a semantically enriched depiction of the internal structure of classes. It will help us to quickly grasp and discuss the internal design of classes and the way they collaborate with other classes.

Using again a medical metaphor we can say that while *detection strategies* help us to detect abnormal fragments of a system's design, the *Class Blueprint* technique helps us to perform a radiography (or a CAT scan) of suspicious design fragments and decide if and how we need to intervene.

4.2 The Class Blueprint

In this section we present a visualization to assist the understanding of classes by representing a semantically augmented call- and access-graph of the methods and attributes of classes.

We only take into account the internal static structure of a class and focus on the way methods call each other and the way attributes are accessed, and the way the classes use inheritance.

This will help us understand the structure of classes without the need to read all of their code. Classes are difficult to understand because of the following reasons:

1. Contrary to procedural languages, the method definition order in a file is not important [Dek02]. There is no simple and apparent top-down call decomposition, which is necessary to break down the complexity of understanding object-oriented code. This problem is emphasized in the context of integrated development environments (IDEs), which disconnect the class and method definitions from their physical storage medium, e.g., directories and files.
2. Classes are organized in inheritance hierarchies in which at each level behavior can be added, overridden or extended. Understanding how a derived class fits within the context of its base class is complex because late binding provides a powerful instrument to build template and hook methods that allow children's behavior to be called in the context of their ancestors. The presence of late binding leads to yoyo effects: To understand the code by following the call-flow the reader has to browse up and down the hierarchy [WH92, DRW00].

The objective of the *class blueprint* is to help a programmer to understand and develop a mental model of the classes he or she browses and to offer support for reconstructing the logical flow of method calls. In short, a *class blueprint* is a semantically enriched and layered visualization of the control-flow and access structure of classes [LD01, DL05][2].

The Principles of a Class Blueprint

A *class blueprint* is structured according to *layers* that group the methods and attributes. The nodes (varying in size depending on

[2] To locate it in the general context of cognitive models [LPLS96, vMV96], it is intended to support the *implementation plans* at the language level, i.e., working in code chunks, in this case classes and methods.

source code information of the metrics) represent a class's methods and attributes and are colored[3] according to semantic information, e.g., whether a method is abstract, overriding other methods, returning constant values, etc.

The Layered Structure of a Class Blueprint

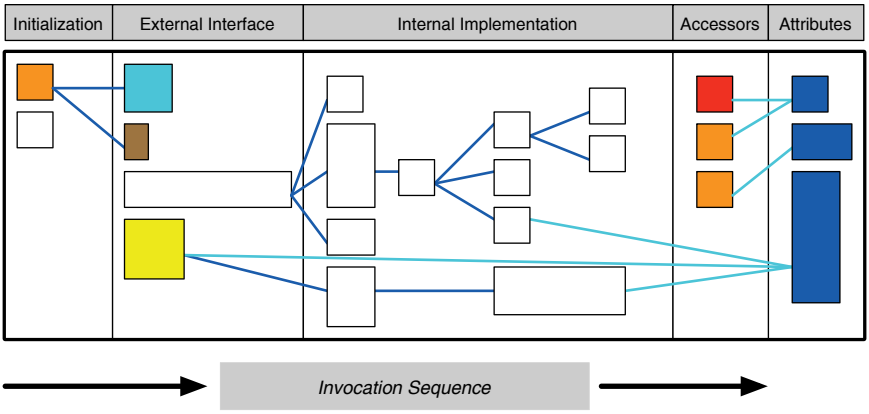


Fig. 4.6. A *class blueprint* decomposes a class into layers.

A *class blueprint* decomposes a class into layers and assigns its attributes and methods to each layer based on the heuristics described below (see Fig. 4.6). The layers support a call-graph notion in the sense that a method node on the left connected to another node on the right is either invoking or accessing the node on the right that represents a method or an attribute.

The layers have been chosen according to a notion of time-flow and encapsulation. The notion of encapsulation is visualized by separating state (to the right) from behaviour (to the left), and distinguishing the public (to the left) from the private part (to the right) of the class' behaviour. Added to this only the actual source code elements are visualized, i.e., we do not represent artificial elements resulting

[3] The colors used in our visualizations follow visual guidelines suggested by Bertin [Ber74], Tufte [Tuf90], Ware [War00], and Pinker [Pin97], e.g., we take into account that the human brain is not capable of simultaneously processing more than a dozen distinct colors.

from a combination/abstraction of source code elements. From left to right we identify the following layers: *initialization layer*, *external interface layer*, *internal implementation layer*, *accessor layer*, and *attribute layer*. The first three layers and the methods contained therein are placed from left to right according to the method invocation sequence, i.e., if method *m1* invokes method *m2*, *m2* is placed to the right of *m1* and connected to an edge.

A *class blueprint* contains the following layers:

1. **Initialization layer.** The methods contained in this first layer are responsible for creating an object and initializing the values of the attributes of the object. A method belongs to this layer if one of the following conditions holds:
   - The method is a constructor.
   - The method name contains the substrings "init(ialize)".
2. **External interface layer.** The methods contained in this layer represent the interface of a class to the outside world. A method belongs to this layer if one of the following conditions holds:
   - It is invoked by methods of the initialization layer .
   - In languages like Java and C++which support modifiers (e.g., public, protected, private) it is declared as *public* or *protected*.
   - It is not invoked by other methods within the same class, i.e., it is a method invoked from *outside* of the class by methods of collaborator classes or subclasses. Should the method be invoked both inside and outside the class, it is placed within the implementation layer .

   We consider the methods of the interface layer to be the *entry points* to the functionality provided by the class. We do not include accessor methods (getters and setters) in this layer, but in a dedicated accessor layer .
3. **Internal implementation layer.** The methods contained in this layer represent the core of a class and are not supposed to be visible to the outside world. A method belongs to this layer if the method is invoked by at least one method of the same class.
4. **Accessor layer.** This layer is composed of accessor methods, i.e., methods whose *sole* task is to get or set the values of attributes.
5. **Attribute layer.** The attribute layer contains the attributes of the class, connected to the method nodes in the other layers by edges representing *access relationships*.

Representing Methods and Attributes

We represent methods and attributes using colored boxes (nodes) of various size and position them within the layers presented previously. We map metric information to the size of the method and attribute nodes, and map semantic information on their colors.

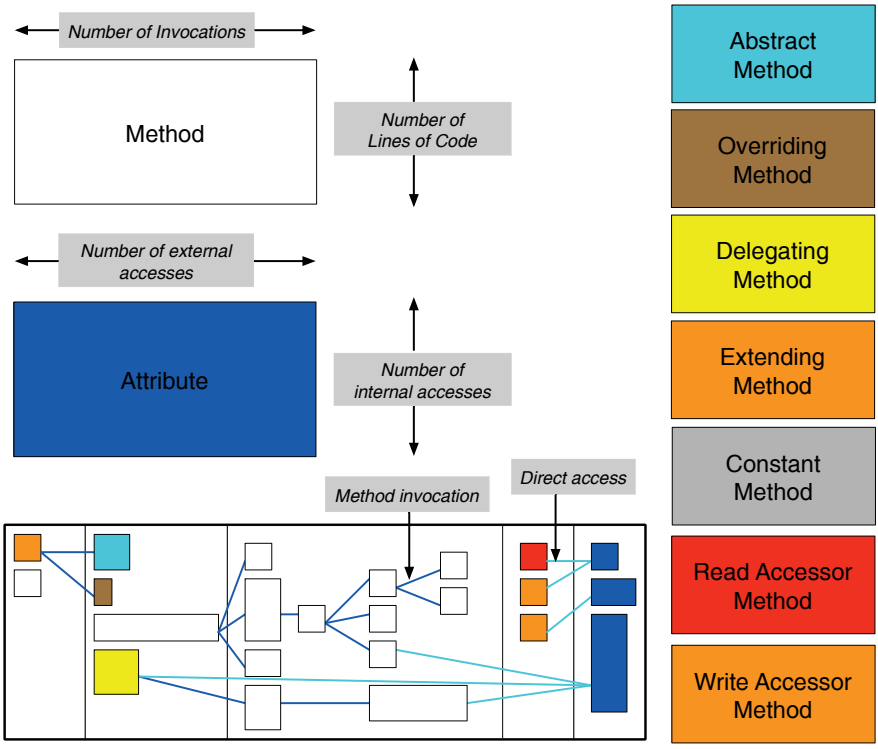


Fig. 4.7. In a *class blueprint* the metrics are mapped on the width and the height of a node. The methods and attributes are positioned according to the layer they have been assigned to.

Mapping metrics information on size. The width and height of the nodes reflect metric measurements of the represented entities, as illustrated in Fig. 4.7. In the context of a *class blueprint*, the metrics used for the method nodes are *lines of code* for the height and *number of invocations* (i.e., number of static invocation going out from the represented node) for the width. The metrics used for the attribute nodes are the number of direct accesses from methods within the

class for the width and the number of direct accesses from methods defined in other classes for the height. This allows one to identify how attributes are accessed.

Description	Color
Attribute	blue node
Abstract method	cyan node
Extending method. A method which performs a *super* invocation.	orange node
Overriding method. A method redefinition *without* hidden method invocation.	brown node
Delegating method, forwards the method call to another object.	yellow node
Constant method. A method which returns a *constant* value.	grey node
Interface and Implementation layer method.	white node
Accessor layer method. Getter.	red node
Accessor layer method. Setter.	orange node
Invocation of a method.	blue edge
Invocation of an accessor. Semantically equivalent to a direct access.	blue edge
Access to an attribute.	cyan edge

Table 4.1. In a *class blueprint* semantic information is mapped on the colors of the nodes and edges.

Mapping semantic information on color. The call-graph is augmented not only by the size of its nodes but also by their color. In a *class blueprint* the colors of nodes and edges represent semantic information extracted from the source code analysis. The colors play an important role in conveying added information [Ber74, Tuf90]. Table 4.1 presents the semantic information we add to a *class blueprint* and the associated colors.

Class Blueprints Exemplified

To show how the *class blueprint* visualization allows one to represent a condensed view of a class's methods, call-flow and attribute accesses, we describe in detail two classes implementing two different domain entities of the Jun framework: The first one defines the concept of a 3D graph for OpenGL mapping and the second is a rendering algorithm. We present the blueprints and some pieces of code to show how the graphical representation is extracted from the source code and how the graphical representation reflects the code it represents, building a trustable model.

To help the reader to understand the first *class blueprint* we also show on the right of the figure a blueprint without metrics in which the method names are shown on the boxes that represent them.

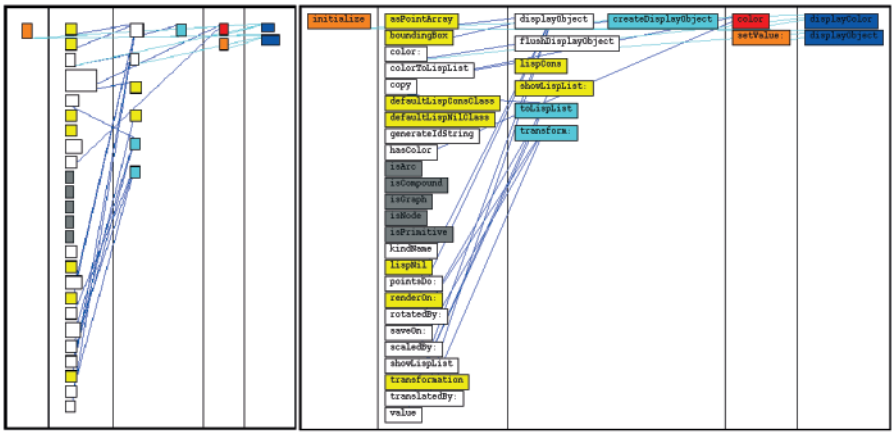


Fig. 4.8. Left: An actual class blueprint visualization of the class JunOpenGL3dGraphAbstract, a class which represents 3D graphs in OpenGL. Right: The same class displayed with method names for illustrating how the methods call each other.

The left part of Fig. 4.8 shows the blueprint of a Smalltalk class named Jun-OpenGL3dGraphAbstract which we describe hereafter. As the named blueprint on the right in Fig. 4.8 shows, this kind of representation does not scale well in practice; additionally, metrics information is not reflected in a named blueprint (i.e., the width and height of nodes is not correlated with metric value). Therefore it is not used in this book.

The code shown is Smalltalk code; however, in order to understand the code sequence being fluent in Smalltalk is not a must as we are only concerned with method invocations and attribute accesses.[4]

Example 1: An Abstract Class

The *class blueprint* shown in Fig. 4.8 has the following structure:

- **One initialization layer method.** This method, called initialize, is positioned on the left. As shown, it extends (invokes) a superclass

[4] In Smalltalk, attributes as local variables are read simply by using the attribute name in an expression. They are written using the := construct. In a first approximation, messages follow the pattern receiver methodName1: arg1 name2: arg2 which is equivalent to the Java/C++syntax receiver.methodName1name2(arg1, arg2). Hence bidiagNorm := self bidiagonalize: superDiag assigns to the variable bidiagNorm the result of the method bidiagonalize.

method with the same name, hence the node color is orange. It directly accesses two attributes, as the cyan line shows. The code of the method initialize is as follows:

```
initialize
   super initialize.
   displayObject := nil.
   displayColor := nil
```

- **Several external interface layer methods.** Note that many of them have a yellow color, i.e., they delegate the functionality. The following method asPointArray is a delegating method:

```
asPointArray
   ^ self displayObject asPointArray
```

The five grey nodes in the interface layer are methods returning constant values as illustrated by the following method isArc. This method illustrates a typical practice to share a default behavior among the hierarchy of classes.

```
isArc
   ^ false
```

- **A small internal implementation layer with two sub-layers.** This layer shows that the blueprint granularity resides at the method level, as the visualization does not specifically represent control flow constructs. The method displayObject performs a lazy initialization, i.e., it initializes the attributes only when the attributes are accessed and acts as an abstract template method by calling the method createDisplayObject which is abstract and thus represented as a cyan node. The method createDisplayObject should then be redefined in the subclasses.

```
displayObject
   displayObject isNil ifTrue:
      [ displayObject := self createDisplayObject ].
   ^ displayObject


createDisplayObject
   ^ self subclassResponsibility
```

- **Two accessors.** There is a read-accessor, color, displayed as the red accessor node and a write-accessor, setValue: displayed as the rightmost orange accessor node.

- **Two attributes.** Note that the read-accessor reads one attribute, while the write-accessor writes the other one. However, no method uses the write-accessor. The attributes are also directly accessed: the initialize method accesses both, while two other methods also directly access the attributes. which is an inconsistent coding practice.

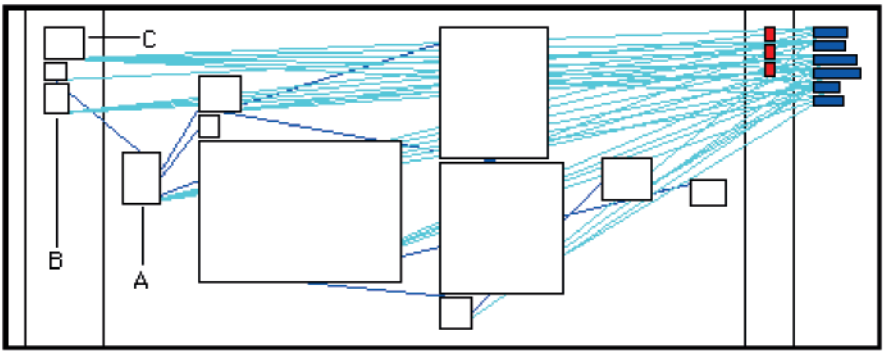


Fig. 4.9. A blueprint of the class JunSVD. This *class blueprint* shows patterns of the type *Single Entry*, *Structured Flow* and *All State*.

Example 2: An Algorithm

The *class blueprint* presented in Fig. 4.9 displays the class JunSVD implementing the algorithm of the same name. Looking at the blueprint we get the following information.

- **No initialization layer method.** The left layer is empty.
- **Three external interface layer methods.** Two of them directly access the attributes of the class. We also see that the second external interface layer method is actually an entry point to all the methods in the internal implementation layer.
- **An internal implementation layer composed of nine methods in five sub-layers.** The class is actually written in a clearly structured way. Therefore the *class blueprint* can also be used to infer a reading order of the methods contained in this class. The blueprint shows that the node *A* which represents the method compute (shown hereafter) invokes the methods bidiagonalize:, epsilon and diagonalize:with:.

```
compute
    | superDiag bidiagNorm eps   |
    m := matrix rowSize.
    n := matrix columnSize.
    u := (matrix species unit: m) asDouble.
    v := (matrix species unit: n) asDouble.
    sig := Array new: n.
    superDiag := Array new: n.
    bidiagNorm := self bidiagonalize: superDiag.
    eps := self epsilon * bidiagNorm.
    self diagonalize: superDiag with: eps.
```

- **Three read-accessor methods.** Although three read-accessors have been defined, they are not used by methods of this class, because they do not have any incoming edges that would exemplify their use.
- **Six attributes.** All the attributes in this class are accessed by several methods, i.e., all the state of the class is accessed by the methods. The blueprint also reveals that the attributes are heavily accessed. The nodes marked as *A*, *B* and *C* consistently access *all* the attributes matrix, n, m, sig, v and u. To understand how this particular behavior is possible we show the code of the method generalizedInverse (*C*). After reading the code we easily understand that this particular behavior for a class is normal for an algorithm and we mentally acknowledge that the other methods are built in a similar fashion.

```
generalizedInverse
    | sp |
    sp := matrix species new: n by: m.
    sp doIJ: [:each :i :j |
      sp row: i column: j put:
        ((i = j and: [(sig at: j) isZero not])
            ifTrue: [(sig at: j) reciprocal]
            ifFalse: [0.0d])].
    ^ (v product: sp) product: u transpose
```

This example shows that the blueprint visualization conveys information which is otherwise hard to notice: all attributes are accessed by the methods. This is an example of how the approach supports opportunistic code reading. First the reader is intrigued by the regularity of the accesses, then reads one method and understands that the methods implement an algorithm. The reader can now extrapolate this knowledge to the other methods of the class.

Example 3: Class Blueprints and Inheritance

Understanding classes in the presence of inheritance is difficult as the flow of the program is not local to a single class but distributed over hierarchies, as mentioned by Wild [WH92] and Lange [LN95]. In the context of inheritance we visualize every *class blueprint* separately and put the subclasses below the superclasses according to a simple tree layout.

In Fig. 4.10 we see a concrete inheritance hierarchy of class blueprints. The superclass defines some behavior that is then specialized by each of the three subclasses named JunColorChoiceHSB, JunColorChoiceSBH and JunColorChoiceHBS. The blueprint of this hierarchy reveals that the subclasses have been developed to satisfy the implementation needs of the superclass: they do not define any extra behavior; it is the superclass that must be analyzed to understand the whole hierarchy.

We see that the root class defines several abstract methods (denoted by the cyan color) that represent color components such as brightness, hue and color and which are overridden (denoted by the brown color) in the three small subclasses. As there is the same number of brown nodes as cyan ones, there is a good chance that the subclasses are concrete classes.

The method named color is a template method that calls three abstract methods as confirmed by the definition of the method color:

```
color
   ^ ColorValue hue: self hue
                saturation: self saturation
                brightness: self brightness
```

We see that the methods xy: (B) and xy (C), play a central role in the design of the class as they are both called by several of the methods of each subclass, as confirmed by the following method of the class JunColorChoiceSBH:

```
JunColorChoiceSBH>>brightness: value
   ((value isKindOf: Number) and:
      [0.0 <= value and: [value <= 1.0]])
        ifTrue: [self xy: self xy x @ 1 - value]
```

This example shows again that the blueprint conveys information which is otherwise hard to notice, e.g., the fact that all the subclasses of the root classes implement only methods which override methods

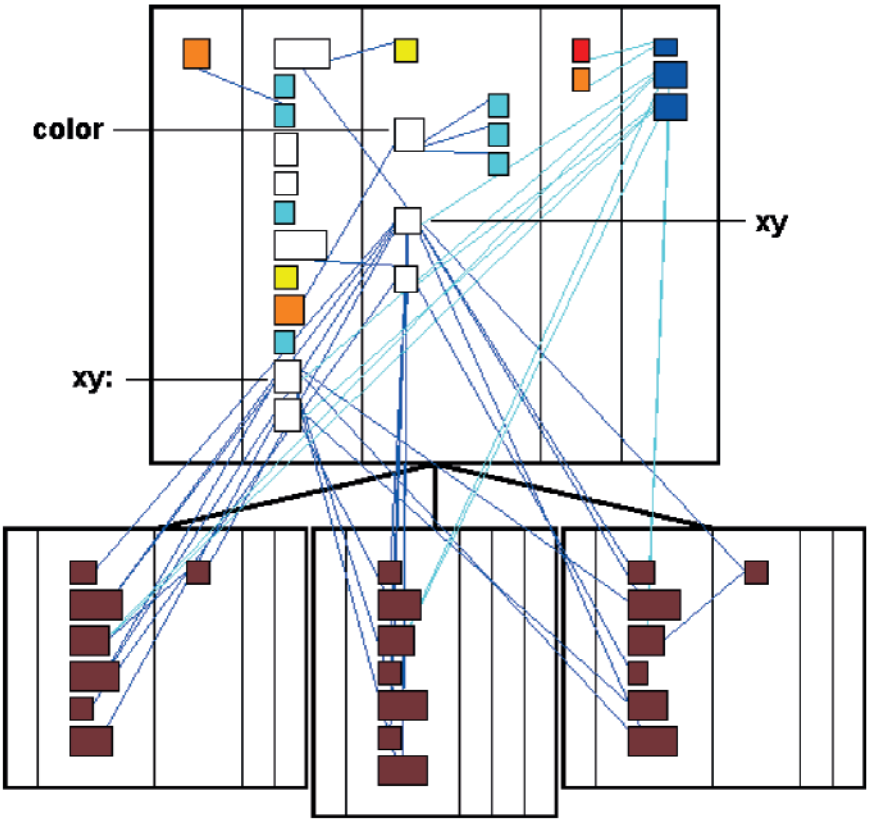


Fig. 4.10. A *class blueprint* visualization of an inheritance hierarchy with the class JunColorChoice as root class. The root class contains an *Interface* visual pattern, while each of the subclasses is a pure *Overrider*. Furthermore, each subclass is a pure *Siamese Twin*.

in the superclass, or it helps to detect the template method design pattern present in the root class.

All these examples illustrate how the blueprints help a software engineer to: (1) build a mental image of the class in terms of method invocations and state access, (2) understand the class/subclass roles, and (3) identify key methods.

Blueprints act as revealers in the sense that they raise questions, support hypotheses, or clearly show important information. When questions are raised, code reading helps confirm the information provided by the visualization. Code reading is not always necessary, but can be used sparingly on identified methods. There is a definitive

synergy between the visual images generated by the blueprint and the code reading. Class blueprints allow one to characterize classes but also represent an important means of communication.

Example 4: Class Blueprints and Design Problems

Class Blueprints provide us with a powerful visual means to inspect the suspects detected by the detection strategies. For example, by applying the *God Class* detection strategy (see page 51) on a case study we found several suspects, one of which is class Modeller.

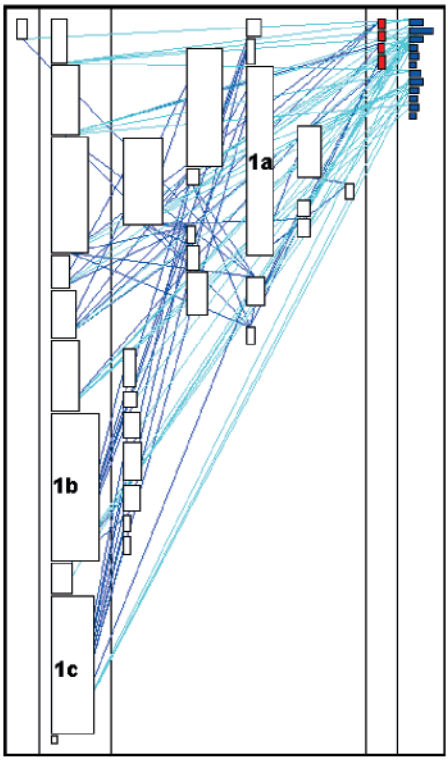


Fig. 4.11. The *class blueprint* of a *God Class* suspect.

By building the *class blueprint* for this class (see Fig. 4.11) we can immediately see that Modeller is not a class with an excessive number of methods, but has a certain number of considerably large and complex methods (3 methods are longer than 100 lines of code, the longest one addDocumentationTag (annotated as 1a in the figure)

is 150 lines code and invoked by three other methods, two of which are the second and third longest methods in this class: addOperation (1b, 116 LOC) and addAttribute (1c, 108 LOC). The *class blueprint* reveals other disharmonies in this class: there are 12 attributes in this class, all of them private (which is good), but there are "only" 4 accessor methods. Moreover, the attributes are accessed both directly and indirectly (using the accessors), denoting a certain inconsistency or lack of access policy.

4.3 Conclusions and Outlook

In this chapter we presented two approaches which will allow us to evaluate the design of object-oriented software systems, the *Detection Strategy* and the *Class Blueprint*:

Detection Strategy: It provides us with a means to detect flawed (from a design point of view) entities in object-oriented systems. The design strategies produce lists of suspects that comply with specific heuristics encoded with metrics.

Class Blueprint: It provides us with a powerful visual means to inspect the suspects detected by the *detection strategies*.

In the beginning of this chapter (see 46) we argued that metrics can help to evaluate designs, but those have to be *meaningful* metrics that are put in the context of rules, best practices and heuristics that express the *harmony of a design*.

Although we partially agree with Fowler stating that "no set of metrics rivals informed human intuition" [FBB+99], there is a big disadvantage: human intuition does not *scale* with the dimensions of today's software systems. Therefore, in order to find and improve disharmonious design fragments in the next three chapters we employ *detection strategies* and the *class blueprint*.

Consequently in the remaining chapters, we present in detail 11 such *design disharmonies*. For each of them we describe the *detection strategy* that helps to detect them automatically using metrics, we look at selected examples using the *class blueprint*, and conclude each disharmony with a discussion of how to *cure* flawed entities using refactorings.

Based on the harmony aspects identified in Sect. 4, we divide the 11 *design disharmonies* in three categories, i.e., *identity collaboration* and *classification* disharmonies:

Identity Disharmonies (Chapter 5): God Class(80), Brain Class(97), Feature Envy(84), Brain Method(92), Data Class(88), Duplication(102)
Collaboration Disharmonies (Chapter 6): Dispersed Coupling(127), Intensive Coupling(120), Shotgun Surgery(133)
Classification Disharmonies (Chapter 7): Refused Parent Bequest(145), Tradition Breaker(152)

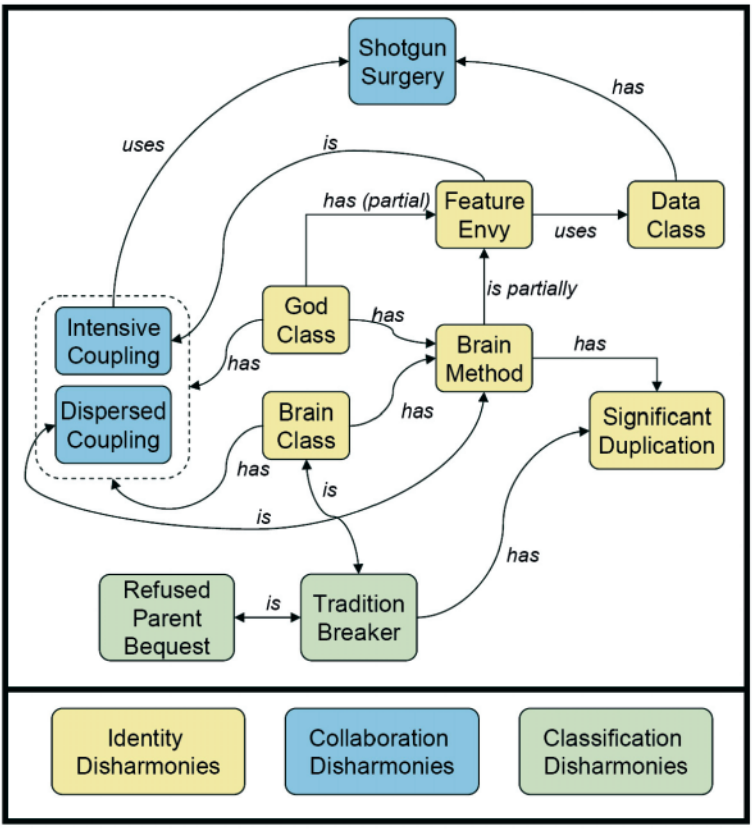


Fig. 4.12. Disharmonies and their correlations.

Each of the following chapters has four major parts:

1. *Harmony Rule(s).* As mentioned before, disharmonies are deviations from a set of principles, rules and heuristics that specify what harmony means. Therefore, before presenting a catalogue of disharmonies, we summarize in the form of one or more *harmony*

rules those aspects of harmony that we took into account when building the catalogue of disharmonies. These *harmony rules* are a concise distillation of various design rules and heuristics found in the literature (e.g., [Rie96, Mar02b, JF88, Lak96, Mey88b, Lis87]).

2. *Overview of Detected Disharmonies*. Most of the times design disharmonies do not appear in isolation. Therefore, before presenting the disharmonies one by one, we provide a brief overview in which we reveal the most common correlations between the various disharmonies. Apart from discussing the correlations, we provide in each case a picture that captures the *web of correlations* involving the disharmonies presented in that chapter. As a sneak preview, in Fig. 4.12 you can see all disharmonies and their most common correlations. complete *web of correlations*
3. *Catalogue of Disharmonies*. The central part of each chapter consists of a catalogue of specific disharmonies that can be detected using a metrics-based approach. Each disharmony is described in a in a pattern-like format.
4. *Summary*. Each chapter ends with a suite of practical guidelines on detecting and recovering from the disharmonies presented in the chapter.

5

Identity Disharmonies

Identity disharmonies are design flaws that affect single entities such as classes and methods. The particularity of these disharmonies is that their negative effect on the quality of design elements can be noticed by considering these design elements *in isolation*.

5.1 Rules of Identity Harmony

As mentioned at the end of Chapter 4 before presenting the various identity disharmonies, let us first take a closer look on the most important harmony rules related to a single design entity.

We identified three distinct aspects that contribute to the identity (dis)harmony of a single entity: its size, its interface and its implementation. We summarize each of these aspects in the form of a rule, a rationale and a set of practical consequences. The three rules of identity harmony that we defined are:

Operations and classes should have a harmonious size i.e., they should avoid both size extremities

Each class should present its identity (i.e., its interface) by a set of services, which have one single responsibility and which provide a unique behavior

Data and operations should harmoniously collaborate within the class to which they semantically belong

---

Proportion Rule

Operations and classes should have a harmonious size, i.e., they should avoid both size extremities

Rationale

When considering quality and harmony the first aspect we think about is proportion. The same applies to object-oriented software design. While this first rule is simple to understand it is crucial to follow it. Most of the maintenance and reuse problems come from an unbalanced distribution of a system's complexity (responsibilities) among classes [Rie96, WBM03] or among operations [FBB+99]. This does not mean that all classes or operations must have the same size; rather, it warns us about the danger of going to extremes. Both extremes can be dangerous: too large classes or operations are a maintenance nightmare, while many tiny classes are in most cases a sign of *class proliferation* and hinder understanding. In the same manner, while it is desirable to have slim operations, sometimes this is abused and we end up with an excessive number of methods, that again hampers maintenance.

Presentation Rule

> *Each class should present its identity (i.e., its interface) by a set of services, which have one single responsibility and which provide a unique behavior*

Rationale

This rule encourages a balanced distribution of a system's intelligence among classes [Rie96, WBM03] and was the underlying idea behind CRC cards (class, responsibility, collaborator) [BS97]. The aim of the rule is to focus each class on a single task (responsibility), expressed in terms as a set of services (i.e., a set of public methods). The rule makes sure that each concrete piece of functionality is implemented once and only once in the system to avoid code duplication.

Practical Consequences

- **Provide services and hide data** – *A class should present itself to others only in terms of a set of provided services (i.e., public methods). Never let a class present itself in terms of its data, as this breaks encapsulation and consequently spoils the maintenance benefits of object-oriented design.*

- **Take responsibility** – *Most non-abstract services of a class should be responsible for implementing a piece of the class's functionality. A class might have some* delegator *(i.e., methods that just forward the call to another method) and some* accessor *methods, but the number of such methods should be limited in each class.*

- **Keep services cohesive** – *Services provided by a class should be focused on one single responsibility. The set of services of a class should have limited size and a high usage cohesion.*

  This is a restatement of the *Interface Segregation Principle* [Mar02b] which states that the clients of a class should not be forced to

depend on interfaces that they do not use. Although this consequence also concerns the size of the interface of a class, the main aspect here is not the proportion, but the avoidance of an eclectic interface.

- **Be unique** – *When classes have a harmonious identity, then each piece of concrete functionality has a unique place, i.e., it is implemented once and only once [Bec00]. Consequently, code duplication is avoided.*

Implementation Rule

Data and operations should collaborate harmoniously within the class to which they semantically belong

Rationale

One of the cornerstones of the object-oriented paradigm is encapsulation that makes sure that the data and the operations are kept together. An abstraction (e.g., a class) is harmonious if its operations use most of data most of the time [Rie96], i.e., if most attributes of a class are used together in most methods of that class. This rule does not allow every piece of the system to be visible and accessible by any other part of the system. Applying the law of Demeter [LH89] stresses that locality of data access is important to avoid to have ripple effects when code changes. Hence keeping data and behavior together is a good practice and an important refactoring [DDN02].

Practical Consequences

- **Operations belong to classes** – *Every operation should belong to a class. Thus, avoid as much as possible global operations.*

- **Keep data close to operations** – *Data and the operations that use it most should be placed as close as possible to one another. In other words, data (e.g., attributes, local variables, etc.) should stay in the class or method where they are used the most.*

- **Distribute complexity** – *The functionality provided by a class should be distributed among its operations in a balanced manner.*

- **Operations use most attributes** – *Within the same class, most operations should collaborate and use most of the data most of the time [Rie96]. Thus, avoid abstractions with disjunct sets of behavior and data.*

5.2 Overview of Identity Disharmonies

The most frequent and easily recognizable sign of an identity disharmony is excessive size and complexity of a class and its methods (*Proportion Rule*). Any investigation that intends to assess and improve the identity harmony of a system usually starts with those classes and methods that stand out due to their size. This is very important, because as we will see also in Sect. 5.9 the process of recovering from design problems uses these outlying design fragments as a starting point.

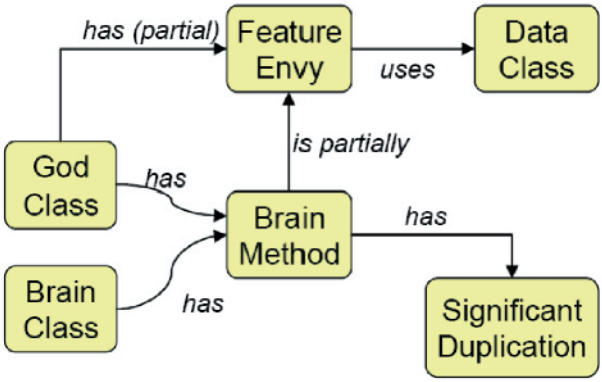


Fig. 5.1. Correlation web between identity disharmonies.

In the remainder of this chapter we present *detection strategies* that capture oversized and overcomplex methods (Brain Method(92)) and the classes that host them (Brain Class(97)). In many cases these outliers are caused by the presence of code duplication; consequently we check for code duplication within classes (Duplication(102)) with excessive size and complexity (see Fig. 5.1).

Another sign of disharmonious identity is the non-cohesiveness of behavior (*Presentation Rule* and *Implementation Rule*) and the tendency to attract more and more features, to gather more and more services (Riel calls such a disharmony a God Class(80) [Rie96]). We defined a *detection strategy* to detect such classes. The more a class tends to become a God Class(80), the more the other classes communicating with it tend to become simple data providers. A data provider does not offer much functionality; instead it merely provides raw data and tends to become a Data Class(88) [Rie96, FBB+99]. As an imme-

diate consequence, the methods of the (God) classes, which use the foreign data, smell of Feature Envy(84) [FBB+99], being more interested in the attributes of other classes than those of their own class.

5.3 God Class

Description In a good object-oriented design the intelligence of a system is uniformly distributed among the top-level classes [Rie96]. The God Class design flaw refers to classes that tend to centralize the intelligence of the system. A God Class performs too much work on its own, 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 is comparable to Fowler's *Large Class* bad smell [FBB+99].

Applies To Classes.

Impact God Class is potentially harmful to a system's design because it is an aggregation of different abstractions and (mis)use other classes (often mere data holder) to perform its functionality (see *Proportion* and *Implementation Rules*). Most of the time they are against the basic principles of object-oriented design which is that one class should have one responsibility. At this point it is important to mention that a God Class is a *real* problem if it hampers the evolution of the software system. Thus a class that has the structural characteristics of a God Class but that resides in a stable and untouched part of the system does *not* pose a problem!

Detection The detection of a God Class is based on three main characteristics (Fig. 5.2):

1. They heavily access data of other simpler classes, either directly or using accessor methods.
2. They are large and complex
3. They have a lot of non-communicative behavior i.e., there is a low cohesion between the methods belonging to that class.

We first detect the classes that strongly depend on the data of other classes, as this is the most significant symptom of a God Class. After that, we filter the first list of suspects by eliminating all the small and cohesive classes. Small classes are eliminated because they are less relevant, while cohesive classes are excused because a high cohesion is a sign of internal harmony between the parts of the class. The *detection strategy* is composed of the following heuristics:

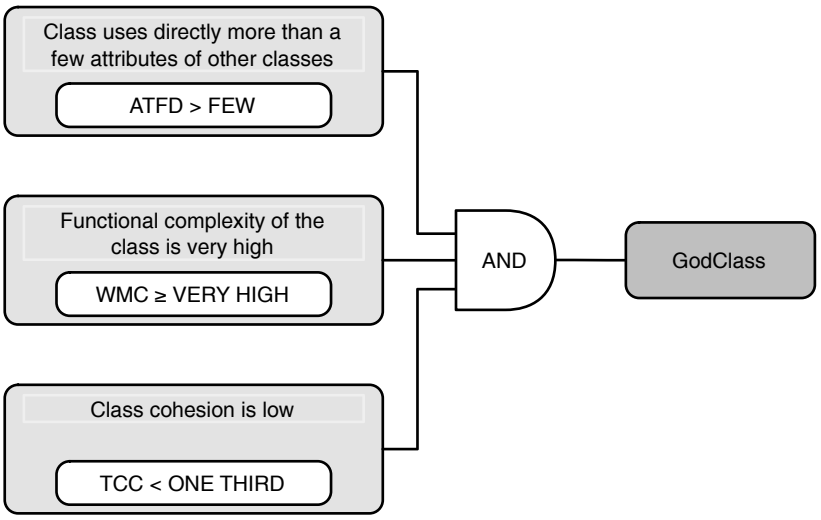


Fig. 5.2. The God Class detection strategy

1. **Class uses directly more than a few attributes of other classes.** Since ATFD measures how many foreign attributes are used by the class, it is clear that the higher the ATFD value for a class, the higher is the probability that a class is (or is about to become) a God Class.
2. **Functional complexity of the class is very high.** This is expressed using the WMC (Weighted Method Count) metric.
3. **Class cohesion is low.** As a God Class performs several distinct functionalities involving disjunct sets of attributes, this has a negative impact on the class's cohesion. The threshold indicates that in the detected classes less than *one-third* of the method pairs have in common the usage of the same attribute.

Example

The general design of *ArgoUML* is good enough so that we could not identify a pure God Class i.e., a class controlling the flow of the application and concentrating all the crucial behavior, which would indicate a clear lack of object-oriented design. However, certain classes in *ArgoUML* acts as a black hole attracting orphan functionalities. Such classes are also detected by the metrics presented above and are still a design problem. A class of *ArgoUML* which clearly stands out is the huge class ModelFacade (see Fig. 3.12). This class implements 453

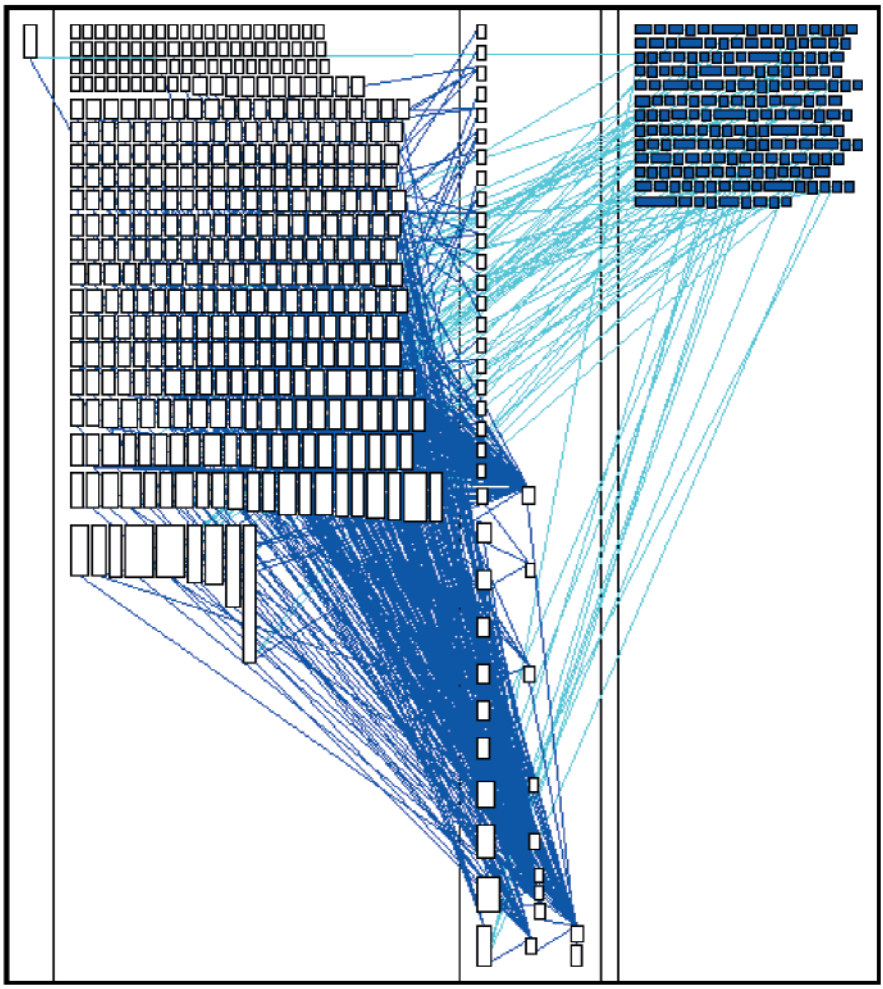


Fig. 5.3. The *Class Blueprint* of ModelFacade

methods, defines 114 attributes, and is more than 3500 lines long. Moreover, all methods and all attributes are static. Its name hints at being an implementation of the *Facade* Design Pattern [GHJV95], but it has become a sort of black hole of functionality. In Fig. 5.3 we see its *Class Blueprint* with a modified layout for the methods and attributes to make this *Class Blueprint* fit on one screen. Looking at the *Class Blueprint* for this class it seems that the developers use it for everything that does not fit into other classes, but the downside is that this class is like a tumor within this system and can only

be removed if somebody makes a great effort to break away pieces of functionality and separate them into other classes. We see from the visualization that many invocations are directed towards distinct methods, pointing to subsets of connected methods that can be extracted.

Refactoring

Refactoring a God Class is a complex task, as this disharmony is often a cumulative effect of other disharmonies that occur at the method level. Therefore, performing such a refactoring requires additional and more fine-grained information about the methods of the class, and sometimes even about its inheritance context. A first approach is to identify clusters of methods and attributes that are tied together and to extract these islands into separate classes. Split Up God Class [DDN02] proposes to incrementally redistribute the responsibilities of the God Class either to its collaborating classes or to new classes that are pulled out of the God Class. Feathers [Fea05] presents some techniques such as Sprout Method, Sprout Class, Wrap Method to be able to test legacy system that can be used to support the refactoring of God Classes.

5.4 Feature Envy

Description Objects are a mechanism for keeping together data and the operations that process that data. The Feature Envy design disharmony [FBB+99] refers to methods that seem more interested in the data of other classes than that of their own class. These methods access directly or via accessor methods a lot of data of other classes. This might be a sign that the method was misplaced and that it should be moved to another class.

Applies To Methods.

Impact Data and the operations that modify and use it should stay as close together as possible. This data-operation proximity can help minimize ripple effects (a change in a method triggers changes in other methods and so on; the same applies for bugs, i.e., in case of a poor data-operation proximity bugs will also be propagated) and help maximize cohesion (see Implementation Rule).

Detection 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 investigation is defined. Feature Envy happens when the envied data comes from a very few classes or only one class. The *detection strategy* (Fig. 5.4) in detail is:

1. **Method uses directly more than a few attributes of other classes.** We use the ATFD[1] (Access To Foreign Data) metric for this.
2. **Method uses far more attributes from other classes than its own.** We use the LAA (Locality of Attribute Accesses) metric for this.
3. **The used "foreign" attributes belong to very few other classes.** We use the FDP (Foreign Data Providers) metric for this. The rea-

[1] In defining the God Class(80) *detection strategy* we also used a metric called ATFD, which counts how many distinct attributes from other classes are accessed by the measured design entity. The only difference is that in the God Class(80) the metric is defined for a class entity, while here it is defined for a method entity.

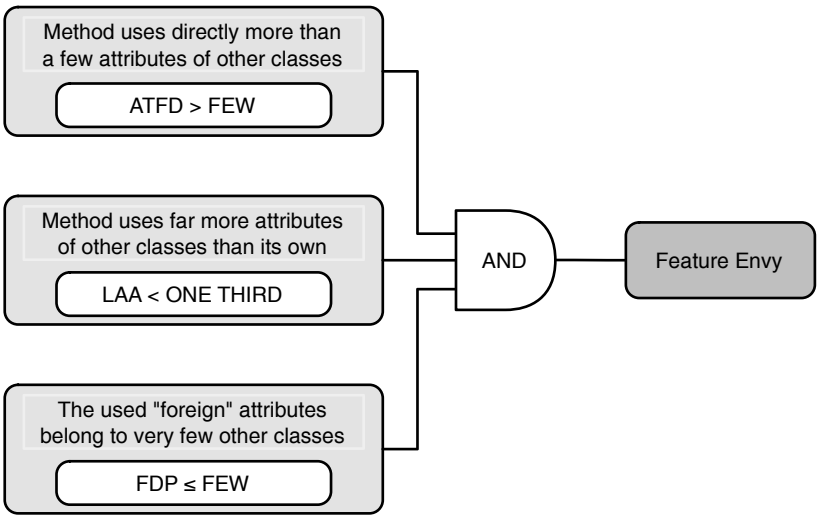


Fig. 5.4. Detection strategy for Feature Envy.

son for introducing this condition is that we want to make a distinction between a method who uses directly data from many different classes, and the case where the method envies especially 1-2 classes. In the first case, it might be that the method acts like a controller [Rie96] and/or that it is a Brain Method(92). But in detecting Feature Envy we are more interested in the second case, as the essence of this disharmony is that the affected method is simply misplaced, and this is reflected by a well targeted dependency on the data from another class.

In analyzing this design disharmony two alternative detection approaches could be used:

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 FDP 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 rely. 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.

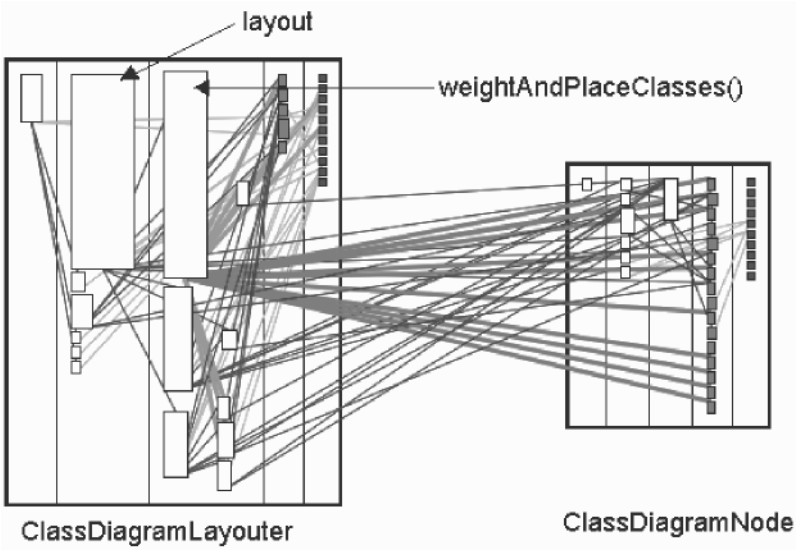


Fig. 5.5. ClassDiagramLayouter is envying the features of ClassDiagramNode. In red we colored the invocations that weightAndPlaceClasses performs towards ClassDiagramNode, while in green we see its class-internal invocations and accesses.

Example

Looking again at the *ArgoUML* system we find a good example of Feature Envy, namely the weightAndPlaceClasses method in the class ClassDiagramLayouter. Although the method uses data from its own class it envies the data encapsulated in the class ClassDiagramNode (i.e., by accessing the data heavily via a large number of accessor methods), as depicted in Fig. 5.5. Looking closer at the figure we notice three interesting aspects:

1. The weightAndPlaceClasses method is excessively large.
2. The envied class, i.e., ClassDiagramNode, contains almost no functionality, but just data which is made accessible via the accessor methods (marked in red). A problem is that the envied class

does not provide a clean interface to clients to offer them functionality, but it exposes its attributes, which is questionable.
3. Looking at ClassDiagramLayouter we notice that the method layout is using several attributes from ClassDiagramNode.

These three observations illustrate the most significant aspect about Feature Envy, namely that it is a sign of an improper distribution of a system's intelligence. While the ClassDiagramLayouter is an excessively complex class (i.e., a Brain Class(97)) with a very high average complexity of methods (AMW = 5.25) the ClassDiagramNode class contains very little functionality, being a Data Class(88). For comparison let us mention that the average complexity of methods in this class, namely the values of the AMW metric, is as low as 1.33.

Finally, as we see in this example, often a Feature Envy method also has some dependencies on its own class, and not only on the envied class. This tells us that in order to recover from this problem, it is very rare that we can move the whole method to the other class. Rather, it is more often that a part of the method can be extracted and moved to the envied class (for a more detailed discussion see also Sect. 5.9).

Refactoring

This problem can be solved if the method is moved into the class to which it is coupled the most. 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 move the newly created method into the envied class. If the method envies two different classes, you should move it to the one that it uses most.

Oftentimes, the class that a method affected by Feature Envy is depending on is a class with not much functionality, sometimes even a Data Class(88). If this is a case then moving the Feature Envy method to that class is even more a desirable refactoring, as it re-balances the distribution of functionality among class and improves the data-behavior locality.

The concrete refactoring technique for Feature Envy is based on the Move Method and Extract Method refactorings [FBB+99]. Furthermore, the Move Behavior Close to the Data reengineering pattern [DDN02] discusses the steps to follow to move behavior close to the data it uses and the potential difficulties.

5.5 Data Class

Description Data Classes [FBB+99] [Rie96] are "dumb" data holders without complex functionality but other classes strongly rely on them. The lack of functionally relevant methods may indicate that related data and behavior are not kept in one place; this is a sign of a non-object-oriented design. Data Classes are the manifestation of a lacking encapsulation of data, and of a poor data-functionality proximity.

Applies To Classes.

Impact The principles of encapsulation and data hiding are paramount to obtain a good object-oriented design. Data Classes break design principles because they let other classes see and possibly manipulate their data, leading to a brittle design (Presentation Rule). Such classes reduce the maintainability, testability and understandability of a system.

Detection We detect Data Classes based on their characteristics (see Fig. 5.6): we search for "lightweight" classes, i.e., classes which provide almost no functionality through their interfaces. Next, we look for the classes that define many *accessors* (get/set methods) and for those who declare data fields in their interfaces. Finally, we confront the lists and manually inspect the lightweight classes that declare many public attributes and those that provide many accessor methods. The *detection strategy* in detail is:

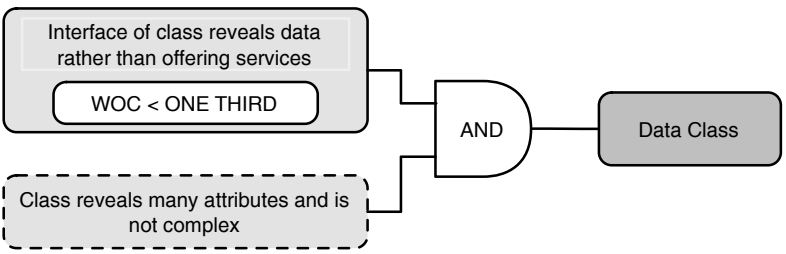


Fig. 5.6. The Data Class detection strategy.

1. **Interface of class reveals data rather than offering services.** The large majority of the class's interface is exposing data rather than providing services. We use the WOC (Weight Of Class) metric for this.

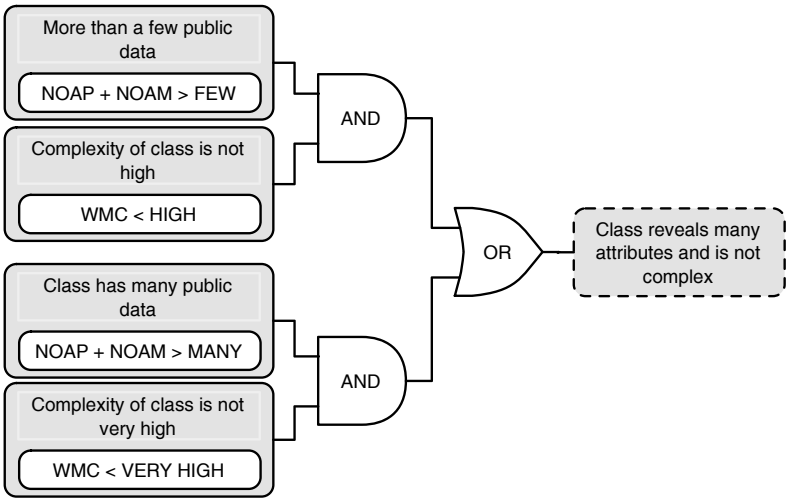


Fig. 5.7. Data Class reveals many attributes and is not complex.

2. **Class reveals many attributes and is not complex.** The WOC metric makes sure that the interface of the class is occupied mainly by data and accessor methods. We also want to be sure that the *absolute* number of these encapsulation breakers is high. We differentiate between two cases (see Fig. 5.7):
   a) The classical Data Class is not very large, has almost no functionality, and only provides some data and data accessors. In this case the class has not a high WMC (Weighted Method Count) value, and we cannot expect to find much public data. Therefore, the only request is that the class has more than a FEW public data holders, expressed using the NOPA (Number Of Public Attributes) and NOAM (Number Of Accessor Methods) metrics.
   b) The other case is that of a rather large class that apparently looks "normal" (i.e., it does also define some functionality), except for the fact that its (large) public interface contains, apart

from the provided services, a significant number of data and data accessors. For this case, in order to consider the class a Data Class, we require that it provides MANY public data. At the same time, we allow the complexity of the class (WMC) to be considerably high, up to the limit of excessively high (because a class with extremely high complexity does not conceptually fit the Data Class term).

Example

In *ArgoUML* we identified several examples of Data Classes, one of which is the class Property (see Fig. 5.8).

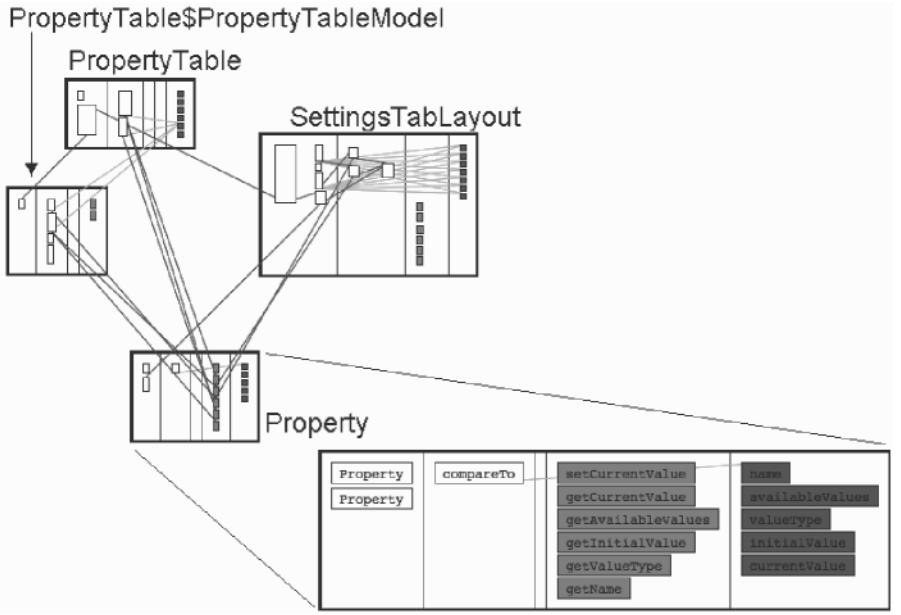


Fig. 5.8. An example of a Data Class: Property.

The name itself already suggests that the class is not really modelling an abstraction in the system, but rather keeps together a set of data. Looking closer, we notice that the class has five attributes. In Fig. 5.8 we depict the Property class together with the classes that use its data. In spite of the fact that all attributes are declared as private, the class is still a pure data holder, due to the fact that all (but one) of its methods are accessors (see methods in red). Thus, the class has no behavior, it just keeps some data, used by three other classes. Although none of the involved classes are large, the fact that data

and behavior are separated makes that design fragment harder to understand and to maintain. The fact that in this class all attributes are private, is a good example of how accessor methods can obey the principle of data hiding and still let the class be a pure data holder.
Speaking about Data Class examples, let us revisit a previous example presented in the context of Feature Envy(84) (see Fig. 5.5 on page 86) in which class ClassDiagramLayouter was envying the attributes of ClassDiagramNode. The Feature Envy problem is mainly due to the fact the ClassDiagramNode is a Data Class(88), and thus its behavior and data are not part of the same class. This reveals an often encountered relation between the two aforementioned disharmonies: a Data Class(88) will make the classes that are using it to envy its data; or, the other way around: when a method is affected by Feature Envy(84), it is rather probable that we will find Data Classes among the classes from which that method accesses data.

Refactoring

The basic idea of any refactoring action on a Data Class is to put together in the same class the data and the operations defined on that data, and to provide proper services to the former clients of the public data, instead of the direct access to this data.

- This *data-operation proximity* (see Implementation Rule) can be achieved in most of the cases by analyzing how clients of the Data Class use this data. In this way we can identify some pieces of functionality (behavior) that could be extracted and moved as services to the Data Class. This refactoring action is very much related to what needs to be done when Feature Envy(84) is encountered. In other words, when refactoring a case of Feature Envy(84), this could lead to a positive effect towards repairing a envied Data Class.
- In some other cases, especially if the Data Class is dumb and has only one or a few clients, we could remove the class completely from the system and put the data it contains in those classes (former clients) where the best *data-operation proximity* is achieved.
- If the Data Class is a rather large class with some functionality, but also with many exposed attributes, it is very possible that only a part of the class needs to be cured. In some cases this could mean extracting the disharmonious parts together to a separate class and applying the classical treatment, i.e., trying to extract pieces of functionality from the data clients as services provided by the new class.

5.6 Brain Method

Description Often a method starts out as a "normal" method but then more and more functionality is added to it until it gets out of control, becoming hard to maintain or understand. Brain Methods tend to centralize the functionality of a class, in the same way as a God Class(80) centralizes the functionality of an entire subsystem, or sometimes even a whole system.

Applies To Operations, i.e., methods or standalone functions.

Impact A method should avoid size extremities (Proportion Rule). In the case of Brain Methods the problem concerns overlong methods, which are harder to understand and debug, and practically impossible to reuse. A well-written method should have an adequate complexity which is concordance with the method's purpose (*Implementation Rule*).

Detection The strategy for detecting this design flaw (see Fig. 5.9) is based on the presumed convergence of three simple code smells described by Fowler [FBB+99]:

- *Long methods* – These are undesirable because they affect the understandability and testability of the code. Long methods tend to do more than one piece of functionality, and they are therefore using many temporary variables and parameters, making them more error-prone.
- *Excessive branching* – The intensive use of switch statements (or if–else–if) is in most cases a clear symptom of a non-object-oriented design, in which polymorphism is ignored.[2]
- *Many variables used* – The method uses many local variables but also many instance variables.

The *detection strategy* in detail is:

[2] The excessive use of polymorphism also introduces testability and analyzability problems [Bin99]. Yet, the emphasis in the context of this design flaw is on a very frequent case in which legacy systems migrated from structured to object-oriented programming.

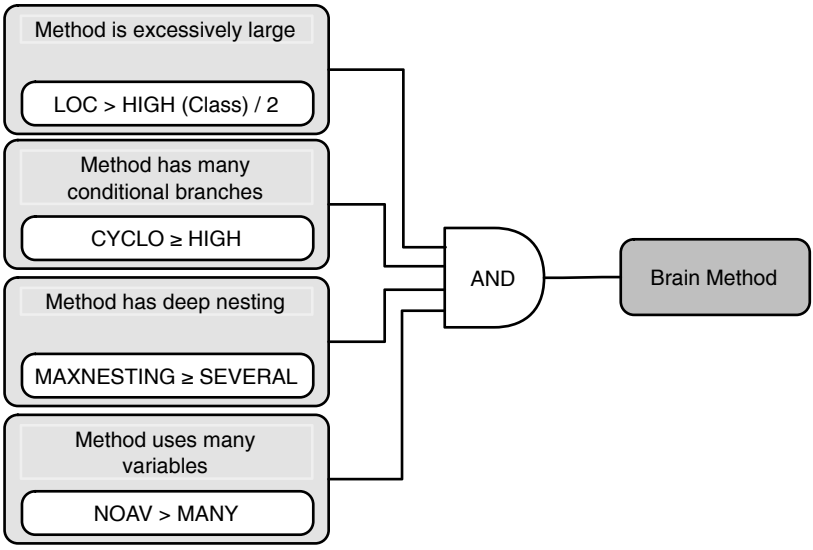


Fig. 5.9. The Brain Method detection strategy.

1. **Method is excessively large.** We are looking for excessively large methods. Based on our practical experience, we used the following heuristic to set the threshold: a method is considered to be excessively large if its LOC count is higher than half of the statistical HIGH threshold for classes (see Table 2.2 for the LOC count of classes) [3].
2. **Method has many conditional branches.** This is computed using the CYCLO (McCabe's Cyclomatic Complexity) metric.
3. **Method has deep nesting level.** This is computed using the MAXNESTING (Maximum Nesting Level) metric i.e., the maximum nesting level of control structures within a method or function.
4. **Method uses many variables.** Method uses more variables than a human can keep in short-term memory. Exceeding this limit always raises the risk of introducing bugs. Notice that all types of variables are counted including local variables, parameters, but also attributes and global variables (in programming languages where this is unfortunately possible). We used NOAV (Number Of Accessed Variables) to compute this.

[3] Only the lines of code in the methods of the class are counted.

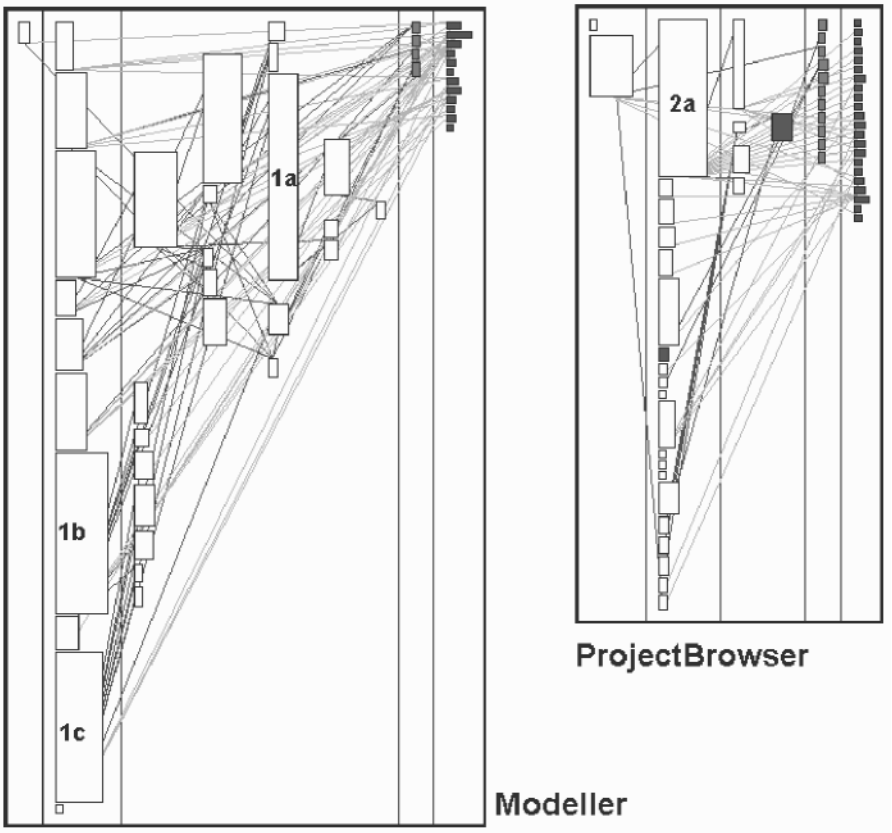


Fig. 5.10. A *Class Blueprint* of Modeller and ProjectBrowser.

Example

Fig. 5.10 shows that Modeller is not a class with an excessive number of methods, but has a certain number of Brain Methods. Some of the methods reach considerable sizes (eight methods are longer than 50 lines of code), the longest one addDocumentationTag (annotated as 1a in the figure) is 150 lines of code and invoked by three other methods, two of which are the second and third longest methods in this class: addOperation (1b, 116 LOC) and addAttribute (1c, 108 LOC).

The *Class Blueprint* reveals other disharmonies in this class: there are 12 attributes in this class, all of them private (which is good), but there are only four accessor methods. Moreover, the attributes are accessed both directly and indirectly (using the accessors), denoting a certain inconsistency or lack of access policy. As we will see

in the chapter on collaboration disharmonies, the class Modeller is also affected by other problems such as Dispersed Coupling(127) and Intensive Coupling(120).

The class ProjectBrowser has a very high ATFD (Access To Foreign Data) value, as it accesses the data of seven other classes (this cannot be seen in the blueprint, since we only display the class itself). As we look closer at the three disharmonious methods of this class we find out that this situation has two different reasons: the less "harmless" one is encountered in the createPanels method (annotated as 2a, 116 LOC) where various UI components are added to an UI panel. There is also a more harmful case, i.e., a violation of Demeter's Law[Lie96] where the programmers build long chains of method calls, most of which are accessor methods. A relevant example is the following code sequence found in the setTitle method:

```
String changeIndicator =
  ProjectManager.getManager().
                 getCurrentProject().
                 getSaveRegistry().
                 hasChanged() ? " *" : "";

ArgoDiagram activeDiagram =
  ProjectManager.getManager().
                 getCurrentProject().
                 getActiveDiagram();
```

The problem with such long invocation chains is that only one of the "links" in the middle has to break (because some method has changed) to make the whole chain break down.

Refactoring

Fowler suggests [FBB+99] that in almost all cases a Brain Method should be split, i.e., that one or more methods (operations) are to be extracted. He also explains how to find the possible "cutting points":

> *[...] whenever we feel the need to comment something, we write a method instead. Such a method contains the code that was commented but is named after the intention of the code rather than how it does it.*

In spite of this simple heuristic, refactoring a Brain Method can be a complex task, which needs a global perspective to solve it. Often we find Brain Methods among the suspects of the Intensive Coupling(120) and Dispersed Coupling(127) design flaws. To properly refactor a Brain

Method we need a complex (interdependent) three-fold analysis, involving all harmony aspects:

1. **Identity harmony.** Implies the already-mentioned aspect of its length which points to a split method refactoring. It may also involve Duplication(102) that implies the extraction of the common part to a method of that class. Additionally, a Brain Method (or a part of it) may exhibit Feature Envy(84). In this case, the refactoring would mean extracting a part of the method or – in some rare cases – moving the method completely to the "data provider".
2. **Collaboration harmony.** As mentioned before, it is often the case that Brain Methods exhibit also Intensive Coupling(120) classification disharmony. This could imply the following refactoring: replace a "cluster" of calls to lightweight methods and the afferent logic with fewer calls to higher-level (more complex) services (see also explanations on Intensive Coupling(120)). This implies extracting a part of the method and moving it to another class.
3. **Classification harmony.** This aspect of harmony might be involved as well if Duplication(102) is detected in the Brain Method. If this is the case, often among the other methods that are the "duplication partners" we find other Brain Methods. Thus, the method can be restructured by factoring out the commonalities in the hierarchy (e.g., apply the Template Method [GHJV95] design pattern).

5.7 Brain Class

Description

This design disharmony is about complex classes that tend to accumulate an excessive amount of intelligence, usually in the form of several methods affected by Brain Method(92).

This recalls the God Class(80) disharmony, because those classes also have the tendency to centralize the intelligence of the system. It looks like the two disharmonies are quite similar. This is partially true, because both refer to complex classes. Yet the two problems are distinct.

The fingerprint of a God Class is not just its complexity, but the fact that the class relies for part of its behavior on encapsulation breaking, as it directly accesses many attributes from other classes.

On the other hand, the Brain Class *detection strategy* is trying to complement the God Class strategy by catching those excessively complex classes that are not detected as God Classes either because they do not abusively access data of "satellite" classes, or because they are a little more cohesive.

Applies To

Classes which are not a God Class(80) and contain at least one method affected by Brain Method(92).

Impact

See impact of the God Class(80) disharmony.

Detection

The detection rule can be assumed as follows (see Fig. 5.11). A class is a Brain Class if it has at least a *few* methods affected by Brain Method(92), if it is very large (in terms of LOC), non-cohesive and very complex. If the class is a "monster" in terms of both size (LOC) and functional complexity (WMC) then the class is considered to be a Brain Class even if it has only one Brain Method(92).[4] The *detection strategy* in detail is:

1. **Class contains more than one Brain Method(92) and is very large.** A class is very large if the total number of lines of code from methods of the class is *very high* (see Fig. 5.12).

[4] Looking carefully at the detection rule and comparing it to the one for God Class(80), you will notice that nothing hinders a God Class from also being detected as a Brain Class. For simplification, we exclude a priori classes classified as God Classes.

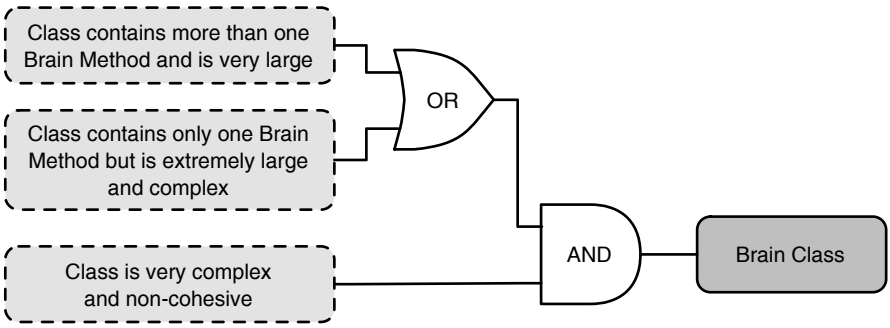


Fig. 5.11. Detection strategy for Brain Class.

2. **Class contains only one Brain Method(92) but is extremely large and complex.** This term covers the above case of a "monster" class in both size and complexity, and which is not captured by the previous term due to the fact that the class has only one *Brain Method*. In other words, this is the case of a class where most methods tend to be excessively large and complex, even if they are not Brain Methods. Compared to the previous term, a special condition (WMC) was added on the complexity of the class. This condition overrides the "normal" filtering condition for WMC, as defined in the third term.
3. **Class is very complex and non-cohesive.** This last term sets a requirement on the increased complexity and low cohesion that characterize mainly all classes with identity disharmonies. This pair of filtering conditions is similar to the one found in the detection rule for God Class(80); in fact there is only one difference: the threshold for the cohesion metrics is more permissive than in the other *detection strategy*, as there the very low cohesion is a more significant characteristic than here.

Example

In Fig. 5.13 we see that the class ParserDisplay not only is visually deformed, but also plays strange tricks in terms of inheritance. As for the visual deformation, this class implements some very large methods, the largest one (the tallest method box) with 576 lines of code (this method is also the largest in the entire system) and another five methods longer than 100 lines. In total 13 methods are longer than 50 lines. Moreover, there is a large amount of intra-method dupli-

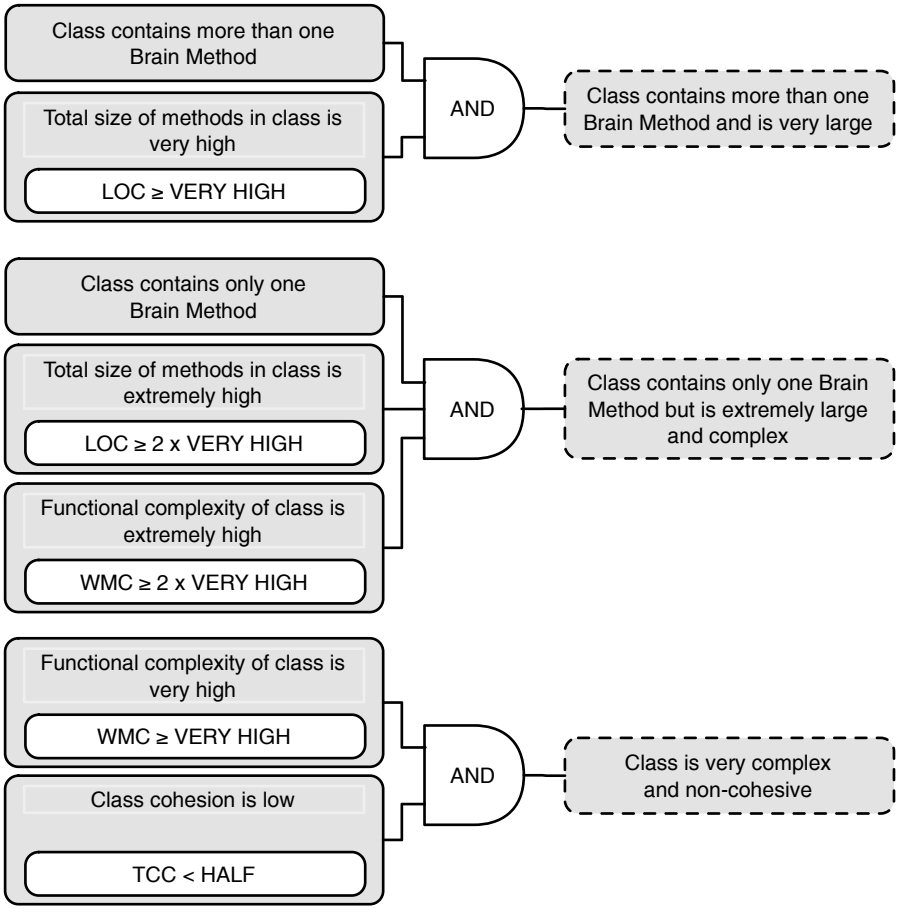


Fig. 5.12. Main components of the Brain Class *detection strategy*.

cation: for example, in the constructor of this class, which contains seven code blocks containing duplication, there are 89 lines of code in total, whereas the constructor has 132 lines in total. Another particular aspect of this class is its inheritance relationship with its superclass, whose discussion we postpone to the section on classification harmony.

The class FigClass is severely affected by the Brain Method(92) disharmony. This class has another problem: code duplication. Eleven of its methods are affected by Duplication(102), nine (!) of which are involved in duplication with three of the sibling classes, especially

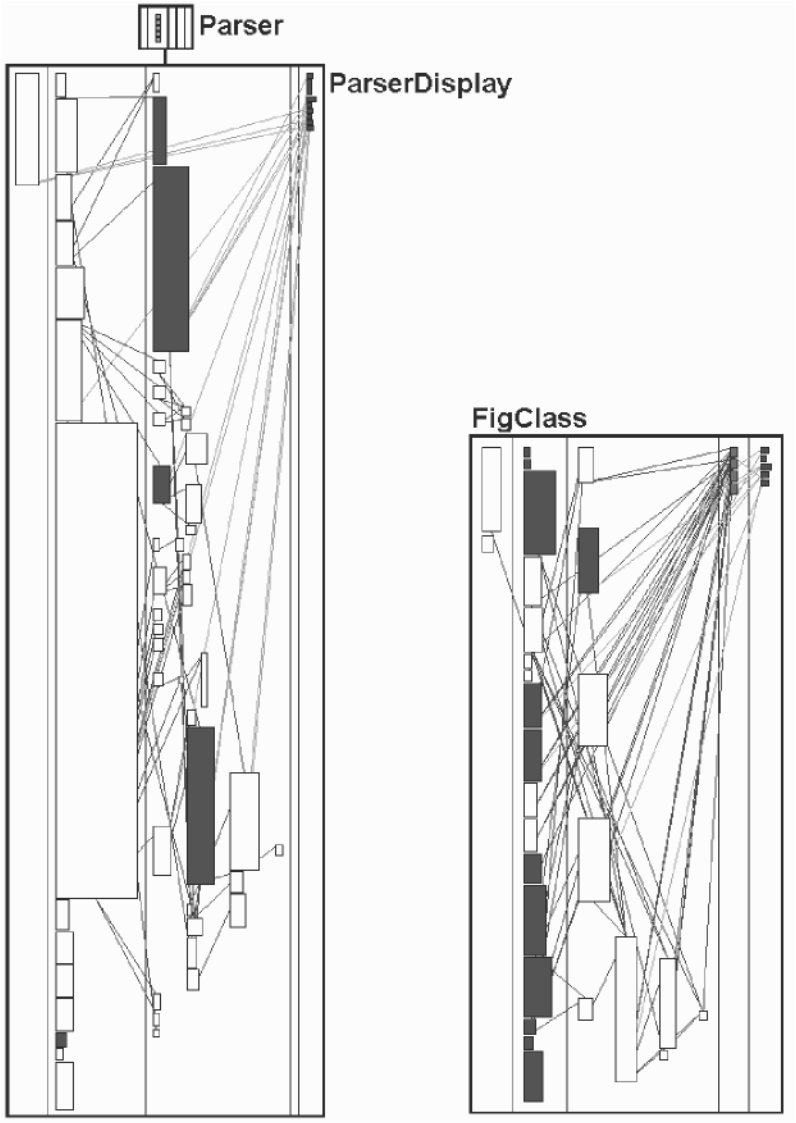


Fig. 5.13. A *Class Blueprint* of ParserDisplay with its completely abstract superclass Parser and a *Class Blueprint* of FigClass.

with FigInterface and FigUseCase. What is even more interesting is that among the nine methods with Duplication(102) we found all three Brain Methods of the class, all of them with significant amounts of duplication. Our conclusion is that if the duplication "plague" were

removed this class would become much lighter, and less problematic. As you can see, in order to restore one aspect of harmony the other aspects must be considered as well. In this concrete example, we would not have found the cause of the Brain Method(92) problem if we had not looked at the duplication within the hierarchy.

Refactoring

The primary characteristic of a Brain Class is the fact that it contains Brain Method. Therefore the main refactoring actions for these classes must be directed towards curing the Brain Method(92) disharmonies. Additionally, in our approach classes affected either by Brain Class(97) or God Class(80) represent the starting point in the detection and correction of identity disharmonies (see Sect. 5.9).

Apart from that, in our experience, there are at least three types of Brain Class, each of them requiring a different treatment:

1. The methods suffering from Brain Method(92) contained in the class are semantically related (oftentimes overloaded methods), and contain a significant amount of duplicated code. Factoring out the commonalities from these methods in form of one or more private or protected methods, while making the initial methods provide only the slight differences would significantly reduce the complexity of the class.
2. A possible type of Brain Method appears when a class is conceived in a procedural programming style. Consequently, the class is mainly used as a grouping mechanism for a collection of somehow related methods that provide some useful algorithms. In this case the class is non-cohesive. Refactoring such a class requires to split it into two or more cohesive classes. Yet, performing such a refactoring requires a substantial amount contextual information (e.g., which class(es) use(s) which parts of the initial class, where is stored the data on which each Brain Method operates on etc.)
3. There are cases where a Brain Class proves to be rather harmless. In several case studies we encountered cases where an excessively complex class was a matured utility class, usually not very much related to the business domain of the application (e.g., a class modelling a Lisp interpreter in a 3-D graphics framework). If, additionally, the maintainers of the system or the analysis of the system's history [RDGM04] show that no maintenance problems have been raised by that class then it makes no sense to start a costly effort of refactoring that class just for the sake of getting better metric values for the system.

5.8 Significant Duplication

Description The detection of code duplication plays an essential role in the assessment and improvement of a design. But detected clones might not be relevant if they are too small or if they are analyzed in isolation. In this context, the goal of this *detection strategy* is to capture those portions of code that contain a *significant* amount of duplication. What does significant mean? In our view a case of duplication is considered significant if:

- It is the largest possible chain of duplication that can be formed in that portion of code, by uniting all islands of *exact clones* that are close enough to each other.
- It is large enough.

Applies To Pairs of operations.

Impact Code duplication harms the uniqueness of entities within a system. For example, a class that offers a certain functionality should be solely responsible for that functionality. If duplication appears, it becomes much harder to locate errors because the assumption "only class X implements this, therefore the error can be found there" does not hold anymore. Thus, the presence of code duplication has (at least) a double negative impact on the quality of a system: (1) the bloating of the system and (2) the co-evolution of clones (the clones do not all evolve the same way) which also implies the cloning of errors.

Detection In practice, duplications are rarely the result of pure copy–paste actions, but rather of copy–paste–adapt "mutations". These slight modifications tend to scatter a monolithic copied block into small fragments of duplicated code. The smaller such a fragment is, the lower the refactoring potential, since the analysis becomes harder, and the granted importance is decreased, too. So, for example, imagine we found two operations that have five identical lines, followed by one line that is different, which is followed by another four identical lines. Did we find two clones (of five and four lines) or one single clone

spread over ten lines (5 + 1 + 4 lines)? In such cases, it is almost always better to choose the second option.

Thus, there are two cases of duplication: the *copy–paste* case and the *copy–paste–adapt* case. This *detection strategy* captures both cases. The first term deals with the case of a brute-force duplication which is significantly large. The second term tackles the case of duplication with slight adaptations, assuming that the largest possible chain of duplication is considered. In both case the key element is the size of the duplication.

In order to introduce the *Significant Duplication detection strategy* (see Fig. 5.14), we need first to present three low-level duplication metrics:

- **Size of Exact Clone (SEC).** An *exact clone* is a group of consecutive line-pairs that are detected as duplicated. Consequently, the *Size of Exact Clone* metric measures the size of a clone in terms of lines of code. The size of a clone is relevant, because in most of the cases our interest in a piece of duplicated code is proportional to its size.
- **Line Bias (LB).** When comparing two pieces of code (e.g., two files or two functions) we usually find more than one *exact clone*. In this context, *Line Bias* is the distance between two consecutive *exact clones*, i.e., the number of non-matching lines of code between two *exact clones*. The LB value may allow us to decide if two *exact clones* belong to the same cluster of duplicated lines (e.g., the gap between the two *exact clones* could be a modified portion of code within a duplicated block of code).
- **Size of Duplication Chain (SDC).** To improve the code we need to see more than just a pile of small duplication chunks. We want to see the big picture, i.e., to cluster the chunks of duplication into a more meaningful block of duplication. This is what we call a *duplication chain*. Thus, a *duplication chain* is composed of a number of smaller islands of *exact clones* that are close enough pairwise to be considered as belonging together, i.e., their LB value is less than a given threshold.

Now, with these metrics in mind we can revisit the example mentioned earlier in this section, with two functions having two *exact clones*. In terms of the low-level duplication metrics introduced in this section, we can now say that the first clone has a SEC value of 5, while the second one has a SEC value of 4. Between the two clones

there is a gap of one line; thus, the LB value is 1. Consequently the SDC metric has a value of 10 lines (5 + 1 + 4 lines).

Based on these low-level metrics we can now introduce the heuristics for this *detection strategy*:

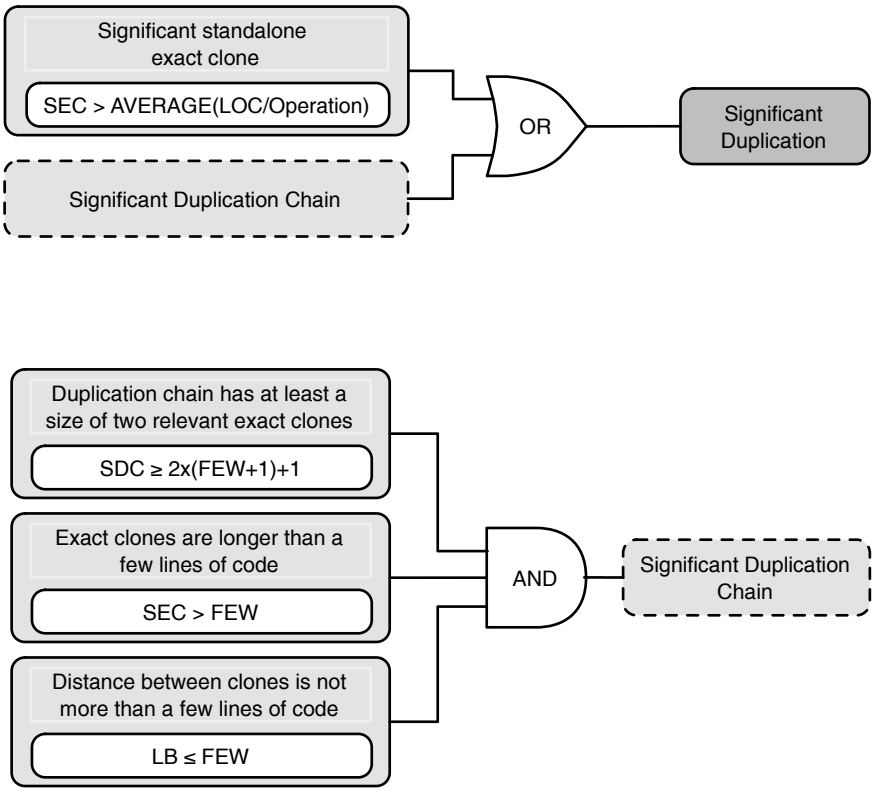


Fig. 5.14. The Significant Duplication detection strategy.

1. **Significant Standalone Exact Clone.** This case captures the case of a contiguous, isolated block of duplication, i.e., a single *exact clone* that has no other clones in its neighborhood. Thus, the only thing that counts is the size of the *exact clone*. We consider a standalone clone to be large enough if it is at least as large as the statistical average size of an operation.
2. **Significant Duplication Chain.** We stated earlier that a block of duplicated code is significant only if it is the largest one that could

be built in a particular area of the two pieces of code that are compared. In other words, we try to build the largest chain of relevant *exact clones* that are not too far from each other. As you might notice, the previous phrase is fuzzy if not associated with a measurement. Based on the low-level metrics defined earlier, we eliminate the "fuzziness" and make the identification of these clusters reproducible. This term is composed of three metrics (see Fig. 5.14), each one carrying out a particular role:

a) **Duplication chain has at least a size of two relevant exact clones.** This threshold is an indirect one, meaning that the duplication chain has at least the total size of two significant exact clones separated by a gap of minimal distance. The term $2 \times (FEW + 1)$ is based on the condition that each of the (minimum) two fragments involves *more than a few duplicated lines*. Because the minimal distance (i.e., the smallest LB value) is *one* line of code, we add to the first threshold term one more line. It ensures that the total length of the duplication chain is large enough to qualify it as significant.

b) **Exact clones are longer than a few lines of code.** This makes sure that the chain is not composed only of irrelevant "duplication crumbs", i.e., that each fragment of the duplication chain is not very small.

c) **Distance between clones is not more than a few lines of code.** This quantifies the "neighborhood" aspect as it ensures that the pieces of the chains are not too far from each other to be considered as belonging to the same duplication chain. In other words, the threshold for the LB metric is used as a stop condition in the process of looking for further neighbor clones.

Example

Looking at the *ArgoUML* case study just shows that code duplication is one of the plagues that are omnipresent; but this can be now quantified. In the case of *ArgoUML* , we checked for *Significant Duplication* and we found that 239 classes (17% of all the classes) are affected by it. Summing the SDC duplication metric at the system level, we end up with more than 10,000 duplication lines![5]

Usually duplication is a design disharmony that often appears in conjunction with other disharmonies. Therefore, we believe that it

[5] Note that one code line may be involved in more than one duplication chain, and thus it is multiply counted; still, the number of lines of code involved in duplication is impressive.

does not make sense to discuss just a single concrete example of duplication. So, the aspect of duplication will occur over and over again, as we discuss in an integrated manner various design problems that we encountered in *ArgoUML* .

Refactoring

The essence of a refactoring that intends to eliminate duplication is based on Beck's *Once and Only Once Rule* [Bec97]:

> *By eliminating the duplicates, you ensure that the code says everything once and only once, which is the essence of good design.*

Thus, it is clear that we have to put all "instances" of a duplicated portion of code into one single location. But what is the proper location? To be able to answer this question we obviously need more information about the *context* of the duplicated entities.

The first problem is that by detecting only exact clones we usually end up with lots of small clones (duplication crumbs) which are irrelevant in themselves; yet, ignoring them would be a mistake as they could be in fact part of a large duplication chain, as a result of an extensive *copy–paste–adapt* process. The *Significant Duplication* strategy helps us in solving this first headache and keeps only significant portions of code affected by duplication.

This brings us to the second headache: How to refactor the code so that duplication is removed? Are all significant duplication blocks the same? Can we apply the same treatment to all? Especially when speaking about object-oriented design, the answer to these questions is definitely (and obviously): No! This is why we identify three different contexts in which duplication appears.[6]

Case I: Duplication Within the Same Class

In this case the two methods involved in a (significant) duplication block belong to the same class. This is probably the easiest refactoring: all that needs to be done is to extract the commonality in the form of a new method and call the new method from both places (see Fig. 5.15).

Case II: Duplication Within the Same Hierarchy

In this second case the two methods that are part of a (significant) duplication block are not part of the same class, but belong to the same

[6] Note that we speak here exclusively about duplication of functional code, i.e., duplication that appears in the bodies of functions and methods.

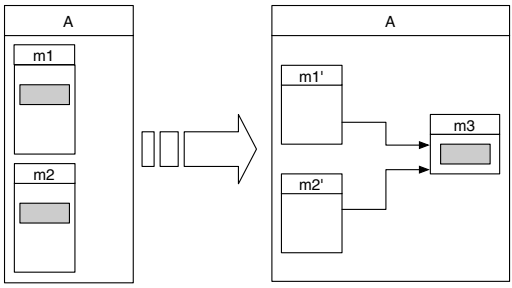


Fig. 5.15. Recovering from Duplication within the same class.

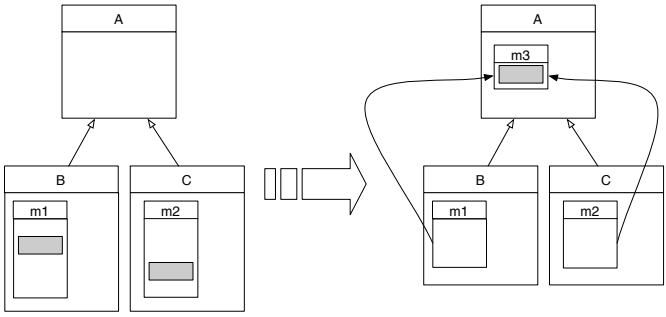


Fig. 5.16. Recovering from Duplication within sibling classes.

class hierarchy, which means that either they are in an ancestor–descendant relation or they share a common base class. This type of duplication can be eliminated in one of the following ways, depending on the inheritance relation between the methods involved in the duplication (see Fig. 5.16):

- *Siblings duplication.* If duplication appears in two methods that have a common ancestor, then the commonality is extracted in the form of a new method placed into the common ancestor.
- *Parent–child duplication.* This case of duplication is a strange one, because the two classes are in a direct relation. Thus, any commonality could have been placed in the base class. The refactoring consists of extracting any common code and placing it in the parent class, where it logically belongs.

A special case is the one where the duplication between two inheritance-related methods is fragmented, i.e., the code is similar but not

identical. In this case you would probably be able to apply the *Template Method* design pattern [GHJV95], as this would help separate the common code (which goes into the closest ancestor class) from the fragments that are different (which will become the hooks from the pattern mentioned above).

Case III: Duplication Within Unrelated Classes

In this third case the two operations that share a duplicated block are neither part of the same class, nor of the same hierarchy; either the two operations are part of two independent classes (in the sense of classification) or they are (one of them or both) global functions.

If you find duplicated code in methods belonging to unrelated classes, there are three major options on where to place the common code, extracted from the two (or more) classes:

- *One hosts, one calls.* In this case we notice that the code belongs to one of the protagonist classes. Thus, it will host the common code, in the form a method, while the other class will invoke that method. This usually applies when the portions of duplication are not very large and especially not encountered in many methods.
  If the duplication between two classes affects many methods, then we probably miss an abstraction, i.e., a third class. Thus, we define the new class and place the duplicated code there. Now, the question is how to relate the two former classes with this third one? The answer depends on the context, boiling down to two options: *association* and *inheritance*.
- *Third class hosts, both inherit.* If we find that the two classes are conceptually related, then they probably miss a common base class. Consequently the third class becomes the base class of the two.
  Good examples for this case are the classes FigNodeModelElement and FigEdgeModelElement which indeed miss a common base class.
- *Third class hosts, both call.* If the two unrelated classes involved in duplication are not conceptually related we need to introduce an association from the two classes to the third one and call from both classes the method that now hosts the formerly duplicated code.

5.9 Recovering from Identity Disharmonies

Where to Start

In practice we do not have enough time to analyze each suspect class or method reported by the *detection strategies*. Therefore, a pragmatic question pops up: How do we find the most important identity harmony offenders? We used the following criteria in selecting the classes that especially need attention with respect to identity harmony:

- Classes that contain a higher number of disharmonious methods have priority.
- Classes in which more than one identity disharmony appears have priority.
- Classes that are affected by other disharmonies (i.e., collaboration or classification disharmonies) go first in order to reveal relations to other aspects of harmony.

This can be done in two steps (see Fig. 5.17):

1. Start with the "intelligence magnets", i.e., with those classes that tend to accumulate much more functionality than an abstraction should normally have. In terms of the *detection strategies* presented so far, this means to make a blacklist containing all classes affected by the God Class(80) or by the Brain Class(97) disharmony.
2. For each of the classes in the blacklist built in *Step 1* find the *disharmonious methods*. A method is considered disharmonious if at least one of the following is true:
   - it is a Brain Method(92);
   - it contains duplicated code;
   - it accesses attributes from other classes, either directly or by means of accessor methods.

We mainly use the following quantification means:

Count disharmonious methods. To both assess and cure such a disharmonious class we need first to examine how much the identity problems have spread among the methods of that class. Therefore, we have to count how many *methods* we can identify as having identity problems. The more disharmonious methods a class has, the more critical its identity is. When do we consider a method as being disharmonious? We do so if at least one of the following conditions holds:

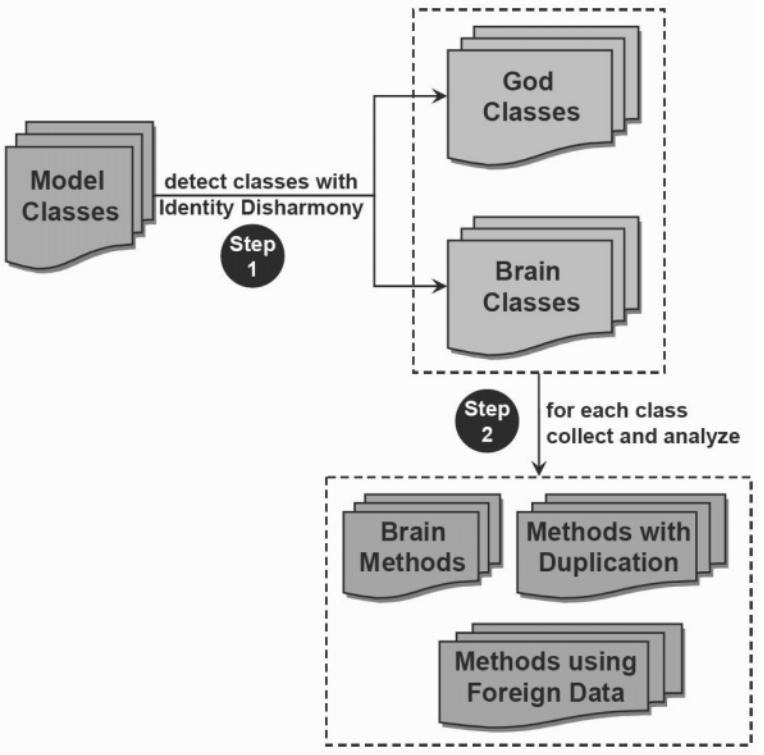


Fig. 5.17. Overview of the assessment process related to Identity Harmony.

1. It contains duplicated code in common with methods of the same class.
2. It is a Brain Method(92).
3. It accesses directly attributes of other classes.

Number of Methods (NOM). This metric gives us information about the functional size of the class. If we correlate the number of disharmonious methods with NOM, we can also determine what percentage of the class is affected by these identity problems.

Number of methods detected as Brain Method(92). We use this number to see how many of the disharmonious methods are detected as being a Brain Method(92).

Duplicated LOC. This metric tells us for each class the amount of intra-class duplication, which refers to source code duplicated within a class, i.e., among the methods defined in the same class.

This information helps us also to get a better understanding of what the problem is with the disharmonious methods.

Access To Foreign Data (ATFD). This metric is included because it quantifies one of the key disharmonies of an identity distortion, i.e., the brute usage of attributes from other classes. As you may notice, this is again one of the reasons that qualify a method as being *disharmonious*.

How to Start

How should you start when you want to improve the identity harmony of your system's classes? Assuming that for a class in the blacklist we have gathered its disharmonious methods, then in order to recover from identity design disharmonies we have to follow the roadmap described in Fig. 5.18, and explained briefly below.

- *Action 1: Remove duplication.* The first thing to be done is to check if a method contains portions of Duplication(102) and remove that duplication in conformity with the indications provided in Sect. 5.8. Because we analyze the class from the perspective of *identity harmony* we concentrate on removing the intra-class duplication first. If a lot of duplication is found, the result of this step can have a significant positive impact on the class, especially on its Brain Methods.

- *Action 2: Remove temporary fields.* Among the bad smells in code described in [FBB+99] we find one called *Temporary Field*. This is an attribute of a class used only in a single method; in such cases the attribute should have been defined as a local variable. Obviously, detecting such situations can be done by checking in the class who other than the inspected method uses a particular attribute. If no one else does, then we need to remove the temporary field and replace it with a local variable. Why do we care? Remember that for both Brain Class(97) and God Class(80) one of the "fingerprints" is a low cohesion. One of the causes of low cohesion could also be a bunch of such *temporary fields*, which do not really characterize the abstraction modelled by the class, and thus hamper the understanding of the class.

- *Action 3: Improve data-behavior locality.* If in our inspection process we reach a *foreign data user*, i.e., a method that accesses

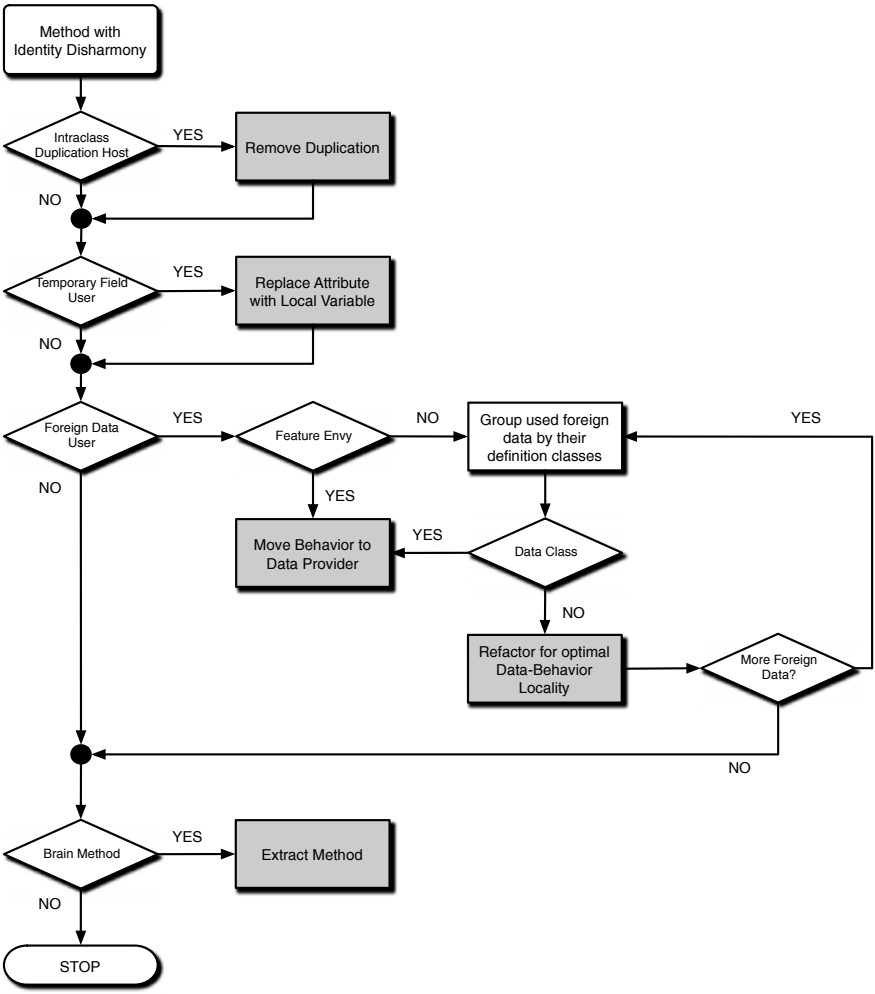


Fig. 5.18. How to address Identity Disharmonies in methods

attributes of other classes, then we have to refactor it so that we reach an optimal *data-behavior locality*. A *foreign data user* has one characteristic: the value for the ATFD metric is at least one. In a simplistic way we can say that refactoring in this case requires one of these two actions:

– *Extract a part of the method and move it to the definition class of the accessed attribute.* The "ideal" case is when the method is affected by Feature Envy(84) and the class that provides the at-

tributes is a Data Class(88). In this case the method was simply misplaced, and needs to be moved to the Data Class(88). But life is rarely that easy, so the situations that you will probably encounter are more "gray" than "black and white". In most cases only a fragment of the method needs to be extracted and moved to another place. This entire discussion is beyond the scope of this book, but here is a rule of thumb that we often use: if the class that provides the accessed attributes is "lightweight" (i.e., Data Class(88) or close to it) try to extract fragments of functionality from the "heavyweight" class and move them to the "lightweight" one.

  - *Move the attribute from its definition class to the class where the user method belongs.* This is very rarely the case, especially in the context of Brain Class(97) and God Class(80). It applies only for cases where the attribute belongs to a small class that has no serious reason to live, and which will be eventually removed from the system.

- *Action 4: Refactor Brain Method.* If you reached this step while inspecting a method that was initially reported as a Brain Method(92), first look if this is still the case after proceeding with *Step 1* and *Step 3*. Sometimes, removing duplication and refactoring a method for better data-behavior locality solves the case of the Brain Method(92). If the problem is not solved, revisit Sect. 5.6 where we discussed the main refactoring cases for a Brain Method(92).

6

Collaboration Disharmonies

6.1 Collaboration Harmony Rule

Collaboration disharmonies are design flaws that affect several entities at once in terms of the way they collaborate to perform a specific functionality.

The principle of low coupling is advocated 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 of classes should be minimized. 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. Responsibility-driven approaches stress the fact that classes should implement well identified responsibilities often by delegating work to others and collaborate with a clearly identified and limited set of collaborators [WBM03].

A harmonious collaboration is one that maintains a balance between the inherent need for communication among the entities (i.e., methods and classes) of a system and the demand to keep this coupling to a minimum. The collaboration harmony rule is:

Collaborations should be only in terms of method invocations and have a limited extent, intensity and dispersion

---

Collaboration Rule

> *Collaborations should be only in terms of method invocations and have a limited extent, intensity and dispersion*

Rationale

The idea behind this rule is summarized by Lorenz and Kidd [1]:

> *You want to leverage the services of other classes, but you want to have services at the right level, so that you want to know only about a limited number of objects and their services. [...] If you had to interact with all the indirectly related objects, we'd have a tangled web of interdependencies and maintenance would be a nightmare [LK94].*

The rule refers both to outgoing and incoming dependencies. *Excessive* outgoing dependencies are undesirable because the more one uses the others, the more *vulnerable* (to changes and malfunction) one becomes. *Excessive* incoming dependencies are also undesirable because the more one is used by the others, the more responsible and thus *immutable* (i.e., rigid, stable, less evolvable) one becomes. At the same time, note that excessive incoming dependencies may also be a good sign of design and functionality reuse, with one condition: the used interfaces are *stable*. An example is given by class libraries implementing collections or common infrastructure. Additionally, it is important to take into account the important role of stable interfaces to support changes. Interfaces play an important role in shielding clients from specific implementation concerns hence reducing the impact of changes.

Practical Consequences

- **Limit collaboration intensity** – *Operations should collaborate (mainly unidirectional) with a limited number of services provided by other classes.*

[1] The rule is also very much related to Pelrines's Object Manifesto which states: *Be private: do not let anybody touch your private data. Be lazy: Delegate as much as possible*

- **Limit collaboration extent** – *Operations (and consequently their classes) should collaborate*[2] *with operations from a limited number of other classes.*

  This is a restatement of "A harmonious system must have services defined at the proper level, so that you need to collaborate directly only with a limited number of other abstractions" [LK94].

- **Limit collaboration dispersion** – *The collaborators (i.e., invoked and/or invoking operations) of an operation should have a limited dispersion within the system. Thus, one should try to make an entity collaborate closely only with a selected set of entities, with a preference for entities (in decreasing order) located in the (0) same abstraction; the (1) same hierarchy; the (2) same package (or subsystem).*

[2] *The term Collaborate* refers both to the active (i.e., call another operation) and to the passive (i.e., be called (invoked) by another operation) aspects.

6.2 Overview of Collaboration Disharmonies

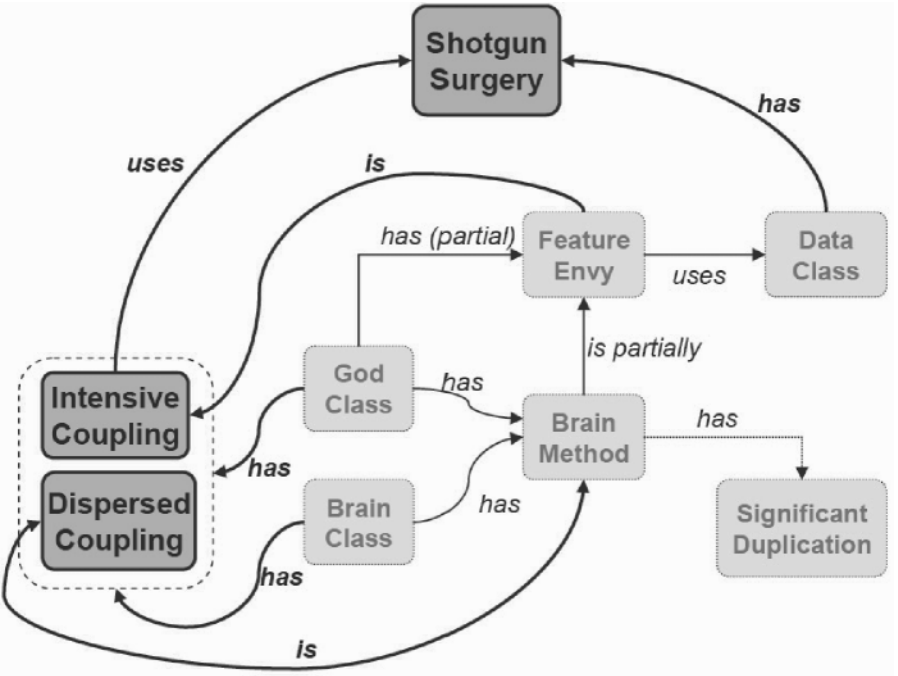


Fig. 6.1. Correlation web of collaboration disharmonies.

The *Collaboration Rule* shows that, especially concerning outgoing coupling, the problem is an *excessive* number of operations which are called from the *disharmonious* operation. A second important aspect is the distribution (dispersion) of these called operations on classes.

Considering the practical consequences above, we can say that an operation is disharmonious in terms of collaboration if it has too many invocations of many other methods.

We capture these collaboration disharmonies using two *detection strategies*, namely Intensive Coupling(120) and Dispersed Coupling(127). While the former captures the case where the method intensively uses a reduced number of classes (invoking lot of method of a particular class), the latter deals with the situation where the dependencies of the disharmonious method are very much dispersed among many classes (invoking methods from too many classes).

In a collaboration, not only the server methods can be disharmonious, but also the client code. Fowler [FBB+99] mentions the case when a small change in a part of a system causes lots of changes to many classes, dispersed all over the rest of the system. They call this bad smell Shotgun Surgery. Inspired by this we captured the disharmony in which a method is excessively invoked by many methods located in many classes (Fig. 6.1), and as a tribute to our inspiration source we called it Shotgun Surgery(133).

Fowler's Shotgun Surgery smell can also take the form of a piece of code which is replicated over and over again in various methods, belonging to various classes which might otherwise not look coupled to each other. For example, when a class is a Data Class(88), its clients often duplicate functionality that would be normally be under the responsibility of that class.Thus, for such cases the Duplication(102) disharmony can also be considered a collaboration disharmony.

6.3 Intensive Coupling

Description One of the frequent cases of excessive coupling that can be improved is when a method is tied to many other operations in the system, whereby these provider operations are dispersed only into one or a few classes (see Fig. 6.2). In other words, this is the case where the communication between the client method and (at least one of) its provider classes is excessively verbose. Therefore, we named this design disharmony Intensive Coupling.

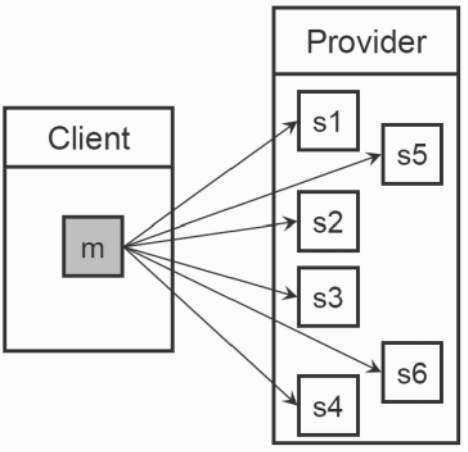


Fig. 6.2. Illustration of Intensive Coupling

Applies To Operations i.e., methods or standalone functions.

Impact An operation which is intensively coupled with methods from a handful of classes binds it strongly to those classes. Oftentimes, Intensive Coupling points to a more subtle problem i.e., the classes providing the many methods invoked by the Shotgun Surgery method do not provide a service at the abstraction level required by the client method. Consequently, understanding the relation between the two sides (i.e., the client method and the classes providing services) becomes more difficult.

Detection

The *detection strategy* is based on two main conditions that must be fulfilled simultaneously: the function invokes many methods and the invoked methods are not very much dispersed into many classes (Fig. 6.3).

Additionally, based on our practical experience, we impose a minimal complexity condition on the function, to avoid the case of configuration operations (e.g., initializers, or UI configuring methods) that call many other methods. These configuration operations reveal a less harmful (and hardly avoidable) form of coupling because the dependencies can be much easily traced and solved.

The *detection strategy* is composed of the following heuristics (see Fig. 6.3):

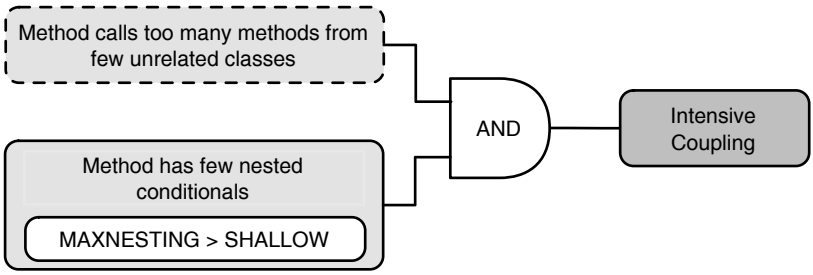


Fig. 6.3. Intensive Coupling detection strategy.

1. **Operation calls too many methods from a few unrelated classes.** The basic condition for a method or function to be considered as having an Intensive Coupling is to call many methods belonging to a few classes (Fig. 6.4). By "unrelated classes" we mean that the provider classes are belonging to the the same class hierarchy as the definition class of the invoking method. We distinguish two cases:
   a) Sometimes a function invokes many other methods (more than our memory capacity) from different classes. Usually among the provider classes there are two or three from which several methods are invoked.
   b) The other case is when the number of invoked methods does not exceed our short-term memory capacity, but all the invoked methods belong to only one or two classes. Thus, the number of methods invoked from the same provider class is high.

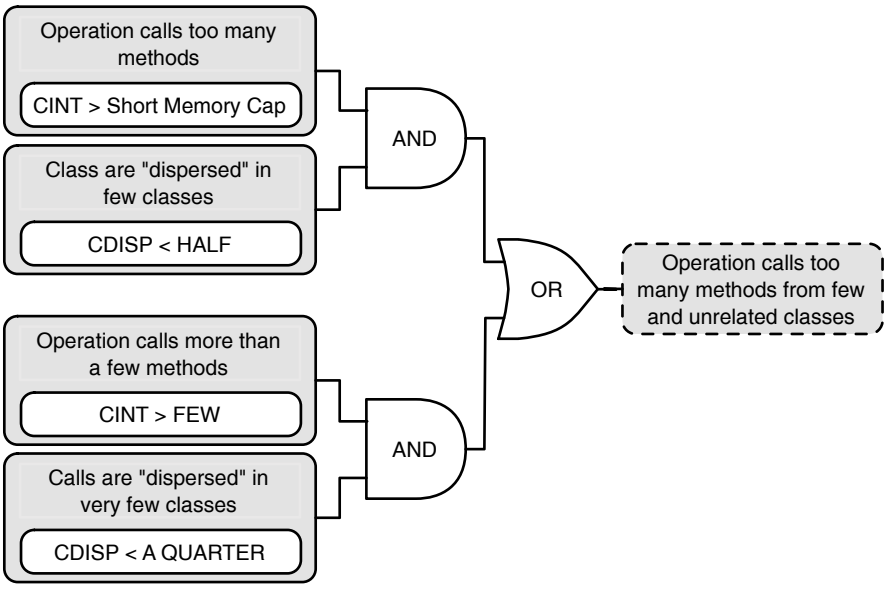


Fig. 6.4. In Intensive Coupling operation calls too many methods from a few unrelated classes

Therefore, we have two branches: one for detecting intensive couplings which are concentrated in one or two classes, and another one dedicated to the more general case when the dispersion ratio of the invoked methods is below 50%.

The used heuristics in the first case are:

a) **Operation calls too many methods.** Too many refers to a number greater than the number of items that can be memorized by the short-term memory. If the caller operation is a method, than only those provider methods are counted that are outside the scope of the caller's definition class.
b) **Calls are dispersed in a few classes.** The methods invoked by a client operation have a low grade of dispersion, i.e., the provider methods belong to a few classes. The threshold tells us that in average more than two methods are invoked from the same provider class.

The used heuristics in the second case are:

a) **Operation calls more than a few methods.**

   b) **Calls are dispersed in very few classes.** The called methods have a very low grade of dispersion, i.e., the threshold tells us that in average more than two methods are called from the same provider class.

2. **Operation has nested conditionals.** A function that calls many methods, but is flat – in terms of the nesting level of its statements – is less complex and from our experience this coupling cases prove to be often less relevant. In many cases such methods are initializers or configuration functions that are less interesting for both understanding and improving the quality of a design. Therefore, as mentioned earlier, we set this condition so that the calling function should have a non-trivial nesting level.

Example

In Fig. 6.5 we see that ClassDiagramLayouter is intensively coupled with a few classes, especially with ClassDiagramNode. The blue edges represent invocations between the methods in the classes. The red nodes represent non-model classes, i.e., Java library classes.

The classes have been laid out according to the invocation sequence: above ClassDiagramLayouter we place all classes that use it, while below it are all classes whose methods get used, i.e., invoked by its methods.

In Fig. 6.6 we see that ClassDiagramLayouter is coupled to ClassDiagramNode mainly because of four large methods, two of which have previously been detected as a Brain Method(92): (1) layout, (2) weightAndPlaceClasses (3) rankPackagesAndMoveClassesBelow and (4) layoutPackages.

In more detail, the method weightAndPlaceClasses invokes 11 methods of the class ClassDiagramNode which by looking at its *Class Blueprint* seems to be a mere data holder without complex functionality. The same goes for the method layout which uses 6 methods of ClassDiagramNode. It looks as, after a few iterations, ClassDiagramLayouter could eventually become a God Class(80).

A strongly suggested refactoring in this case is splitting those methods, since they do several things at once, as their names suggest. For example, weightAndPlaceClasses could be split into a method that weighs and another one that places the classes.

The prediction about ClassDiagramLayouter eventually becoming a God Class(80) or at least a complex class is supported by the fact that so far as the other classes in these figures are concerned, ClassDiagramLayouter uses only small parts of them. This does not re-

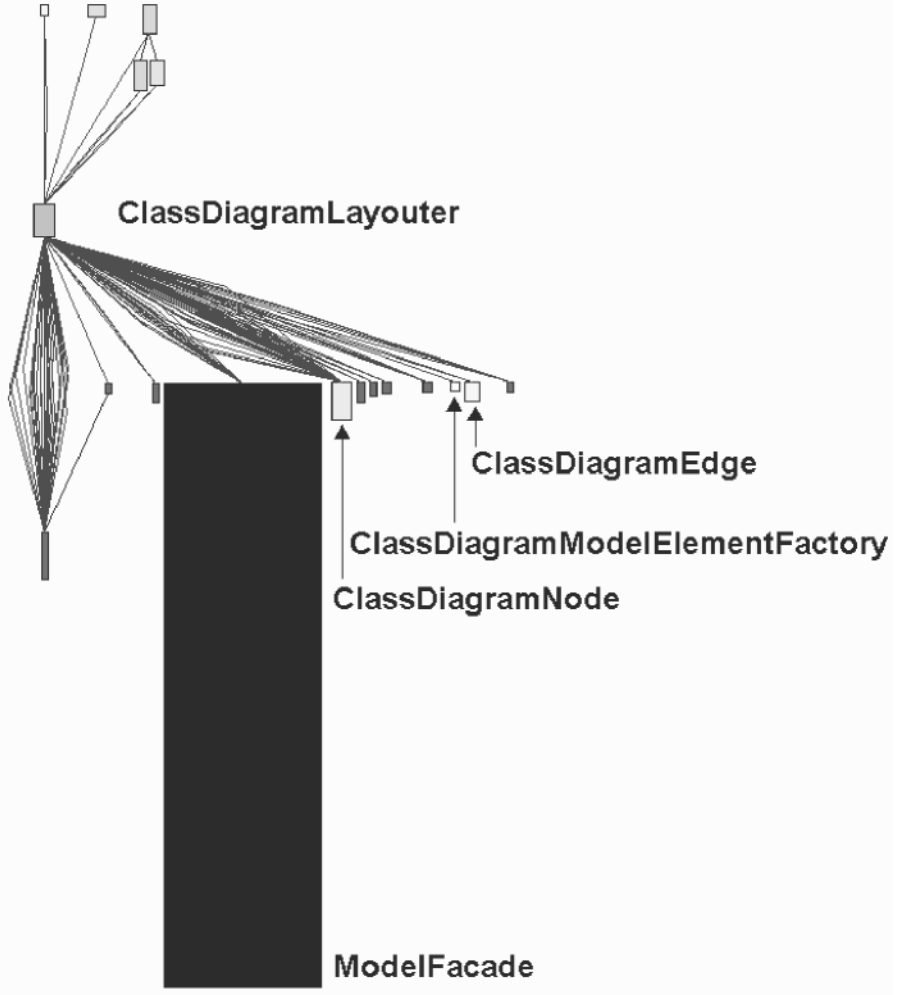


Fig. 6.5. The class ClassDiagramLayouter is intensively coupled with a few classes, especially ClassDiagramNode. The red classes are non-model classes, i.e., belong to the Java library. The classes have been laid out according to the invocation sequence: above ClassDiagramLayouter we place all classes that use it, while below it are all classes whose methods get used, i.e., invoked by its methods.

ally represent a problem, although some of the coupling relationships seem to be very weak and probably do not require much work to be cut off and decrease the couplings of ClassDiagramLayouter.

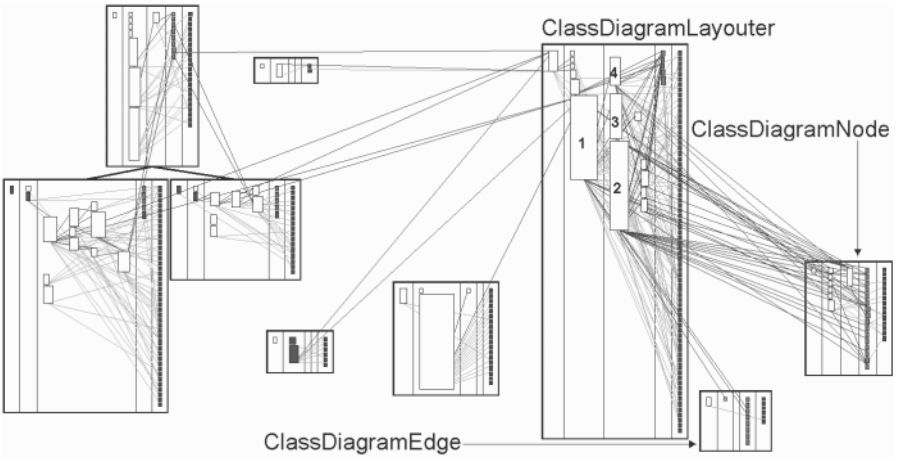


Fig. 6.6. The class ClassDiagramLayouter is intensively coupling with a few classes, especially ClassDiagramNode.

Refactoring

In the case of an operation with Intensive Coupling the intensity of coupling is high, while the dispersion is low. This guarantees that we will find one or more *clusters of methods* invoked from the same (provider) class. Therefore, a first refactoring action is to *try* to define a new (more complex) service in the provider class and replace the multiple calls with a single call to the newly defined method (see Fig. 6.7).

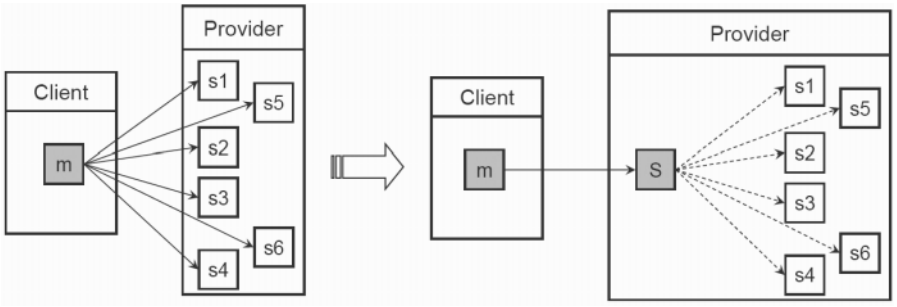


Fig. 6.7. The essence of the refactoring solution in case of Intensive Coupling

If this cluster of methods invoked from the same class consists mainly of lightweight methods, some of which are affected by Shotgun Surgery(133), then it is highly probable that the aforementioned

refactoring will also have a positive impact on the design quality of the class that contains those lightweight methods, in the sense that the the provided class will offer higher-level services.

The main reasons for reducing coupling are not that the code will look cleaner after. Most of the time reducing coupling is required to be able to use one component without the others or to make easier the replacement of one component by another one. Therefore having smaller communication channels is an important task. However, reducing coupling between classes is a complex task. Indeed we can reduce the metrics values by grouping or factoring the methods belonging to the same class and tunnelling thus the communication between the classes. However such a practice even if it can improve the overall design of the system by making more precise the communication channel between the classes should not hide that often reducing coupling is a more complex. Indeed either a dependency was useless and this is easy to fix it or it is necessary and moving it around will not solve the root of the problem. To reduce coupling often requires to change the *flow* of the application or to introduce extra indirections. In addition the coupling can change over time and run-time registration mechanisms such as Transform Type Checks to Registration [DDN02] may be the solution to decouple clients and providers of services.

Finally, coupling or dependencies are often the results of misplaced operations, therefore it is worth checking if the Law of Demeter [LH89] or reengineering patterns like Move Behavior Close to the Data and Eliminate Navigation Code [DDN02] can be applied.

6.4 Dispersed Coupling

Description

This disharmony reveals a complementary aspect of coupling than the one described as Intensive Coupling(120). This is the case of an operation which is excessively tied to many other operations in the system, and additionally these provider methods that are dispersed among many classes (see Fig. 6.8). In other words, this is the case where a single operation communicates with an excessive number of provider classes, whereby the communication with each of the classes is not very intense i.e., the operation calls one or a few methods from each class.

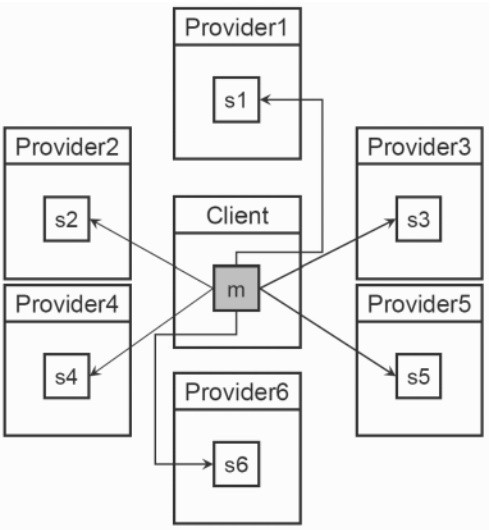


Fig. 6.8. Illustration of Dispersed Coupling

Applies To

Operations, e.g., methods or standalone functions.

Impact

Dispersively coupled operations lead to undesired ripple effects, because a change in an dispersively coupled method potentially leads to changes in all the coupled and therefore dependent classes.

Detection

The detection rule is defined in the same terms as the the one defined for Intensive Coupling(120), with only one complementary difference: we capture only those operations that have a high dispersion of their coupling (Fig. 6.9). The *detection strategy* in detail is:

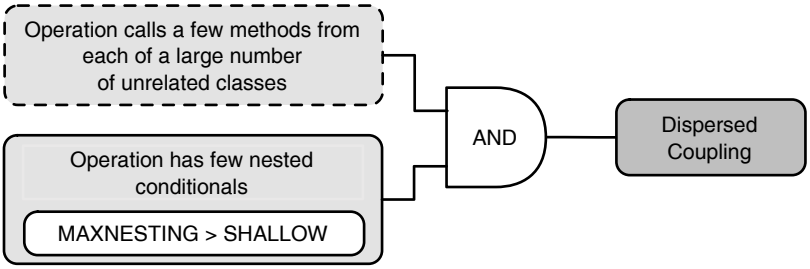


Fig. 6.9. Dispersed Coupling detection strategy

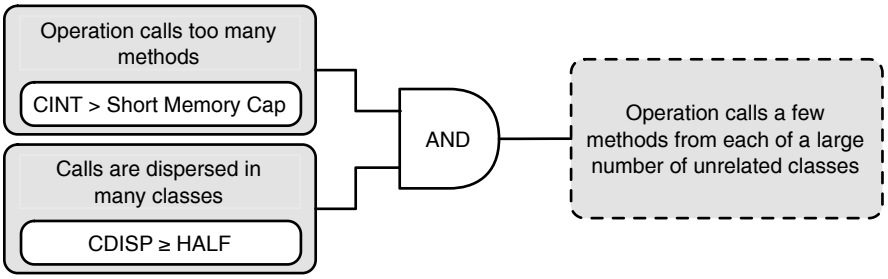


Fig. 6.10. In Dispersed Coupling operation calls a few methods from each of a large number of unrelated classes.

1. **Operation calls a few methods from each of a large number of unrelated classes.** This term of the detection rules imposes two conditions: an intensive coupling, i.e., the invocation of many methods from other classes (CINT - Coupling Intensity), and a large dispersion among classes of these invoked operations (CDISP - Coupling Dispersion). The metrics used in this case are the same as those already used in the context of detecting Intensive Coupling(120).

2. **Operation has few nested conditionals.** Exactly as for Intensive Coupling(120), we also set the condition that the calling operation should have a non-trivial nesting level, to make sure that irrelevant cases (like initializer functions) are skipped.

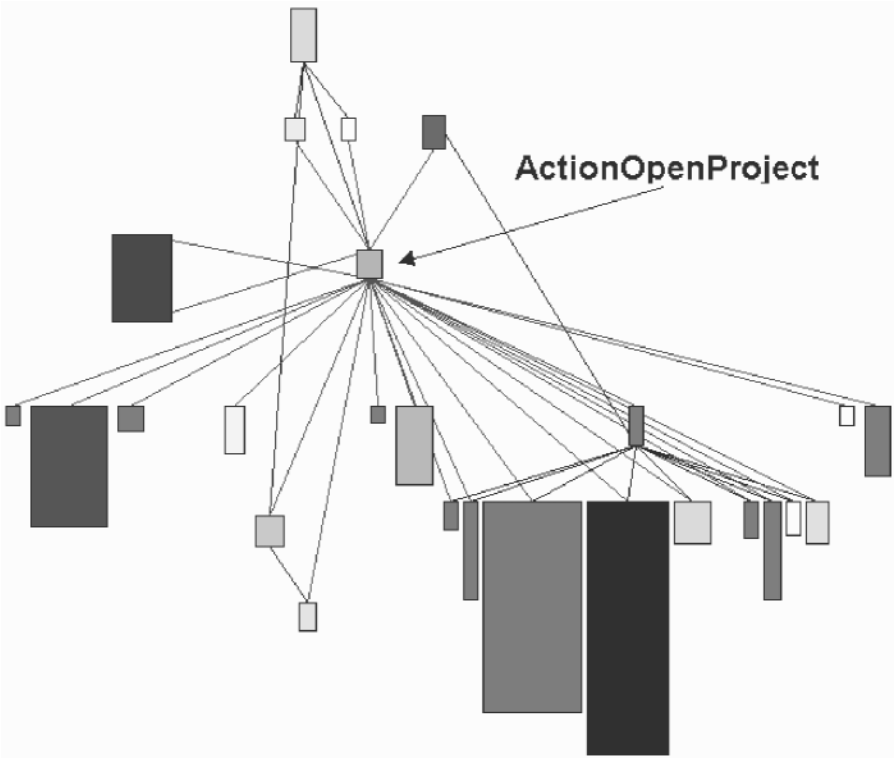


Fig. 6.11. The class ActionOpenProject is coupled with many classes. The red classes are non-model classes, i.e., belong to the Java library. The blue edges represent invocations.

Example

An interesting example of Dispersed Coupling is found in class ActionOpenProject. We see in Fig. 6.11 that the class is coupled with many other classes. Even ignoring the calls to non-model classes (colored in red) we still see that this methods of this class invoke methods located in many other classes, resulting in a great dispersion of the invocations. Looking closer at the methods of ActionOpenProject we discover that most of the coupling in this class is caused by two

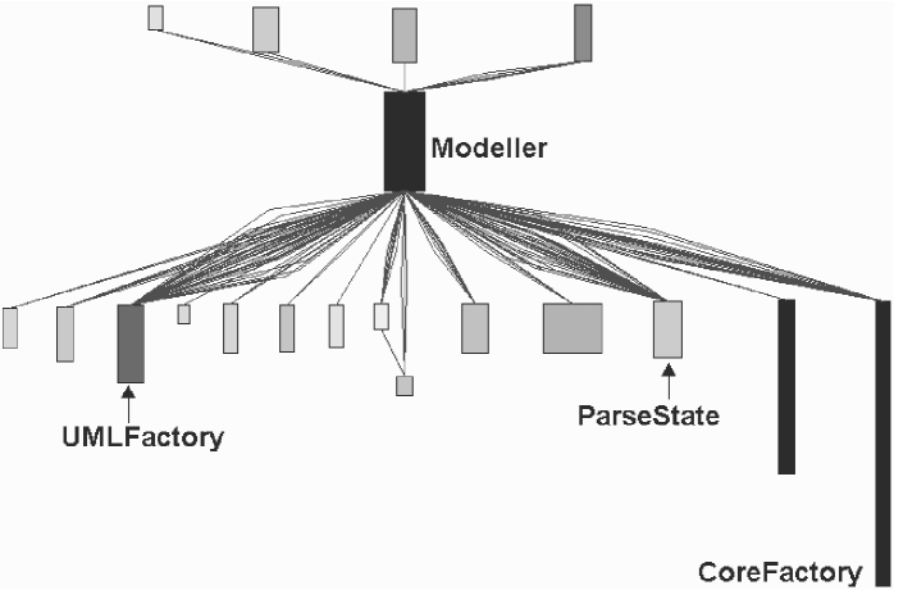


Fig. 6.12. The class Modeller is coupled with many classes and suffers itself from many other problems.

methods, i.e., actionPerformed and loadProject, both revealed by the *detection strategy* as being affected by Dispersed Coupling. By looking closer at these two methods we identify another interesting aspect: each of them is also a Brain Method(92). Moreover, some fragments of actionPerformed (exactly those fragments where many invocations appear) are also duplicated in three methods from sibling classes of ActionOpenProject. All these facts determine us to believe that the cause of the excessive coupling is the improper distribution of functionality among the methods of the system. In other words, the Dispersed Coupling detected in these methods is an additional sign that the two methods are doing more than one single task.

Another relevant example of Dispersed Coupling is found in a class already mentioned in the context of the Brain Method(92) disharmony, i.e., Modeller. This class has two methods affected by Dispersed Coupling. Again, one of the two methods (addImport) is a Brain Method(92); the other one (addClassifier) is also significantly large and complex. As we see in Fig. 6.12, Modeller is indeed dispersively coupled with many classes and especially coupled with ParseState,

CoreFactory and UMLFactory. A manual analysis of both these two methods has revealed two different causes:

1. Methods are too large and not focused on a single task. As a result, in each method, we find a portion of code that reveals an intensive coupling towards the ParseState class. Furthermore, inspecting ParseState we found out that this class provides only very simple services, which are composed into higher-level services by the client methods. This partially explains the need for many different methods invocations from that class.
2. In addClassifier apart from the aforementioned aspect, we found that the Dispersed Coupling disharmony is partially due to invocation chains that break the Law of Demeter [LH89]

Refactoring

Refactoring an operation affected by Dispersed Coupling is not a straight forward action. It needs more contextual information to proceed, but here are a few hints from our experience in dealing with this problem:

- In many cases the operation that exhibits Dispersed Coupling is also a Brain Method(92). In this case, the detailed knowledge about coupling will support the refactoring of the operation in terms of Brain Method(92). In other words, if you encountered a Brain Method the refactoring should address both aspects simultaneously.
- For the other cases (rather rare) the refactoring process should be centered on identifying called methods that are *lightweight* and/or affected by Shotgun Surgery(133), always with the question in mind: Isn't there anything in the client method (i.e., the one affected by Dispersed Coupling)that could be moved to one of the lightweight methods that it invokes.
- Calling many methods from lots of classes might also have another cause than the excessively large size of the invoker method. The cause might be that the operation invokes the wrong classes, i.e., that it it is coupled to classes that are at lower abstraction level than the client method [Rie96]. Thus, it would be necessary to identify the proper abstraction and let the client function communicate with that class. Although this sounds easy, it is hard to accomplish because it requires a good understanding of the system domain to be able to introduce a new abstraction into the system.

As mentioned while discussing refactoring solutions for Intensive Coupling(120), eventually coupling or dependencies are often the results of misplaced operations, therefore it is worth checking if the Law of Demeter [LH89] or the reengineering patterns Move Behavior Close to the Data and Eliminate Navigation Code [DDN02] can be applied.

6.5 Shotgun Surgery

Description

Not only *outgoing* dependencies cause trouble, but also *incoming* ones. This design disharmony means that a change in an operation implies many (small) changes to a lot of different operations and classes [FBB+99] (see Fig. 6.13). This disharmony tackles the issue of strong *afferent* (incoming) coupling and it regards not only the coupling *strength* but also the coupling *dispersion*.

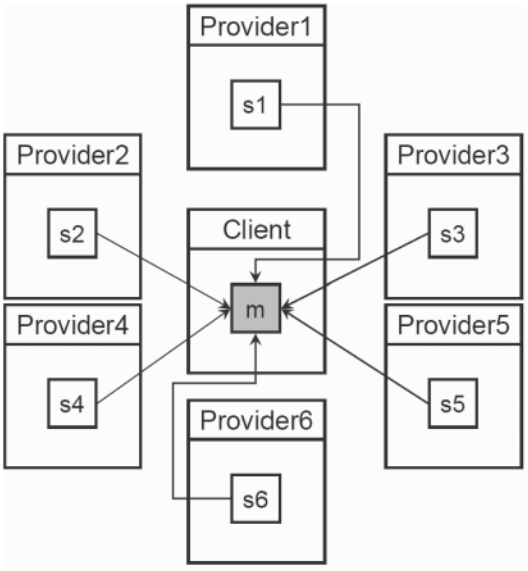


Fig. 6.13. Illustration of Shotgun Surgery

Applies To

Operations, e.g., methods or functions.

Impact

An operation affected by Shotgun Surgery has many other design entities depending on it. Consequently, if a change occurs in such an operation myriads of other methods and classes might need to change as well. As a result, it is easy to miss a required change causing thus maintenance problems.

Detection

We want to find the classes in which a change would significantly affect many other places in the system. In detecting the methods most affected by this disharmony, we consider both the *strength* and the *dispersion* of coupling. In contrast to Intensive Coupling(120) and Dispersed Coupling(127), here we are interested exclusively in *incoming dependencies* caused by function calls. In order to reveal especially those cases where dependencies are harder to trace, we will count only those operations (and classes) that are neither belonging to the same class nor to the same class hierarchy with the measured operation.

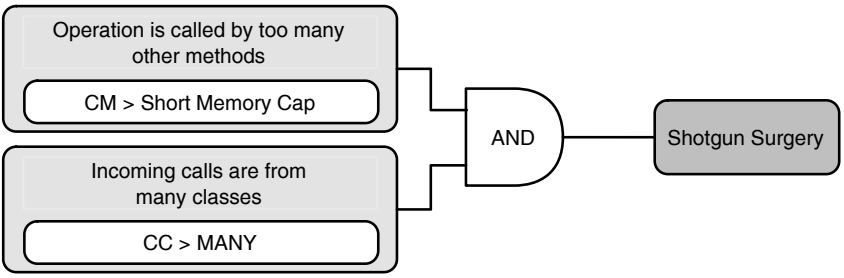


Fig. 6.14. Shotgun Surgery *detection strategy*

Based on all the considerations above, the detection technique is now easy to describe (see Fig. 6.14). First, we pick up those functions that have a *strong* change impact, and from these we keep only those that also have a high *dispersion* of changes. The *detection strategy* in detail is:

1. **Operation is called by too many other operations.** When a change in the measured operation occurs we must fix all the other operations that depend on it. If this exceeds our short-term memory capacity the risk of missing a dependency increases This justifies both the selection of the metric and of the threshold.
   An alternative way to quantify the strength of incoming dependencies is to count the *number of calls* instead of *the number of callers* (like CM (Changing Methods) does). The metric called *Weighted Changing Method* (WCM) defined it [Mar02a] does just that.
2. **Incoming calls are from many classes.** Using this metric and this threshold has the following rationale: assuming that we have

two operations, and that a change in each of them would affect 20 other operations, from these two, the one for which the 20 clients are spread over more classes is worse than the other one. In other words, if all dependencies were to come from methods of a few classes then the potential changes that need to be operated on these client methods would be more localized, reducing thus the risk of missing a needed change. As a consequence, the maintenance effort (and risk) involved in managing all changes would be more reduced. Therefore, we use the CC (Changing Classes) metric to quantify the *dispersion* of the changes, so that only those Shotgun Surgery functions causing most maintenance problem are detected.

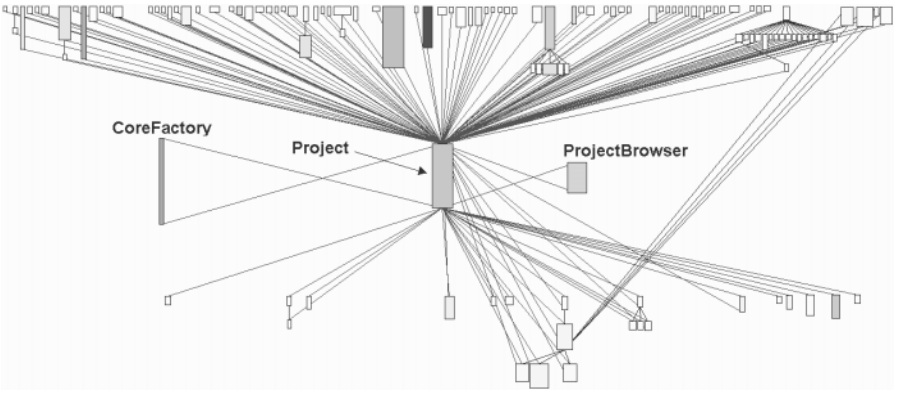


Fig. 6.15. Project provides an impressive example of a class with several methods affected by Shotgun Surgery(133). Due to these methods, Project is coupled with 131 classes (ModelFacade has been elided from the screenshot). Furthermore, the class has cyclic invocation dependencies with ProjectBrowser and CoreFactory. In the figure, the classes above Project depend on it, while Project itself depends on (i.e., invokes methods of) the classes below it.

Example

In Fig. 6.15 we see an extreme case of Shotgun Surgery(133) that involves several methods of class Project. The class is coupled with 131 classes (10% of *ArgoUML*) and has cyclic invocation dependencies with the classes ProjectBrowser and CoreFactory (the second largest class in the system). The classes above Project depend on it, while Project itself depends on (i.e., invokes methods of) the classes below

it. The view reveals how fragile the system is if a major change is performed on the class Project. Lots of classes in the whole system are potentially affected by changes.

Refactoring

How should we deal with Shotgun Surgery? We identified a number of refactoring options:

- Move more responsibility to the classes defining Shotgun Surgery methods, from the client classes of these functions. Do this especially when the definition classes of the Shotgun Surgery methods is *small* and/or *not complex* and/or it is or has a tendency to become a Data Class(88). Move Behavior Close to the Data [DDN02] presents step by step how to move behavior close to the data it uses and can be helpful here.
- The Shotgun Surgery methods that are very large and complex (e.g., tending to become a Brain Method) should be especially analyzed and taken care of by the maintainers of the system (e.g., by increasing the number of test cases for the method, or by trying to refactor it). We recommend this as such methods have a higher potential for change and/or malfunction potentially having a severe impact on the rest of the system. Identifying clearly the stable interfaces in the system is a good way to reduce the candidates for refactorings.

6.6 Recovering from Collaboration Disharmonies

Where to Start

As already mentioned in the previous chapter, in practice we need some criteria to prioritize the collaboration harmony offenders, so that we know which classes and methods are the most dangerous ones, the ones that require most our attention and that need a refactoring action to would improve the design. We use the following criteria in prioritizing the classes which host classification disharmonies:

- Classes that contain a higher number of disharmonious methods have priority
- Classes that are affected by other types disharmonies go first in order to reveal relation to other aspects of harmony.

How to Start

The three collaboration disharmonies Shotgun Surgery(133), Intensive Coupling(120) and Dispersed Coupling(127) address the issue of coupling from two complementary perspectives: While the issue of excessive coupling is addressed by Shotgun Surgery(133) from the provider's viewpoint, the other two tackle the same issue from the client's perspective. These two perspectives are like the two faces of the same coin: therefore, they cannot be addressed separately in a refactoring process.

Next we will provide some advice on how to approach the problem of coupling using the aforementioned *detection strategies*.

1. For each operation affected by Intensive Coupling group the invoked methods by their definition class. There will be one or more such groups containing 3 or more methods from one single class.
2. After that, collect the groups of "3-or-more-methods" (from the same provider class) from all operations affected by Intensive Coupling, and try to identify common groups. For those groups of methods that are used together in several client operations, try to introduce a new method in the provider class, and replace the multiple calls with a single call of the new method. Such a refactoring could have multiple beneficial consequences:
   - If these groups contain methods affected by Shotgun Surgery (and they usually do) the refactoring would reduce the number of clients for these methods, and thus reduce their incoming coupling. In many cases such a refactoring would help them recover from the Shotgun Surgery(133) disharmony.

- Provider classes for groups of "3-or-more-methods" are often lightweight classes, i.e., they do not have very much functionality, sometimes even being reported as being a Data Class; additionally, such classes may participate in violations of the Law of Demeter. Such refactorings would move some of the functionality to them, thus improving also the identity harmony.

3. Dispersed Coupling is a design problem that oftentimes affects Brain Method(92), because of the following reason: An excessively large and complex operation is almost always non-cohesive, doing more than one thing; and therefore there will be many invocations to methods from many classes. Thus, methods affected by Dispersed Coupling should be first checked to see whether they are also detected as Brain Method. If so, the Brain Method problem should be solved first, as this might eliminate as well the Dispersed Coupling.
4. Assume now that we consider the Brain Method problem as solved, but there are still methods affected by Dispersed Coupling. In this case, collect the groups of invoked methods from all the operations affected by Dispersed Coupling. Look at these groups trying to identify *clusters of methods* invoked from multiple client operations (affected by Dispersed Coupling(127)). For each such cluster, check if this is not an invocation chain, thus violating the Law of Demeter [LR89], and try to remove it [DDN02]. After all, you should check for such invocation chains in all methods affected by Dispersed Coupling, as in our experience these violations of the Law of Demeter are (apart from the operation being a Brain Method(92)) the primary cause of this disharmony.

As a final remark, note that if the aforementioned *detection strategies* will not flag anymore certain classes or methods as being affected by Shotgun Surgery– as a result of applying the proposed refactorings – it does not necessarily mean that you will be safe. We consider the proposed refactoring as a first step towards often more challenging issues. Rationalizing the communication between classes is a good approach to better understand deeper problems. As we already pointed out, changing the coupling between classes is not simple, since one dependency may still force you to load a complete package and moving dependencies around is often not as trivial as it seems. Some solutions may lead you to rethink totally the flow of the communication within your application or to introduce dynamic mechanisms to deal with the temporal aspects of the dependencies [DDN02].

7

Classification Disharmonies

7.1 Classification Harmony Rules

The object-oriented programming paradigm captures the *is-a-kind-of* relationship among classes with inheritance. This allows developers to write flexible and reusable code, but it can lead to disastrous designs if misused.

It is not enough for a class to be in harmony with itself; it also needs to be in harmony with its its ancestor and its descendant classes. The major cause of classification disharmonies is the misconception that inheritance is mainly a vehicle of code reuse (i.e., subclassing) rather than a means to assure that more specific objects can substitute more general ones (i.e., subtyping) [LP90, LW93a, Mar02b]. When inheritance is used solely for code reuse purposes maintenance can become painful because abstractions are not derived consistently.

The classification harmony rules are:

Classes should be organized in hierarchies having harmonious shapes

The identity of an abstraction should be harmonious with respect to its ancestors

Harmonious collaborations within a hierarchy are directed only towards ancestors, and serve mainly the refinement of the inherited identity

---

Proportion Rule

Classes should be organized in hierarchies having harmonious shapes

Rationale

Inheritance is at the same time a curse and a blessing of the object-oriented paradigm. The one extreme is given by applications where basically inheritance is ignored and the application is a flat collection of classes, leading to limited code reuse. The other extreme is overuse of inheritance, where the code is so heavily decomposed in a hierarchy that reading the code is equivalent to browsing excessively up and down the classes and methods in the hierarchy. Inheritance should be used with care and style.

Practical Consequences

- **Avoid wide hierarchies** – *Class hierarchies should not become too* wide*, i.e., avoid inflation of subclasses.*

  Excessively wide hierarchies oftentimes appear because of copy-paste-and-adapt patterns: the developers prefer to copy and modify existing code instead of refactoring the existing code. Since developers do not get to *see* such things they underestimate the effect of copy-paste practices.

- **Avoid tall hierarchies** – *Class hierarchies should not become too* tall*. Avoid very narrow and deep hierarchies.*

Presentation Rule

The identity of an abstraction should be harmonious with respect to its ancestors

Rationale

The concept of inheritance allows for writing compact code and also for the reuse of the code already implemented in one of its ancestor classes. In this sense a descendant should always be *in sync* with what has been defined by its ancestors, and not reinvent the wheel or duplicate the code.

Practical Consequences

- **Extend interface smoothly** – *Keep an harmonious proportion between* tradition *and* novelty*. In other words, keep a balance between the inherited interface and its extension (through addition of new services).*

- **Specialize behavior smoothly** – *Keep an harmonious proportion between* evolution *and* revolution *of behavior. In other words, do not refuse (deny, "cut off") any parts of an ancestor's interface and specialize rather than override the inherited services (i.e., the inherited public methods).*

- **Decrease abstractness smoothly** – *The abstractness level for the set of services of a class (together with their "inheritable" helper methods i.e., the protected ones) should be inversely proportional to the distance to the top of the hierarchy. Thus, root classes should be rather abstract or the other way around: abstract classes should be situated close to the top of a hierarchy and not somewhere in the middle of a hierarchy.*

Implementation Rule

> *Harmonious collaborations within a hierarchy are directed only towards ancestors, and serve mainly the refinement of the inherited identity*

Rationale

This rule could be rephrased as: *The implementation dependency of a class on its ancestor should be unidirectional and serve the refinement of the inherited services.* The rule states that a subclass should not depend on its ancestors just for the sake of code reuse (i.e., by calling methods from its base classes (only) from newly defined methods).

Practical Consequences

- **Dependencies go bottom-up** – *Base classes should not depend on their descendants.*

  Despite the truth behind this, there is a hidden world of horrors where developers who do not have a "complete picture" of the system just reuse pieces of code whenever they see something useful to them.

- **Dependencies serve specialization** – *Inherited operations should be used (i.e., redefined, called, specialized) most of the time in the context of refining (specializing) the inherited services, rather than calling them from newly added services.*

7.2 Overview of Classification Disharmonies

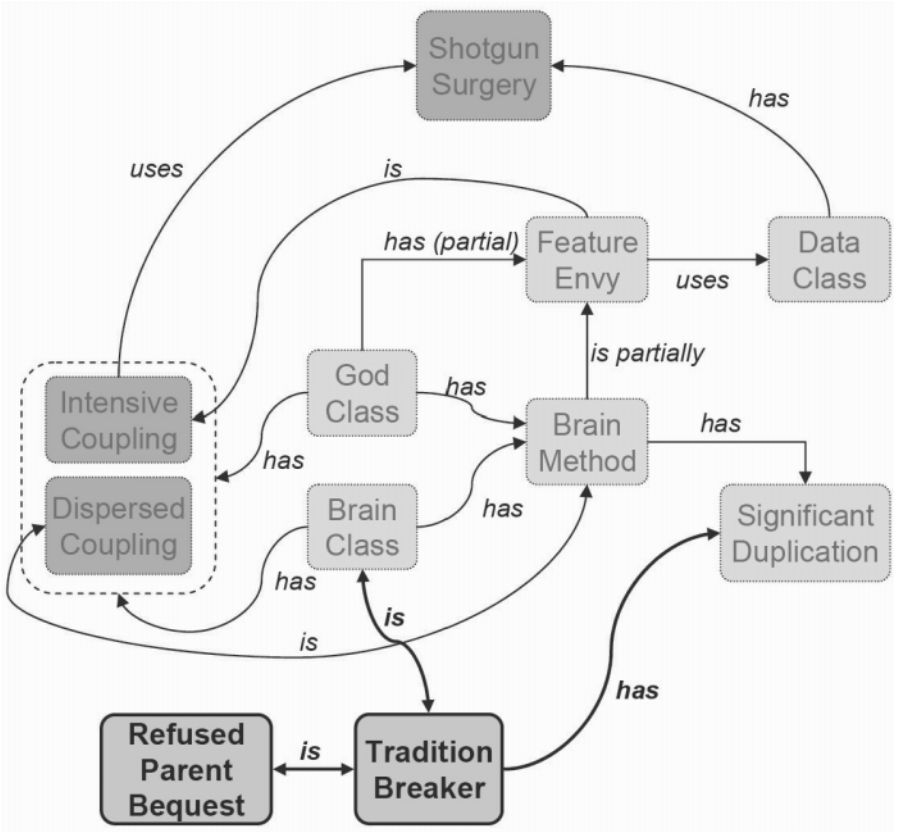


Fig. 7.1. Correlation web of classification disharmonies.

Considering the three harmony rules presented above, and based on our experience with analyzing object-oriented systems, we defined a set of patterns that capture the most disturbing classification disharmonies.

We have to mention Duplication(102) again, this time between inheritance-related classes. So, this is the first disharmony we are going to consider. This is often a symptom that goes together with other disharmonies. But even if there is no duplication in a hierarchy, it still needs to be harmonious with respect to its ancestors, as stated by the Presentation Rule (see Sect. 7.1). Distortions of this harmonious relation to the parent class(es) 7.1 appear as:

1. The derived class denies the inherited *bequest* [FBB+99] (Refused Parent Bequest(145)).
2. The derived class massively extends the interface of the base class with services that do not really characterize that family of abstractions (Tradition Breaker(152))

The shape of the hierarchy itself says a lot about the classification harmony. As we will see, in most cases the Refused Parent Bequest(145) and Tradition Breaker(152) disharmonies appear in an over-bloated hierarchy with an inflation of classes.

In conclusion, while inheritance is (also) a powerful mechanism to reuse code, subtyping is the actual point because it supports a better understanding of a hierarchy than subclassing, since a subclass is a more specialized version of its ancestor and not an unrelated concept that is there because it can reuse some code.

Another difficult issue related to inheritance is when is it useful to introduce a new class in the system. Often developers are afraid of having many small classes and prefer to work instead with fewer but larger classes. Developers often believe that they will have less complexity to manage if they have to deal with fewer classes. It is better to have more classes conveying meaningful abstractions than having a single large one. However, having useless classes or classes without meaningful behavior is not good either because they pollute and complicate the abstraction space: The challenge is to find the right level of abstraction.

7.3 Refused Parent Bequest

Description

Inheritance is a mechanism dedicated to support incremental changes. Consequently, the relation between a parent class and its children is intended to be an intimate one, more special than the collaboration between two unrelated classes. This special collaboration is based on a category of members (methods and data) especially designed by the base class to be used by its descendants, i.e., the protected members. But if a child class refuses to use this special bequest prepared by its parent [FBB+99] then this is a sign that something is wrong within that classification relation.

Applies To

Classes. The Following conditions are assumed:

1. the inspected class has a superclass;
2. the superclass is neither a third-party class (e.g., library class), nor is it an interface.

Impact

The primary goal of inheritance is certainly code reuse. However, extending base classes without looking at what they have to offer introduces duplication and in general class interfaces that become incoherent and non-cohesive. An often overlooked part of the process when adding or extending subclasses is to study the superclasses and determine what can be reused, what must be added and finally what could be pushed into the superclasses to increase generality.

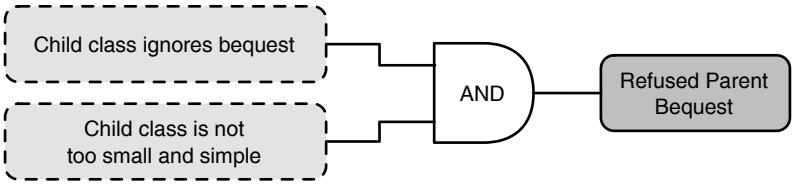


Fig. 7.2. Detection strategy for Refused Parent Bequest.

Detection

As illustrated in Fig. 7.2 the detection of such disharmonious classes is based on two main conditions: (1) a low usage of inheritance-specific members from the base class and (2) the detected class must have at least an average size and complexity, otherwise the finding is irrelevant as the bequest refusal might be due to its small size. In other words, the second condition ensures that the bequest refusal is an intentional rather than a circumstantial fact. The *detection strategy* in detail is:

1. **Child class ignores bequest.** What do we mean by "a child class uses the parent's *bequest*"? We mean that it does one of the following:
   - it calls a protected method defined in the parent class
   - it accesses a protected attribute defined in the parent class
   - it overrides or specializes a method defined in the parent class

   To assess how much a child class depends on its parent class in an inheritance-specific way, we used two metrics: (1) The Base-class Usage Ratio (BUR), which quantifies the usage of protected members; and (2) the Base-class Overriding Ratio (BOvR), which quantifies the degree of overriding and specialization of base class methods. The third metric we use, Number of Protected Members (NPrM), just makes sure that there is a specific bequest to use, i.e., that there are at least several protected members. We use these metrics in the following way:
   a) **Parent provides more than a few protected members.** The bequest prepared by the parent class should be significant i.e., the base class has more than a few members declared as protected (in other words, members intended to be used specifically in the context of the inheritance relation).
   b) **Child uses only little of parent's bequest.**
   c) **Overriding methods are rare in child.** Overriding or specializing methods from the base class is a rare case in the derived class. Thus, the fraction of base class methods that are overridden or specialized is very low.
2. **Child class is not too small and simple.** We say about a child class that it intentionally refuses a bequest if it is large and complex enough; otherwise the child class can have the excuse of refusing the bequest because it is too small. Therefore, this term finds those classes that are both significantly large (in terms of methods (NOM)) and complex (Fig. 7.3).
   There are two alternative conditions for considering the complexity of the class significant: either the average CYCLO/method is high

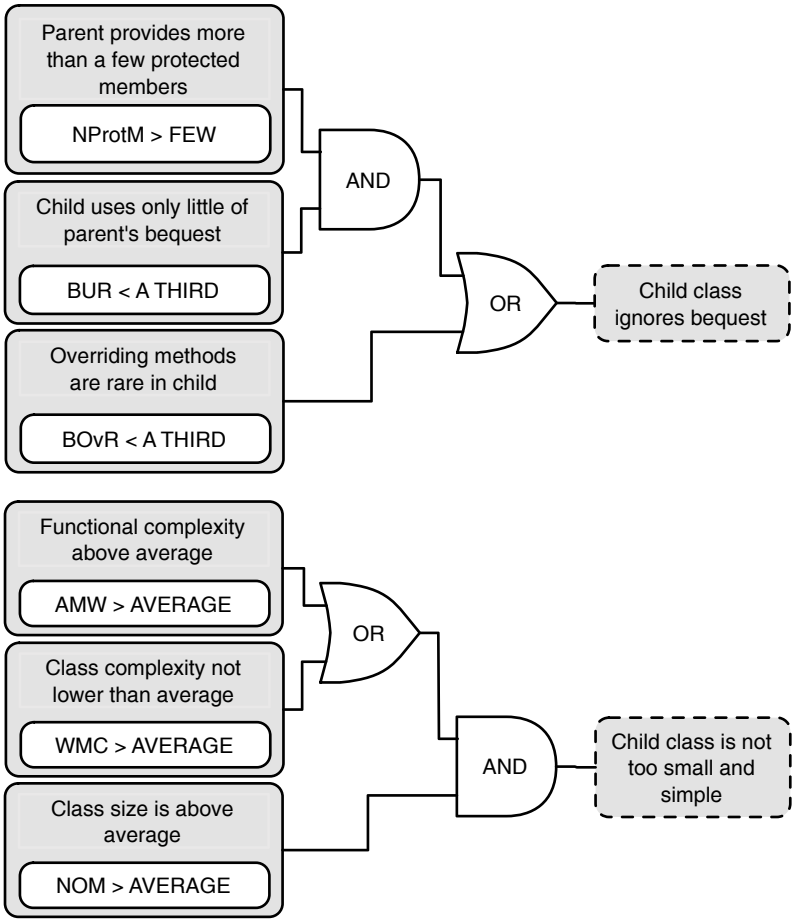


Fig. 7.3. Main components of the Refused Parent Bequest detection strategy.

enough, or the class is large and thus the cumulative complexity (WMC) makes it relevant. The used metrics are:

a) **Functional complexity above average.**
b) **Class complexity not lower than average.**
c) **Class is above average.**

Example

The unusual form of this hierarchy (see Fig. 7.4) already gives us a first hint that its classes are afflicted by some problems. Moreover, the fact that there is an abstract class (called ToDoPerspective) in the

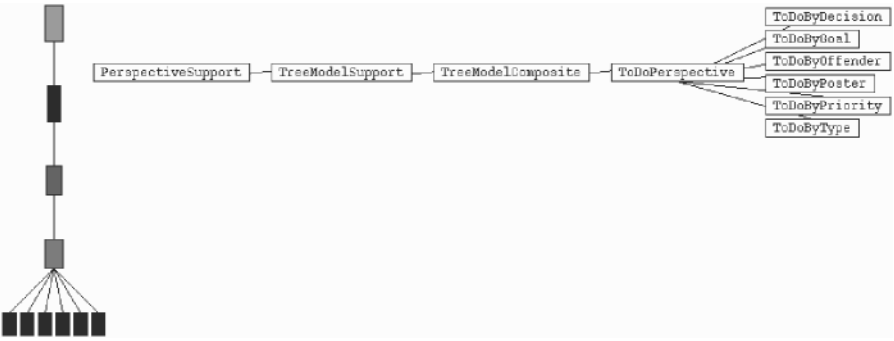


Fig. 7.4. A *System Complexity* view of the PerspectiveSupport hierarchy.

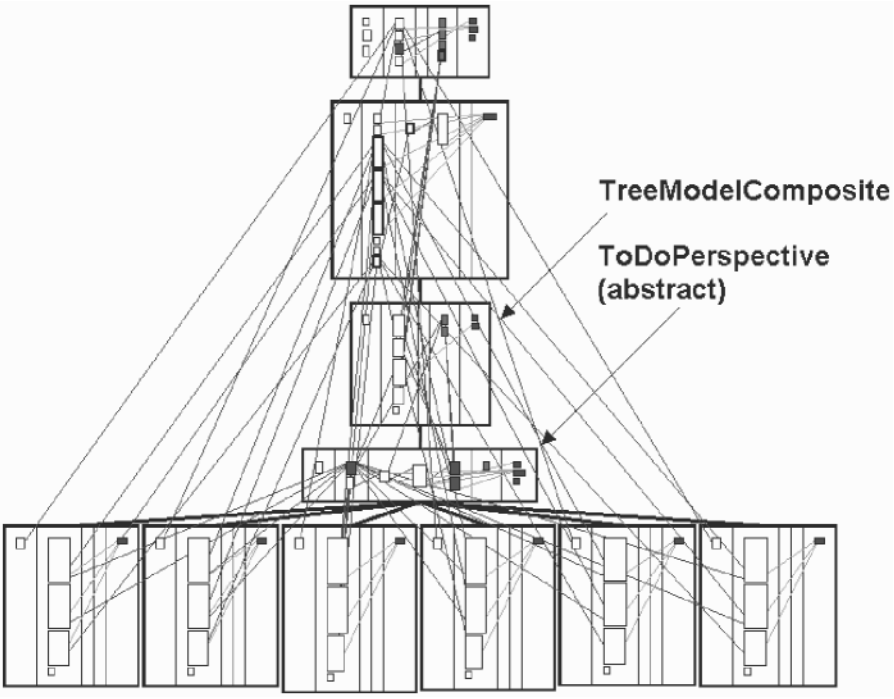


Fig. 7.5. A *Class Blueprint* view of the PerspectiveSupport hierarchy.

middle of the hierarchy also gives us hints about potential problems related to inheritance. The *Class Blueprint* of this hierarchy depicted in Fig. 7.5 shows a suspicious regularity in size among the methods implemented in the six leaf classes, hinting at duplication. The

class TreeModelComposite is affected by Refused Parent Bequest: it basically ignores what is implemented in the two superclasses.

Refactoring

If we want to remove a Refused Parent Bequest disharmony from a class then we need to follow the detailed (inspection and refactoring) process depicted in Fig. 7.6. The figure has three areas (labeled A, B and C) corresponding to one of the three identified causes for a Refused Parent Bequest. Notice that some of the three causes might co-exist. Next, we are going to describe these three cases in detail.

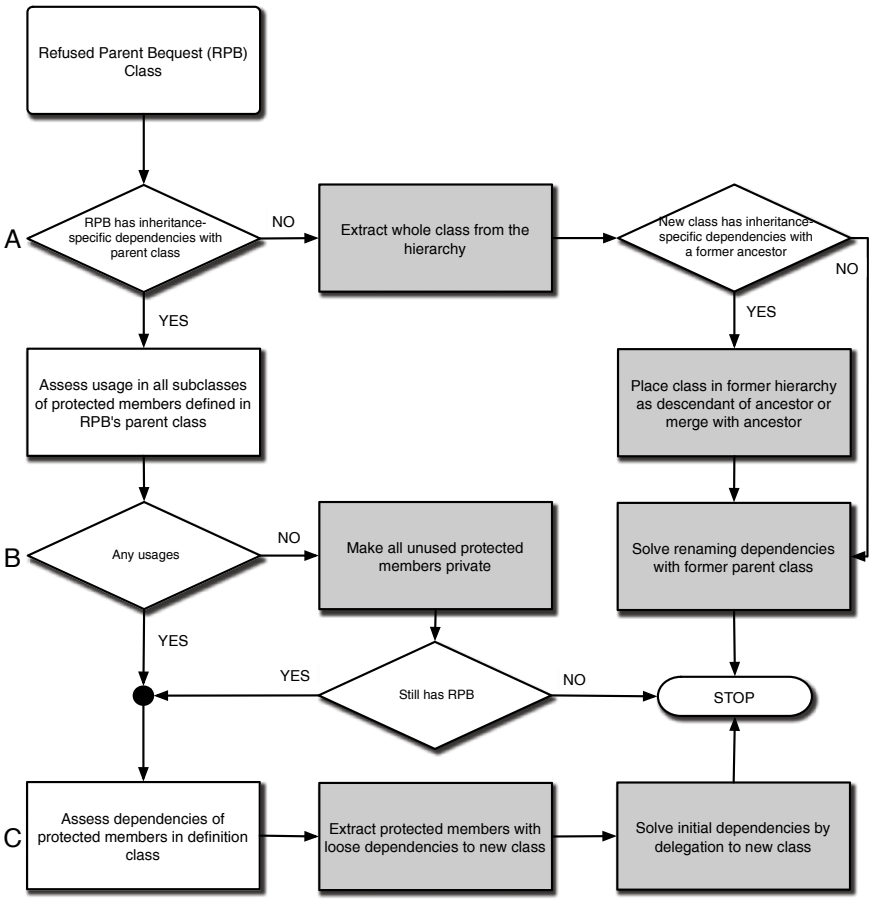


Fig. 7.6. Inspection and refactoring process for a Refused Parent Bequest.

Case A: False Child Class

In this case the cause of the problem is that the child class simply does not belong in the hierarchy; in other words, the hierarchy might be ill-designed. The more relevant symptom for this case is when the child class has *no inheritance-specific dependencies* on the parent class.

In some cases this goes together with the Tradition Breaker(152) disharmony. An interesting aspect is that in some cases the "false child" *does* belong to the hierarchy, but as a child class of another parent (i.e., an initial "grandparent" or ancestor). This can be found out by analyzing the dependencies between the disharmonious class and the other ancestors.

Case B: Irrelevant Bequest

In this case the Refused Parent Bequest design flaw appears as a result of the fact that the space of inheritance-specific members is over-populated with methods and attributes that have no relevance in the context of the inheritance relation.

But how do we detect that a (part of the) bequest is irrelevant? We have to count, for each protected member, the number of usages from derived classes; in case of protected methods, this includes overriding or specialization of that method in derived classes). If the number of dependencies is null, i.e., if a member is used only from inside the definition class, then it should be moved to a private scope.

Case C: Discriminatory Bequest

The third case, probably the most interesting one, is when the parent class has many child classes, and the bequest offered by it is relevant only for some of these siblings, but not for the class affected by Refused Parent Bequest. By cumulating the bequest needed by various subsets of descendants, the total bequest becomes excessively large. Consequently, the main symptoms in this case are:

- A large number of descendants.
- Often, there is more than one class exhibiting Refused Parent Bequest in the same hierarchy.
- Each descendant uses a small, non-overlapping portion of the total bequest.

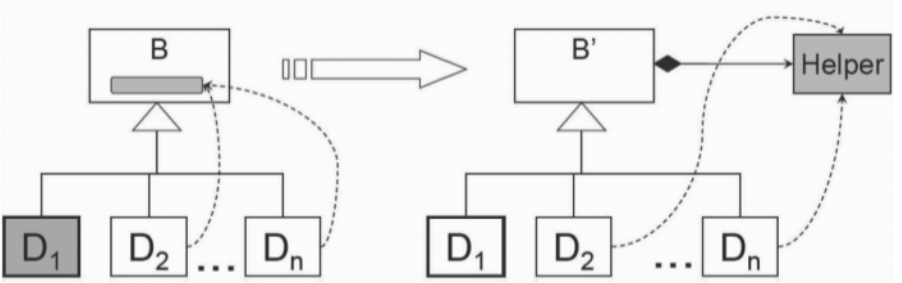


Fig. 7.7. Refactoring for Refused Parent Bequest in case of *Discriminatory Bequest*.

At first sight we could improve the design in this case by splitting class B in two classes, (B" derived from B') adding an intermediary layer in the inheritance tree and letting each initial subclass of B be derived either from B' or B" depending on which bequest they need to inherit. Unfortunately, this applies only for simple cases which involve few protected members used in common by only subsets of the derived classes.

For the general case, the situation can be improved by extracting the parts that are not used by all descendants to a helper class, and letting the parent class have a reference to an instance of the helper class (see Fig. 7.7). If this refactoring is applicable, then this could be also the sign that the base class was capturing more than a single abstraction. This way, the base class is easier to understand because it does contain less protected members which do not characterize the entire hierarchy.

7.4 Tradition Breaker

Description This design disharmony strategy takes its name from the principle that the interface of a class (i.e., the services that it provides to the rest of the system) should increase in an evolutionary fashion. This means that a derived class should not break the inherited "tradition" and provide a large set of services which are unrelated to those provided by its base class.

Of course, it is OK for a child class to contain more intelligence than its parent i.e., to offer more services. But if the child class hardly specializes any inherited services and only adds brand new services which do not depend much on the inherited functionality, then this is a sign that something is wrong either with the definition of the child's class interface or with its classification relation. In the Suggested Refactoring (155) section we analyze in more detail the possible causes and solutions for this problem.

Applies To Classes. If C is the name of the class, the following conditions are assumed: (1) C has a base class B, (2) B is not a third-party class and (3) B is not an interface.

Impact When adding subclasses without examining the functionalities implemented in the superclass(es) one might break the tradition kept up by the superclasses. This could be called "disrespectful" inheritance.

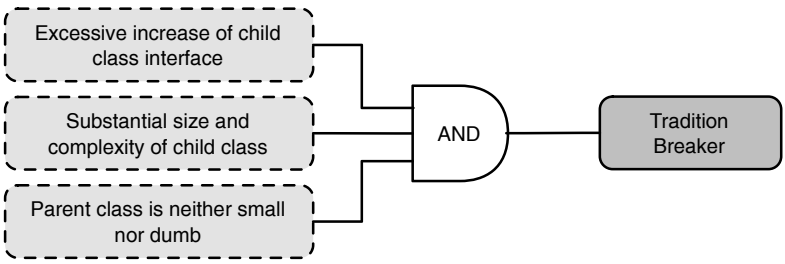


Fig. 7.8. The Tradition Breaker detection strategy.

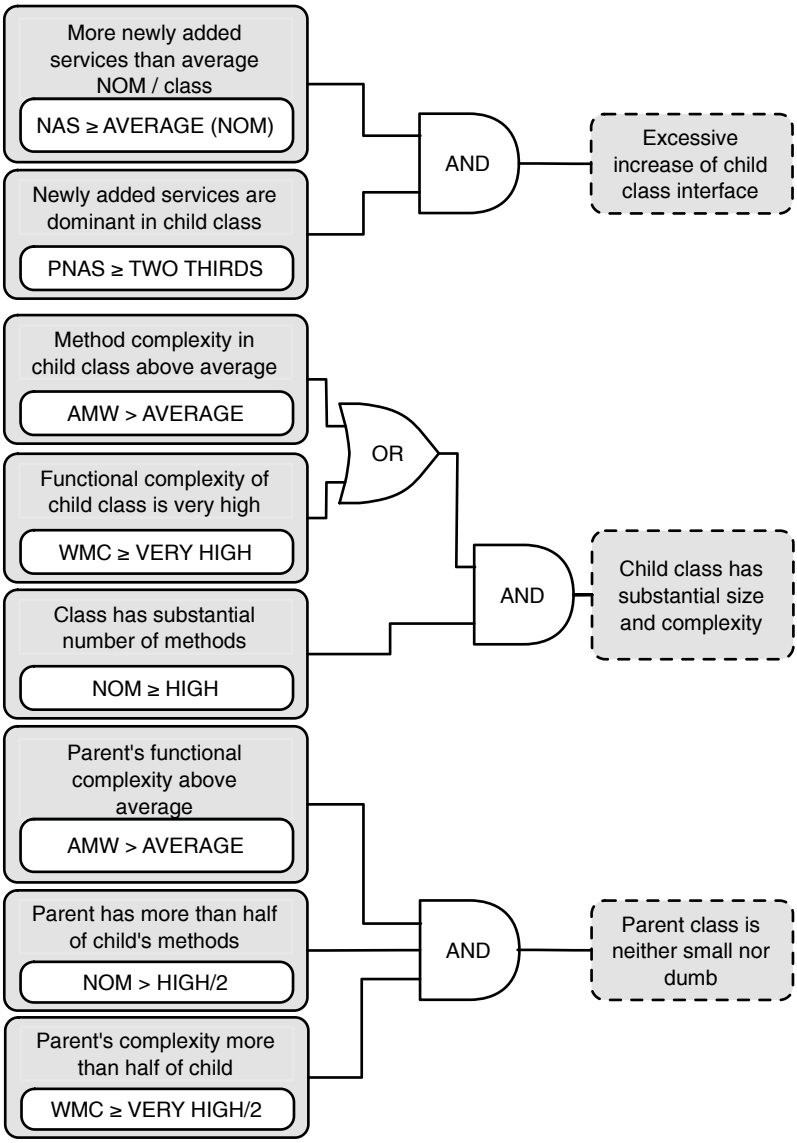


Fig. 7.9. Main components of the Tradition Breaker detection strategy

In Fig. 7.8 we see a high-level view of the detection rule for a Tradition Breaker. There are three main conditions that must be simultaneously fulfilled for a class to be put on the blacklist of classes that

Detection

break the inherited tradition by the interface that they define. These conditions are:

- The size of the public interface of the child class has increased excessively compared to its base class.
- The child class as a whole has a considerable size and complexity.
- The base class, even if not as large and complex as its child, must have a "respectable" amount of functionality defined, so that it can claim to have defined a tradition.

1. **Excessive increase of child class interface.** To quantify the evolution of a child's public interface compared to that of its parent, we use two measures: (1) Newly Added Services (NAS) tells us in absolute values how many public methods were added to the class; and (2) the Percentage of Newly Added Services (PNAS) which shows us the percentile increase, i.e., how much of the class's interface consists of newly added services. We used these metrics in the following way:
   a) **More newly added services than average number of methods per class.** This threshold is based on the statistical information related to the number of methods per class (see Table 2.1), using the following logic. If a class adds more new methods than the *average* number of methods (public or not) of a class then the measured class is an outlier with respect to NAS. For Java this average value[1] is 6.5.
   b) **Newly added services are dominant in child class.** We use this metric to make sure that the absolute value provided by the NAS is a significant part of the entire interface of the measured class. Therefore, PNAS is a normalized metric and we set the threshold so that NAS represents at least *two-thirds* of the public interface.
2. **Child class has substantial size and complexity.** To speak about a relevant Tradition Breaker the child class must contain a substantial amount of functionality. This means that it must have a substantial size (measured in this case by the number of methods) and accumulate a significant amount of logical complexity. Therefore we require either the average complexity or the total complexity of the class to be high. An additional requirement is that the child class has a significant number of methods (NOM). We use the following metrics (see Fig. 7.9):

[1] Computed as the average between the *lower value* and *upper value* of NOM/Class.

a) **Method complexity in child class above average.**
b) **Functional complexity of child class is very high.**
c) **Class has a substantial number of methods.**

3. **Parent class is neither small nor dumb.** We cannot say that a child class breaks a tradition if the tradition defined by the parent class is insignificant. In other words, this term sets a minimal condition on the size and complexity of the parent class, i.e., this must satisfy *at least half* of the requirements imposed on the child class. Additionally, its average complexity must be higher than the average value. In this context, AMW and WMC are the two metrics used to quantify the average and the total amount of functional complexity respectively, while NOM quantifies the size of the class in terms of method number. The used metrics are:
   a) **Parent's functional complexity above average.**
   b) **Parent has more than half of child's methods** With respect to NOM, the parent class should satisfy at least *half* of the requirements we set for the child (see term "*Child class has substantial size and complexity*").
   c) **Parent's complexity more than half of child** With respect to Weighted Method Count (WMC), the parent class should satisfy at least *half* of the requirements we set for the child (see term "*Child class has substantial size and complexity*").

Example

In Fig. 7.10 we see a *System Complexity* view of the hierarchy whose root class is named FigNodeModelElement. Visually striking is that the hierarchy is top-heavy (the root class is by far the largest in terms of methods and attributes) and unbalanced (there is a sub-hierarchy on the left). Moreover, many direct subclasses of FigNodeModelElement look similar "from the outside" (i.e., they have a similar shape, pointing to a possible duplication problem), and as we will see also from the inside.

From the point of view of the disharmonies, nearly half of the classes of this hierarchy are afflicted by at least one of two classification disharmonies: Refused Parent Bequest(145) or Tradition Breaker.

Among the subclasses of FigNodeModelElement there is one in particular which is striking because it is the only one which is both affected by Refused Parent Bequest(145) and is also a Tradition Breaker, namely FigObject. Additionally this class is also a Brain Class(97) that contains two methods which are Brain Method(92).

Refactoring

If we want to remove a Tradition Breaker then we need to follow the

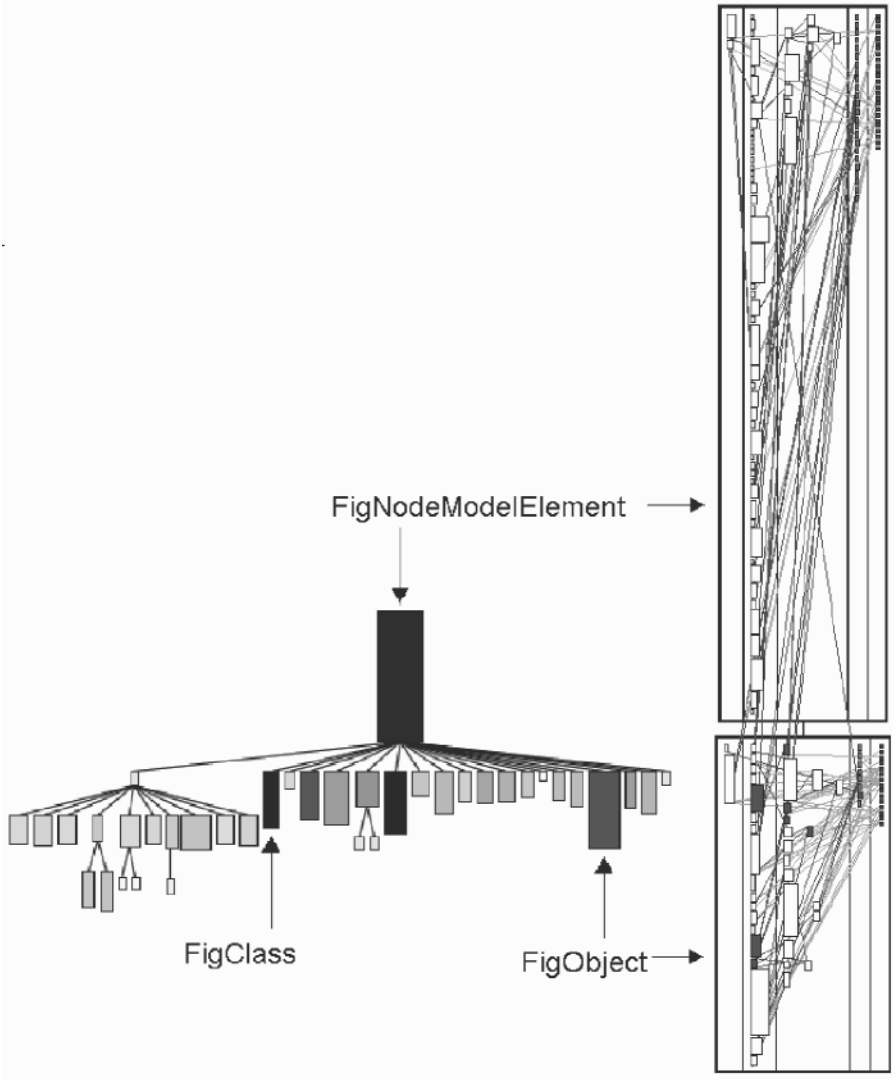


Fig. 7.10. A *System Complexity* view of the FigNodeModelElement hierarchy.

detailed (inspection and refactoring) process depicted in Fig. 7.11. The figure has four areas (labeled A, B, C and D) corresponding to one of the four identified cases, which may also co-exist, for a Tradition Breaker: irrelevant tradition in subclass, denied tradition in base class, double-minded subclass, or misplaced subclass.

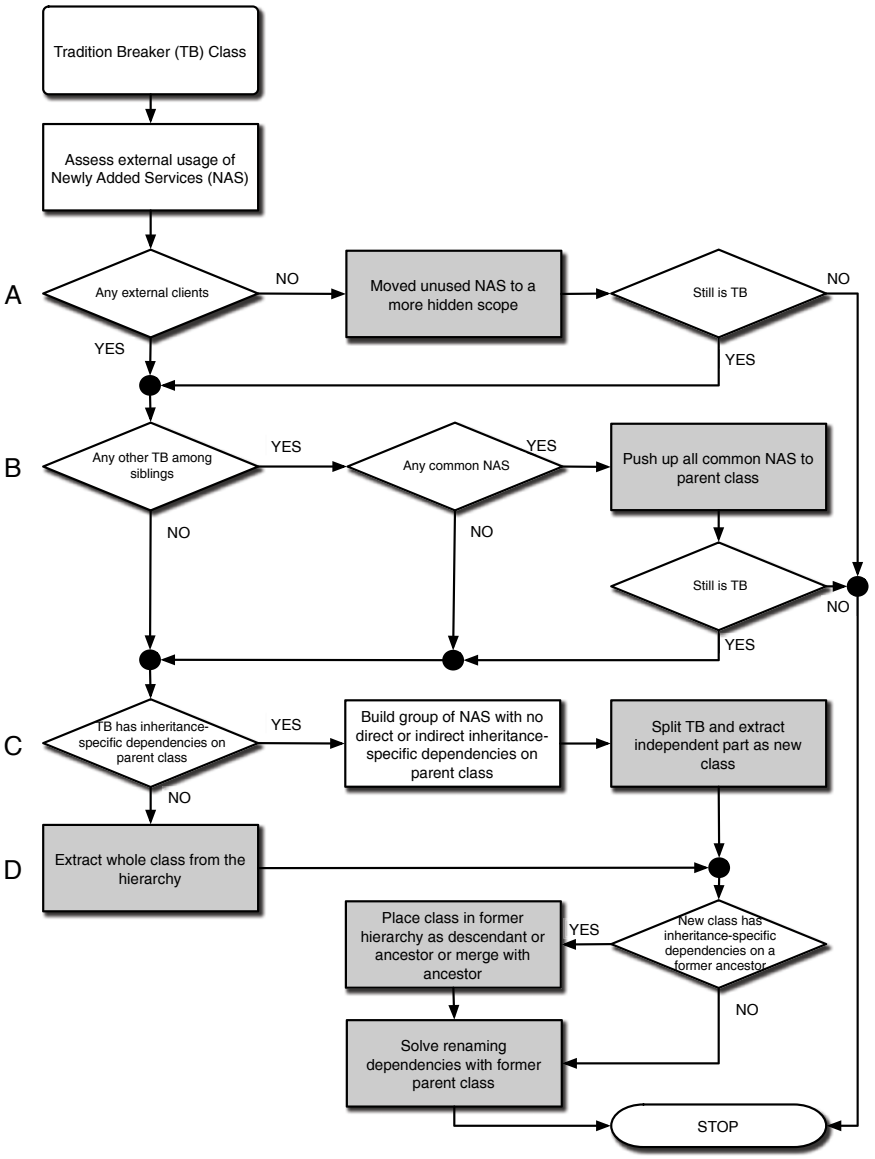


Fig. 7.11. Inspection and refactoring process for a Tradition Breaker.

Case A: Irrelevant Tradition

In this case the derived class has an excessively large interface, i.e., it includes in its interface methods that should have been declared

protected or private. In other words, the methods newly added in the interface of the Tradition Breaker class are just helper methods, mistakenly declared public. This can be be found out by analyzing the usages of the method from other classes.

Case B: Denied Tradition

The base class does not include a set of services that are implemented in all (or most) derived classes. Consequently, it is common that some of the Tradition Breaker's siblings also show the symptoms of a Tradition Breaker. In most of these cases Duplication(102) is also present.

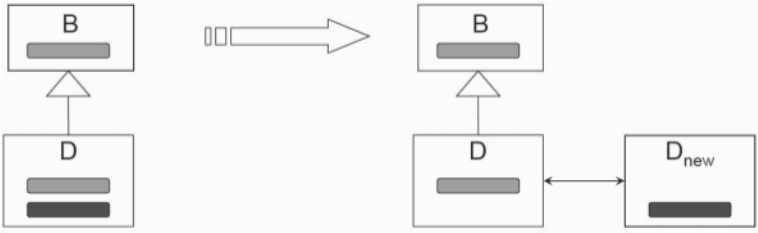


Fig. 7.12. Extracting the "second mind" of a Tradition Breaker to a separate class.

Case C: Double-Minded Descendant

In this case the problem is that the derived class is "double-minded", whereby only a part of its interface (and implementation) belongs to the hierarchy where it was placed. The part of the class that wants to stay in the hierarchy can be identified by a set of methods that override/specialize/use methods of the base class. The "other mind" of the class breaks the tradition, by doing something totally different, that has nothing in common with the base class. In this case, the part that does not belong to the hierarchy could be moved outside the hierarchy to a separate class (see Fig. 7.12).

Case D: Misplaced Descendant

The extreme case of Case C is when the whole Tradition Breaker defines a behavior that is not extending (specializing) in any way the behavior found in its base class. Thus, the interfaces of the base class and of the derived class are totally different and there is no real inheritance-specific dependencies on the base class. When this is the case, it is highly probable that the class is also affected by Refused Parent Bequest(145).

7.5 Recovering from Classification Disharmonies

Where to Start

In order to recover from from classification disharmonies (i.e., design problems related to inheritance) it is insufficient to look at the individual suspect classes; hierarchies must be analyzed as a whole. In this context it becomes important to know *how to group* the different classes identified as affected by various disharmonies and also, how to prioritize the hierarchies that need more urgent attention. In practice, we use the following criteria in selecting the hierarchies with the most significant amount of classification disharmonies:

- Hierarchies with more classes affected by classification disharmonies have priority.
- If the disharmonies are distributed on many hierarchy levels (i.e., if the sub-hierarchy affected by disharmonies is *deep*) the inspection priority for the hierarchy increases.
- Hierarchies where most distinct classification disharmonies appear have a higher priority.
- Hierarchies where other types of disharmonies (i.e., identity and collaboration) co-exist with violations of classification harmony must also be regarded with increased interest.

For the purpose of prioritizing the hierarchies to be inspected first, we mainly use the following quantification means:

- *Number (and Percentage) of Classes with Classification Disharmonies.* These numbers tell us how much the classification disharmonies are spread within the hierarchy. In addition to the absolute number of classes, we also display the percentage of disharmonious classes, as this indicator is more relevant and easier to interpret for larger hierarchies. The higher these numbers are, the higher also is the probability that the whole hierarchy must be restructured.
- *Hierarchy Depth of Disharmonious Classes.* In addition to the previous values, we found that it is important to know also how deep in the hierarchy (rooted by the class in the table) we find disharmonies. If disharmonies are propagated on many inheritance levels then such hierarchies must be definitely revisited.
- *Distinct Classification Disharmonies in Hierarchy.* Often the same disharmony (Tradition Breaker(152)) affects many subclasses of a

hierarchy. But if the number of *distinct* problems in the same hierarchy is high then the hierarchy has a more complex problem that needs to be addressed. At the same time the co-existence of some classification disharmonies (e.g., Refused Parent Bequest(145) and Tradition Breaker(152)) could help us in addressing the problem properly.

- *Distinct Number of Other Disharmonies in Hierarchy.* To have an even better overview of all possible disharmonies that appear in the same hierarchy, we also count how many *distinct* design problems, other than the classification ones (i.e., problems related to identity or collaboration), can be found.

How to Start

Assessing and improving the classification harmony of a system is a complex process, because a large number of classes are involved (i.e., all (or most) of the classes in the hierarchy) and also because the real cause of such design problems is not localized in one single class (e.g., a child class is detected, but the real cause of the problem is in the base class). Additionally, the inspection and refactoring process is painful because of the existence of various correlated design disharmonies (see Fig. 4.12) that might occur in the classes of the hierarchy and that must be solved at the same time.

Because of all these reasons, for each of the two classification disharmonies discussed in this chapter, i.e., Refused Parent Bequest(145) and Tradition Breaker(152), we addressed in detail the potential refactoring solutions. We noticed that the *order* in which the problems are addressed is very important. Therefore, we recommend inspecting and refactoring each disharmonious hierarchy in your system using the sequence described in Fig. 7.13.

Doing the refactorings in this order is important because on the one hand the refactoring action for one disharmony can have positive consequences with respect to the following ones (in the sense that the refactoring effort is reduced); but on the other hand they can also introduce additional cases of classification disharmonies that must be addressed as well.

Let us see how it happens. We start by solving the Duplication(102) problem. By doing so, it could be possible that methods are extracted from some siblings and moved to their parent class. This can contribute in some cases to a reduction – or even a total elimination – of the Tradition Breaker(152) disharmony. But at the same time,

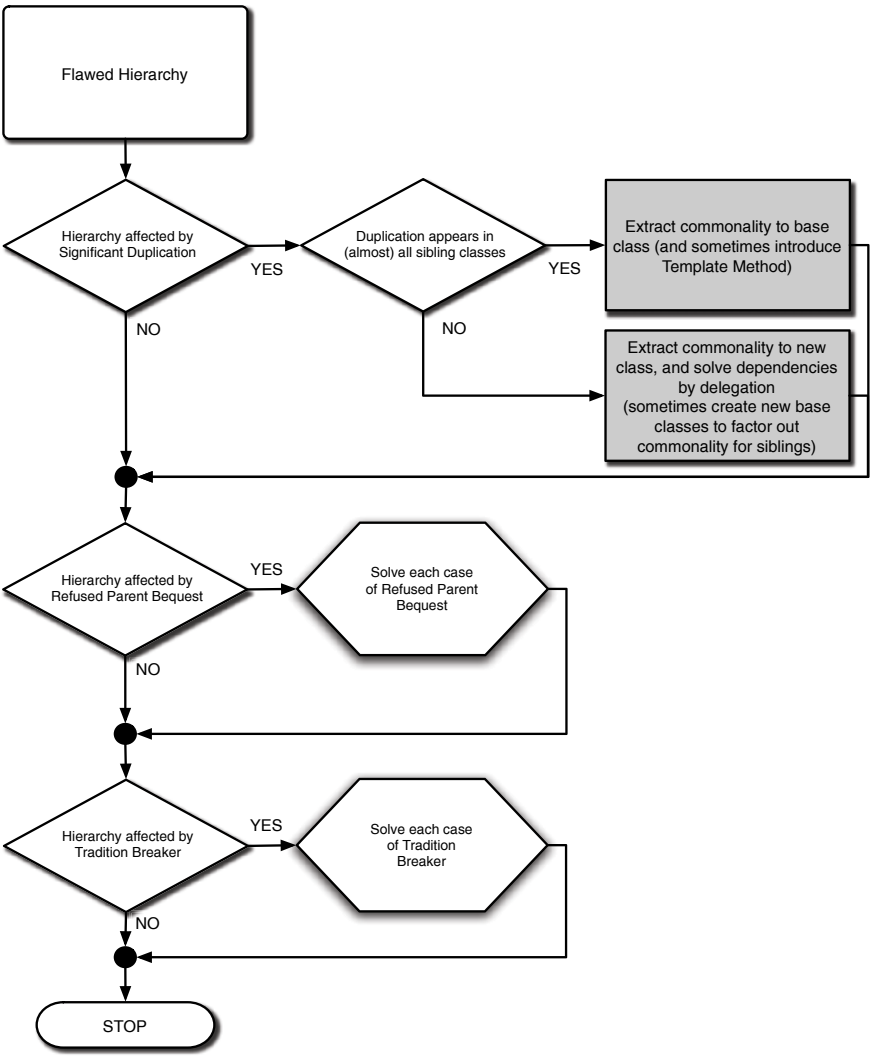


Fig. 7.13. How to address classification disharmonies.

as we move new methods to a parent class, we might cause a Refused Parent Bequest(145) disharmony for some other siblings, as the bequest provided by the parent class has increased as a result of the refactoring. Thus, it is important to deal with Duplication(102) before addressing the Refused Parent Bequest(145) and the Tradition Breaker(152) disharmonies.

Now, which one of these two problems should we address next? We suggest dealing first with Refused Parent Bequest(145), because by refactoring a part of the class (or even the whole child class) needs to be removed from the hierarchy. Thus, this provides a new perspective in dealing with the cases of Tradition Breaker(152).

A

Catalogue of Metrics Used in the Book

In this appendix you will find definitions of the metrics used throughout this book. These metrics are neither the best in the world, nor magic. We chose them in the context of the *detection strategies* presented in Chapter 4. Thus, their most efficient use is in the context of these strategies.

A.1 Elements of a Metric Definition

Before presenting the catalogue with the metrics definitions we need to introduce the main elements that appear in a metric definition. As we have discussed in Sect. 1 (see page 7), all metrics have something in common: they can be expressed in terms of three orthogonal elements: HAVING, USING, BEING.

Therefore, because all definitions of the object-oriented metrics that we are going to present next are defined in these terms, we will first describe how these three elements become concrete in the object-oriented design.

HAVING in Object-Oriented Design

In Fig. A.1 we see all the containment (HAVE and BELONGS-TO) relations that are relevant in the context of object-oriented design, i.e., what other entities does the measured entity **have** (**contain**), in the sense of being a scope for these entities? This also includes the inverse relation: to which entity does the measured entity **belong to**? For example, an operation *has* parameters and local variables, while it *belongs to* a class.

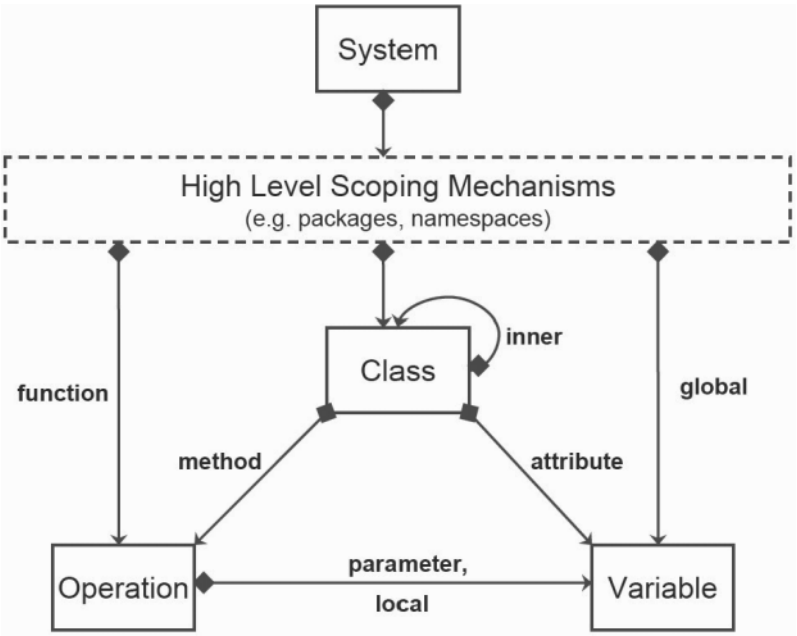


Fig. A.1. The HAVING relations in object-oriented design.

USING in Object-Oriented Design

In Fig. A.2 we see all direct usage (USE and USED-BY) relations that are relevant in the context of object-oriented design, i.e., what entities does the measured entity **use**; and again the inverse relation: by which entities is the measured one **being used**? For example, an operation is *using* the variables that it accesses, while it *is used by* the other operations that call (invoke) it. A class uses another class by extending it through inheritance, but also uses other classes by communicating with them.

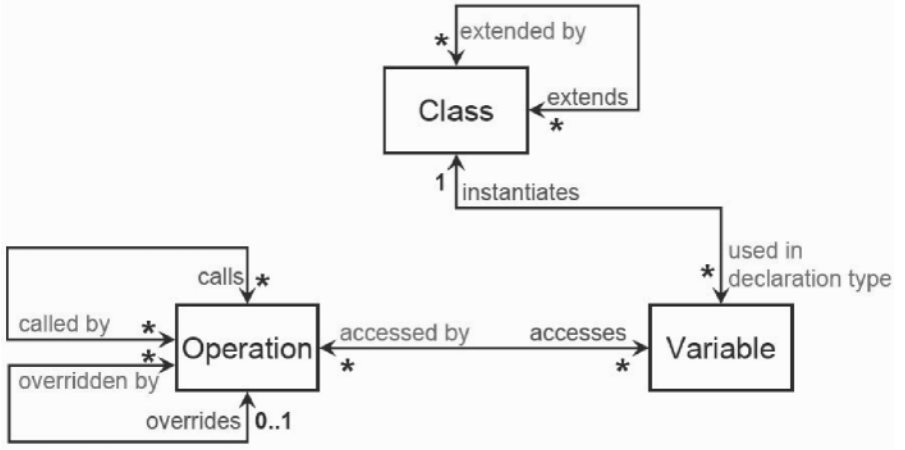


Fig. A.2. The USING relations in object-oriented design.

BEING in Object-Oriented Design

One of the most frequently encountered dilemmas when reading any metric definition is: What is *really* counted? Just think about a simple metric like *Number of Methods*. At first sight it looks straightforward. But at second thoughts various questions pop up: Are constructors included? Are inherited methods counted as well? What about accessor methods (i.e., getters/setters)?

Browsing through an extensive set of object-oriented design metrics we identified a set of recurring issues that appear in the definition of metrics. We summarize them below in form of a non-exhaustive set of questions:[1]

- **Constructors/Destructor.** Should constructors and destructor be counted as methods of a class?
- **Abstract Methods.** Should abstract methods be counted as methods of a class?
- **Inherited Members.** Should members (data and operation) inherited from ancestor classes be counted in a derived class?
- **Static Members.** When should class members, i.e.,*static* attributes and operations, be counted?

[1] The goal of this book is neither to list all possible questions nor to answer them, but to put you, the reader, in a position where you can ask and answer such questions yourself.

A.2 Alphabetical Catalogue of Metrics

AMW - Average Method Weight								
Definition	The average static complexity of all methods in a class. McCabe's cyclomatic number is used to quantify the method's complexity (Mar02a, McC76)							
Used for	Refused Parent Bequest(145), Tradition Breaker(152)							
Measured Entity	Class	Definition	Abstract					
		user-defined	–					
Involved Relations								
HAS	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		measured class	all	+	+	+	+	

ATFD - Access To Foreign Data								
Definition	The number of attributes from unrelated classes that are accessed directly or by invoking accessor methods (Mar02a)							
Used for	God Class(80), Feature Envy(84)							
Measured Entity	Class	Definition	Abstract					
		user-defined	+					
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		user-defined	all	+	+	+	–	
Involved Relations								
USES (accesses)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		measured class or method	all	+	–	+	–	
	Attribute	Definition	Visibility	Static	Const.			
		neither measured class nor a class from the same hierarchy	public	+	–			
USES (calls)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		measured class or method	all	+	–	+	–	
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		neither measured class nor a class from the same hierarchy	public	only	–	–	–	

BOvR - Base Class Overriding Ratio							
Definition	The number of methods of the measured class that override methods from the base class, divided by the total number of methods in the class						
Used for	Refused Parent Bequest(145)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
USES (overrides)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	–	–	–
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		in base class of measured class	all	+	–	–	+
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		in measured class	all	+	–	–	–

BUR - Base Class Usage Ratio							
Definition	The number of inheritance-specific members used by the measured class, divided by the total number of inheritance-specific members from the base class						
Used for	Refused Parent Bequest(145)						
Measured Entity	Class	Definition	Abstract				
		user-defined	+				
Involved Relations							
USES (accesses)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	+	+	–
	Attribute	Definition	Visibility	Static	Const.		
		in base class of the measured class	prot.	+	+		
USES (calls)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class or method	all	+	+	+	–
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		neither measured class, nor a class from the same hierarchy	prot.	only	+	–	–

CC - Changing Classes							
Definition	The number of classes in which the methods that call the measured method are defined in (Mar02a)						
Used for	Shotgun Surgery(133)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	+
Involved Relations							
IS USED (called by)	Class	Definition	Abstract				
		user-defined, scope of operations called from the measured operation (see CM)	+				

CDISP - Coupling Dispersion							
Definition	The number of classes in which the operations called from the measured operation are defined, divided by CINT						
Used for	Intensive Coupling(120), Dispersed Coupling(127)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-
Involved Relations							
USES (calls)	Class	Definition	Abstract				
		user-defined, scope of operations called from the measured operation (see CINT)	+				

CINT - Coupling Intensity							
Definition	The number of distinct operations called by the measured operation						
Used for	Intensive Coupling(120), Dispersed Coupling(127)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-
Involved Relations							
USES (calls)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	+

CM - Changing Methods							
Definition	The number of distinct methods that call the measured method (Mar02a)						
Used for	Shotgun Surgery(133)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	+
Involved Relations							
IS USED (is called by)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

CYCLO - McCabe's Cyclomatic Number							
Definition	The number of linearly-independent paths through an operation (McC76)						
Used for	Brain Method(92)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

FDP - Foreign Data Providers							
Definition	The number of classes in which the attributes accessed — in conformity with the ATFD metric — are defined						
Used for	Feature Envy(84)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-
Involved Relations							
USES (called by)	Class	Definition	Abstract				
		user-defined, scope of attributes and accessor methods used as in ATFD	+				

HIT - Height of Inheritance Tree							
Definition	The maximum path length from a class to its deepest subclass (Mar02a)						
Used for	*Overview Pyramid*(24)						
Measured Entity	Class	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	+
Involved Relations							
USES (extended by)	Class	Definition	Abstract				
		direct or indirect subclasses of measured class	+				

LAA - Locality of Attribute Accesses							
Definition	The number of attributes from the method's definition class, divided by the total number of variables accessed (including attributes used via accessor methods, see ATFD), whereby the number of local attributes accessed is computed in conformity with the LAA specifications						
Used for	Feature Envy(84)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-
Involved Relations							
USES (accesses)	Attribute	Definition	Visibility	Static	Const.		
		definition class of measured method	priv.	+	+		

LOC - Lines of Code							
Definition	The number of lines of code of an operation, including blank lines and comments (LK94)						
Used for	Brain Method(92), Brain Class(97)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

MAXNESTING - Maximum Nesting Level							
Definition	The maximum nesting level of control structures within an operation						
Used for	Intensive Coupling(120), Dispersed Coupling(127)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

NAS - Number of Added Services							
Definition	The number of public methods of a class that are not overridden or specialized from the ancestor classes						
Used for	Tradition Breaker(152)						
Measured Entity	Class	Definition	Abstract				
		user-defined	+				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		in measured class, not overriding methods	public	+	-	-	+

NOAM - Number of Accessor Methods							
Definition	The number of accessor (getter and setter) methods of a class						
Used for	Data Class(88)						
Measured Entity	Class	Definition	Abstract				
		user-defined	-				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	public	only	-	-	-

NOAV - Number of Accessed Variables							
Definition	The total number of variables accessed directly from the measured operation. Variables include parameters, local variables, but also instance variables and global variables						
Used for	Brain Method(92)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-
USES (accesses)	Attribute	Definition	Visibility	Static	Const.		
		user-defined	all	+	–		

NOM - Number of Methods							
Definition	The number of methods of a class						
Used for	Refused Parent Bequest(145), Tradition Breaker(152)						
Measured Entity	Class	Definition	Abstract				
		user-defined	+				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	+	+	+

NOPA - Number of Public Attributes					
Definition	The number of public attributes of a class				
Used for	Data Class(88)				
Measured Entity	Class	Definition	Abstract		
		user-defined	-		
Involved Relations					
HAS (contains)	Attribute	Definition	Visibility	Static	Constant
		measured class	public	-	–

NProtM - Number of Protected Members								
Definition	The number of protected methods and attributes of a class							
Used for	Refused Parent Bequest(145)							
Measured Entity	Class	Definition	Abstract					
		user-defined	+					
Involved Relations								
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		measured class	prot.	+	–	+	+	
	Attribute	Definition	Visibility	Static	Const.			
		measured class	prot.	+	+			

PNAS - Percentage of Newly Added Services							
Definition	The number of public methods of a class that are not overridden or specialized from the ancestors, divided by the total number of public methods						
Used for	Tradition Breaker(152)						
Measured Entity	Class	Definition	Abstract				
		user-defined	+				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		in measured class, not overriding methods	public	+	–	–	+
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		in measured class	public	+	–	–	+

TCC - Tight Class Cohesion							
Definition	The relative number of method pairs of a class that access in common at least one attribute of the measured class (BK95)						
Used for	God Class(80), Brain Class(97)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
USES (accesses)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	–	+	–
	Attribute	Definition	Visibility	Static	Const.		
		measured class	all	+	+		

WMC - Weighted Method Count							
Definition	The sum of the statical complexity of all methods of a class. The CYCLO metric is used to quantify the method's complexity (CK94, McC76)						
Used for	Refused Parent Bequest(145), Tradition Breaker(152), God Class(80), Data Class(88), Brain Class(97)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	+	+	+

WOC - Weight Of a Class							
Definition	The number of "functional" public methods divided by the total number of public members (Mar02a)						
Used for	Data Class(88)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	public	–	–	+	–
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	public	only	–	–	–
	Attribute	Definition	Visibility	Static	Const.		
		measured class	public	+	–		

B

iPlasma

B.1 Introduction

iPlasma [1] is an integrated environment for quality analysis of object-oriented software systems that includes support for all the necessary phases of analysis: from model extraction (including scalable parsing for C++and Java) up to high-level metrics-based analysis, or detection of code duplication. *iPlasma* has three major advantages: extensibility of supported analysis, integration with further analysis tools and scalability, as were used in the past to analyze large industrial projects of the size of millions of code lines (e.g., Eclipse and Mozilla).

B.2 *iPlasma* at Work

Fig. B.1 presents the layered structure of the *iPlasma* quality assessment platform. Notice that the tool platform starts directly from the source code (C++or Java) and provides the complete support needed for all the phases involved in the analysis process, from parsing the code and building a model up to an easy definition of the desired analyses including even the detection of code duplication, all integrated by a uniform front-end, namely INSIDER. Let's take a closer look at the layers of *iPlasma* .

MEMORIA and the Model Extractors

An essential task in a software analysis process is the construction of a proper model of the system. The information contained in the model

[1] *iPlasma* stands for Integrated PLAtform for Software Modeling and Analysis.

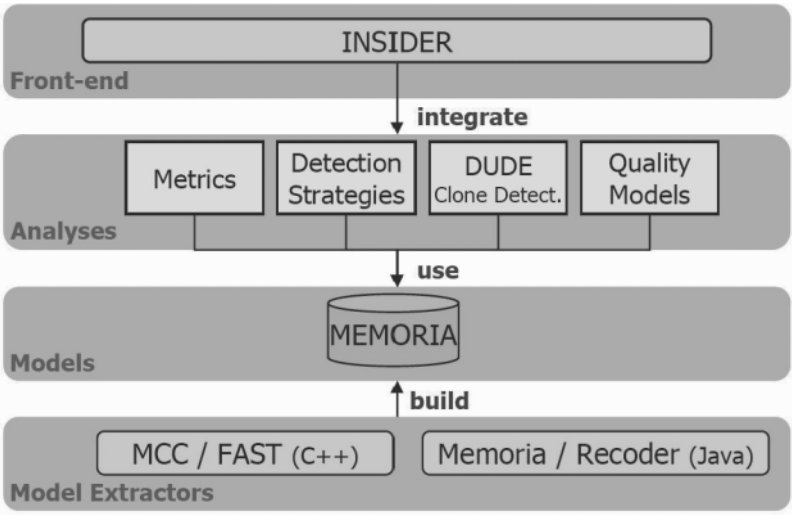


Fig. B.1. The layered structure of the *iPlasma* quality assessment platform.

strongly depends on its usage scenarios. As *iPlasma* is intended to support mainly analyses focused on object-oriented *design*, it is important to know the *types* of the analyzed system, the *operations* and *variables* together with information about their *usages* (e.g., the *inheritance relations* between classes, the *call-graph* etc.). In *iPlasma* we defined MEMORIA as an object-oriented meta-model that can store all the above information (and more). One of the key roles of MEMORIA is to provide a consistent model even in the presence of incomplete code or missing libraries, to allow the analysis of large systems and to ease the navigation within a system.

Extracting such a model from the source code requires powerful and scalable parsing techniques. Currently, *iPlasma* supports two mainstream object-oriented languages *i.e.,* C++and Java. For Java systems we use the open-source parsing library called RECODER [2] to extract all the information required by the MEMORIA meta-model. For C++code we have MCC, a tool which extracts the aforementioned design information from the source code (even incomplete code!), and produces a set of related (fully normalized) ASCII tables containing the extracted design information (including even information about templates). Although this information is eventually loaded in form

[2] See http://recoder.sourceforge.net/

of a MEMORIA model, the ASCII tables could be easily loaded in a RDBMS and interrogated in the form of SQL queries.

Analyses for Quality Assessment

Based on the extracted information several types of analyses (e.g., metrics, metrics-based rules for detecting design problems, quality models, etc.) can be defined.

Metrics and Detection Strategies

iPlasma contains a library of more than 80 state-of-the-art and novel design metrics, measuring different types of design entities from operations to classes and packages. All the metrics used in this book are defined in *iPlasma* . In Sect. 4.1 we showed how *detection strategies* allow us to combine metrics in more complex rules for detecting design problems. In *iPlasma* *detection strategies* can be implemented and adapted. Of course, all the *detection strategies* introduced in this book are already available for use.

Dude: Detection of Code Duplication

As we have seen in the book, an important issue is the detection of code duplication. In *iPlasma* the detection of code duplication is supported using the DUDE tool. DUDE uses textual comparison at the level of lines of code in order to detect portions of duplicated code. It has a powerful detection engine which can also cover some fine changes to the duplicated code such as renaming of some variables, changes in indentation or comments. The most important aspect about DUDE is that it can annotate a MEMORIA model with all extracted information about the presence of duplicated code. This makes possible to *correlate* duplications with their context (e.g., detect operations from sibling classes that contain duplication).

Insider: the Integrating Front-end

Assessing the design quality of an object-oriented system requires the collaboration of many tools. Using them independently can easily transform the analysis process into a nightmare, making it completely unscalable for usage on large-scale systems. One of the key aspects of *iPlasma* is that all these analyses are *integrated* and can

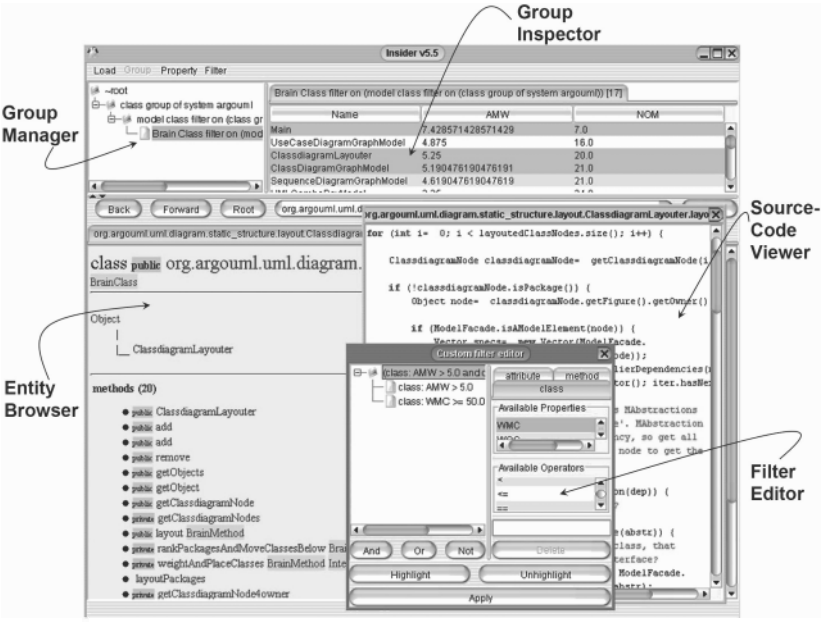


Fig. B.2. Key elements of INSIDER, the front-end of *iPlasma* .

be used in a uniform manner through a flexible front-end, called INSIDER. In other words, INSIDER is a front-end (see Fig. B.2) which offers the possibility to integrate independent analyses (in the form of plugins) in a common framework. This approach makes INSIDER *open implemented* and thus easily extendable with any further needed analyses.

Using INSIDER

In order to use INSIDER first a project must be loaded by indicating the folder where the source code of the project is located. During the loading phase, the sources are parsed and the model is constructed. After that, the system can be analyzed using the three major zones of the user interface (see Fig. B.2), namely:

- *Group Inspector.* In the top-right part of the screen a selected group of design entities (e.g., classes and operations) are displayed. Initially, the *Group Inspector* displays a group with only one entity: the system itself. In a display we can choose a number of metrics

(over 80) that should be displayed. As seen in Fig. B.2 the metrics are displayed for all the entities in the group. By selecting an entity in the group we can display another group associated with that entity (e.g., for a class, display the group of methods defined in the class or the group of its ancestors, etc.).

- *Group Manager.* During a software analysis we usually need to work with more than a single group. The groups that are currently open are displayed on the top-left side of the screen. The *Group Manager* allows us to select a group that we want to see in the *Group Inspector*. It also allows us to delete those groups that are no longer relevant for the analysis. Last but not least, the *Group Manager* allows us to create a new group, by filtering the entities of the selected group based on a filtering condition, i.e., a combination of metrics (as in *detection strategies*). Apart from the predefined filters, new filters can be defined at run-time using the *Filter Editor* (see windows at the bottom-right of Fig. B.2). The *Filter Editor* can be used not only to create new (sub)groups, but also for revealing the entities in a group that fulfill the defined filtering condition (see red highlighting in the *Group Inspector* in Fig. B.2).
- *Entity Browser.* When an entity is selected in the *Group Inspector* on the bottom part of the screen we see various details about that entity. For example, for a class we see the position of the class in the class hierarchy, its methods and attributes, etc. The big advantage of the *Entity Browser* is that any reference to another design entity (e.g., the base class of the selected class) is a *hyperlink* to that entity. Thus, by clicking it the details of that entity will be displayed in a new tab of the *Entity Browser*. Additionally, for operations, INSIDER allows us to get quick access to the actual source code of the operation. The code appears on demand in a separate window, namely the *Source Code Viewer* (several such windows can be open at the same time).

B.3 Industrial Validation

Although *iPlasma* was developed as a research tool, it is not a toy. It was successfully used for analyzing the design of more than ten real-world, industrial systems including very large open-source systems (>1 MLOC), like **Mozilla** (C++, 2.56 million LOC) and **Eclipse**, (Java, 1.36 million LOC). *iPlasma* was also used during several consultancy

activities for industrial partners, most of them involved in developing large software applications for telecom systems.

B.4 Tool Information

The implementation of *iPlasma* was started in 1998 by Radu Marinescu in the context of the European FAMOOS ESPRIT Project. Over the years *iPlasma* has gradually evolved from a set of individual tools to an integrated suite of tools. It has been used for various industrial consultancy projects since 2002.

Tool Availability

iPlasma is implemented in Java — excepting the model extractor for C++, i.e.,MCC — and it is free to use. It was successfully run on all major platforms (Windows, Linux, Mac OS) and is freely available for download. Currently more information about *iPlasma* , including the possibility of downloading it, can be found at:

http://loose.upt.ro/iplasma/

C

CodeCrawler

C.1 Introduction

CodeCrawler is a software and information visualization tool [SDBP98, War00] which implements the *polymetric views* [Lan03b, LD03] presented in Chapter 2.

CodeCrawler relies on the FAMIX meta-model [DTD01] which models object-oriented languages such as C++, Java, Smalltalk, but also procedural languages like COBOL. FAMIX has been implemented in the MOOSE reengineering environment that offers a wide range of functionalities like metrics, query engines, navigation, etc. [DGLD05].

In this appendix we highlight some of the implementation characteristics.

C.2 CodeCrawler at Work

One aspect of the visualizations used in this book that cannot be shown is their intrinsic interactivity. There is no such thing as a *perfect* visualization and the reader should therefore keep in mind that the visualizations presented here were obtained after interacting with them. There are many publications [Lan03a, Lan03b, LD03, LD05, DL05, LDGP05] on the *CodeCrawler* tool and we prefer to direct the reader to those rather than trying to cover all the aspects of *CodeCrawler* here. We limit ourselves to illustrating a small example.

Example. An example of interactivity (see Fig. C.1) is that when the user passes the mouse pointer over a node or an edge, the item is highlighted and information about that subject item is displayed in

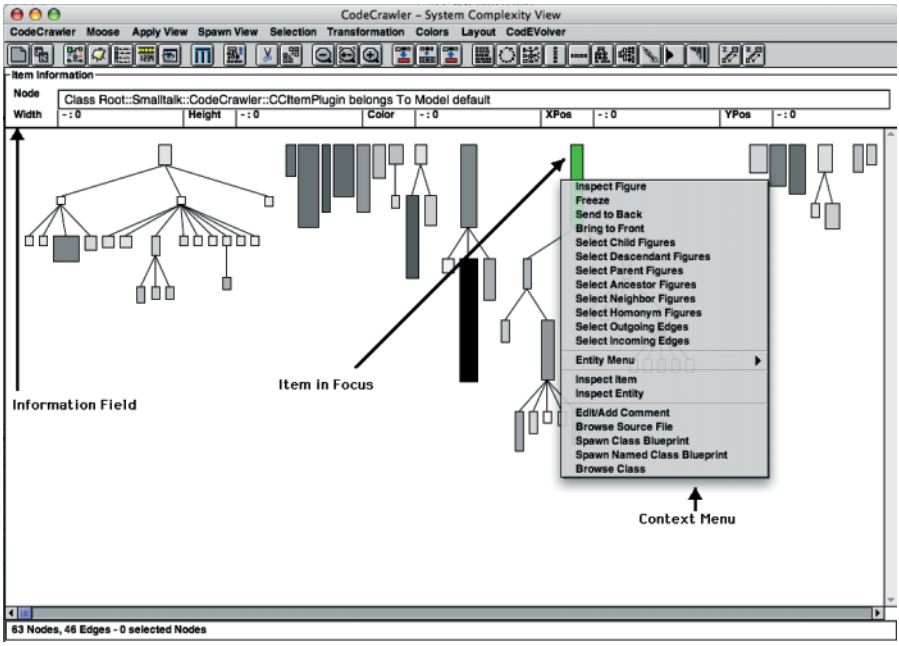


Fig. C.1. *CodeCrawler* at work. Every visible item can be interacted with in its own customized way.

CodeCrawler 's top input field, e.g., metric values and other semantic information, whether the class is abstract, etc. Moreover, using a context menu the viewer can interact with the item in focus.

The *polymetric views* (implemented in the *CodeCrawler* tool) can be created either programmatically by constructing the view objects, or using a View Editor, where each view can be composed using drag and drop. In Fig. C.2 we see *CodeCrawler* 's View Editor with the specification of the *System Complexity* view: the user can freely compose and specify the types of items that will be displayed in a view and also define the way the visualization will be performed. The user can choose among various types of nodes (class, method, package, etc.) and edges (inheritance, invocation, containment, etc.). For every node and edge the user can choose the figure type and assign to the figures the metrics to be used; there are several dozens of metrics that can be used, but as we will see in the remainder of this book only certain metrics make sense for certain *polymetric views*. The user can also choose the layout that he or she wants to use. In the case of Fig. 3.8

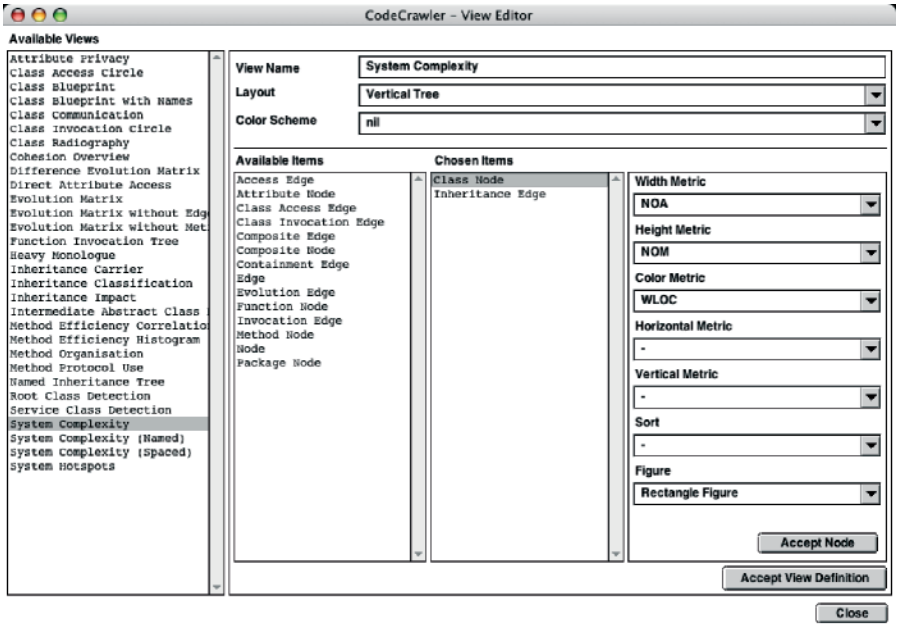


Fig. C.2. *CodeCrawler* 's View Editor.

where class nodes and inheritance edges have been chosen, a simple tree layout has been used.

C.3 Industrial Validation

CodeCrawler has been used several times to reverse-engineer industrial systems and is also used in conjunction with MOOSE by consultants to assess software systems. Due to non-disclosure agreements with our industrial partners we cannot provide detailed descriptions of our experiences, but limit ourselves to providing a list of case studies (industrial and non-industrial) that we have performed.
In Tab. C.1 we see that the systems were written in different programming languages and have sizes quite different from each other. The point common to all the case studies was the narrow time constraints imposed on us: we never had more than one week to reverse-engineer the systems.

Case Study	Language	Lines	Classes	Duration
Z (Network Switching)	C++	1,200,000	> 2300	1 Week
Y (Network Switching)	C++/Java	140,000	>400	1 Week
X (Multimedia)	Smalltalk	600,000	>2500	3 Days
W (Payroll)	COBOL	40,000	-	3 Days
SORTIE (Forest Management)	C++	28,000	70	2 Days
Duploc (Research Prototype)	Smalltalk	32,000	> 230	2 Days
Jun (3D Framework)	Smalltalk	135,000	> 700	3 Days
JBoss	Java	300,000	> 4500	1 Week
V (Logistics)	C++	120,000	> 300	2 Days
ArgoUML (Modeling)	Java	220,000	> 1300	4 Days
U (Telecom Services)	C++	2,500,000	> 10,000	1 Week

Table C.1. A list of some of the case studies CodeCrawler was applied on.

C.4 Tool Information

CodeCrawler's implementation commenced in 1998 as part of Michele Lanza's Masters and PhD work, in the context of the European FAMOOS ESPRIT Project. It has been used for various industrial consultancy projects since its first implementation. In its latest implementation it has become a general information visualization tool (e.g., visualization of concept lattices [Aré03] and websites) and also supports 3D visualizations [Wys05]. *CodeCrawler* uses the HotDraw framework for the 2D visual output and the Jun framework for the 3D visual output. It uses the MOOSE reengineering environment for the data input.

Tool Availability

CodeCrawler is implemented in Smalltalk under the BSD license: it is free and open-source software. It runs on every major platform (Windows, Mac OS, Linux, Unix) and is freely available for download. Currently the webpage is located at:

http://www.iam.unibe.ch/∼scg/Research/CodeCrawler/

Moreover, *CodeCrawler* is also available as a free goodie on the Visual-Works Smalltalk CD, a professional, commercial development environment developed and sold by the company Cincom, which, however, also exists in a non-commercial version freely available for download at:

http://www.cincomsmalltalk.com/

D

Figures in Color

Figure 1.1 *(on page 6)*

Object-Oriented Metrics in Practice — in a Nutshell.

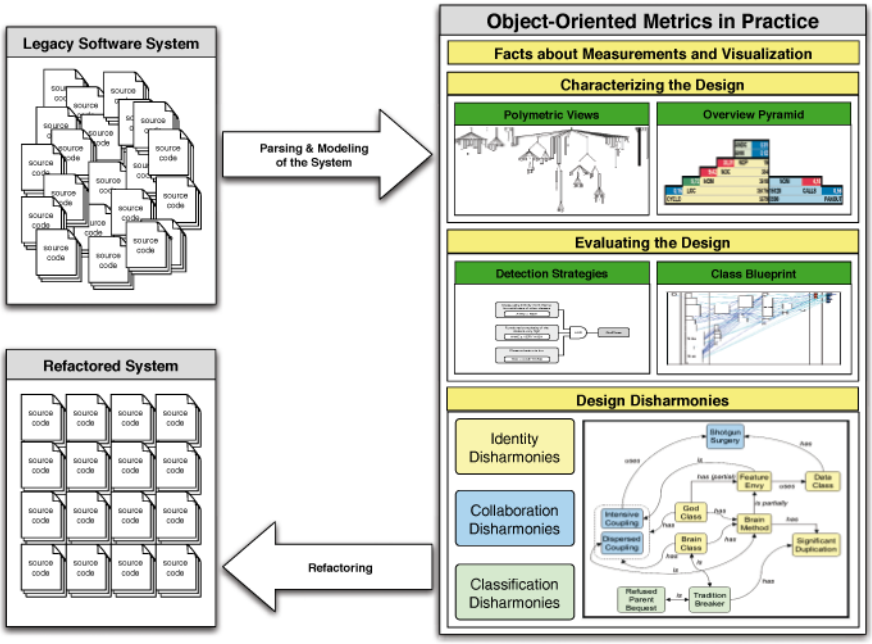


Figure 3.11 *(on page 41)*

The *Overview Pyramid* applied to *ArgoUML* .

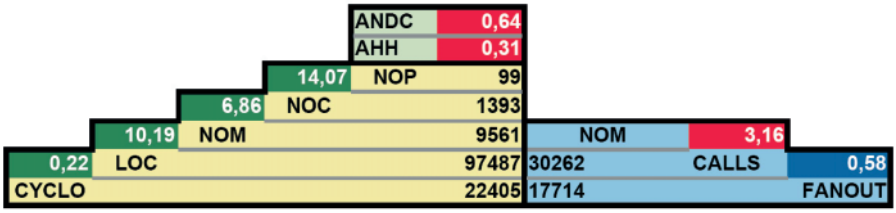


Figure 5.5 *(on page 86)*

ClassDiagramLayouter is envying the features of ClassDiagramNode. In red we colored the invocations that weightAndPlaceClasses performs towards ClassDiagramNode, while in green we see its class-internal invocations and accesses.

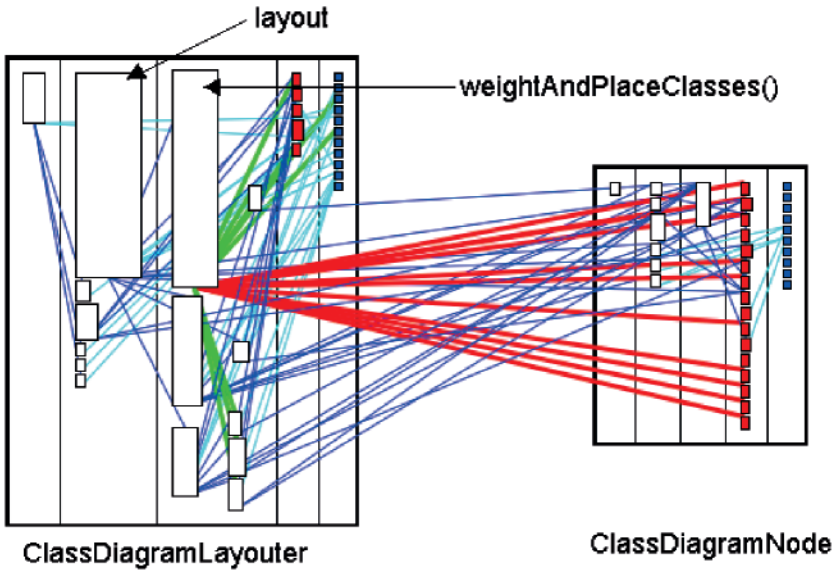


Figure 5.8 *(on page 90)*

An example of a Data Class: Property

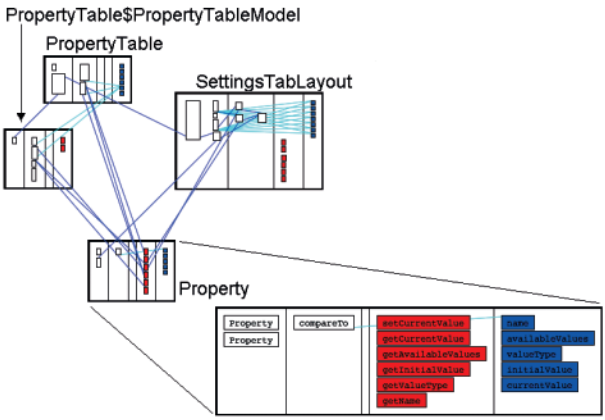


Figure 5.10 *(on page 94)*

A *Class Blueprint* of Modeller and ProjectBrowser.

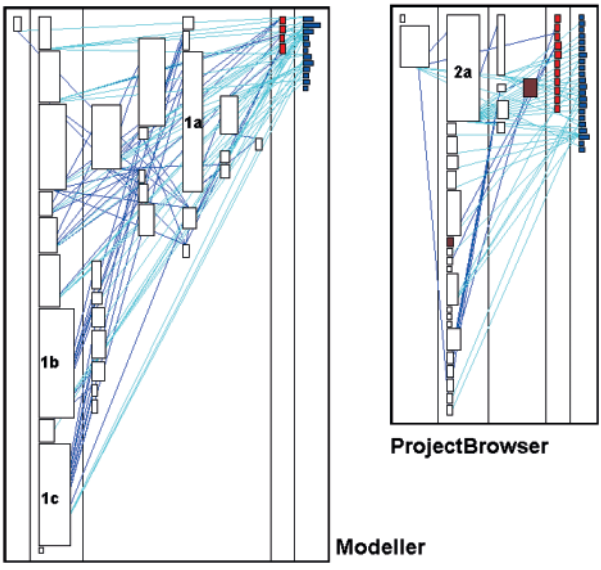


Figure 5.13 *(on page 100)*

A *Class Blueprint* of ParserDisplay with its completely abstract superclass Parser and a *Class Blueprint* of FigClass.

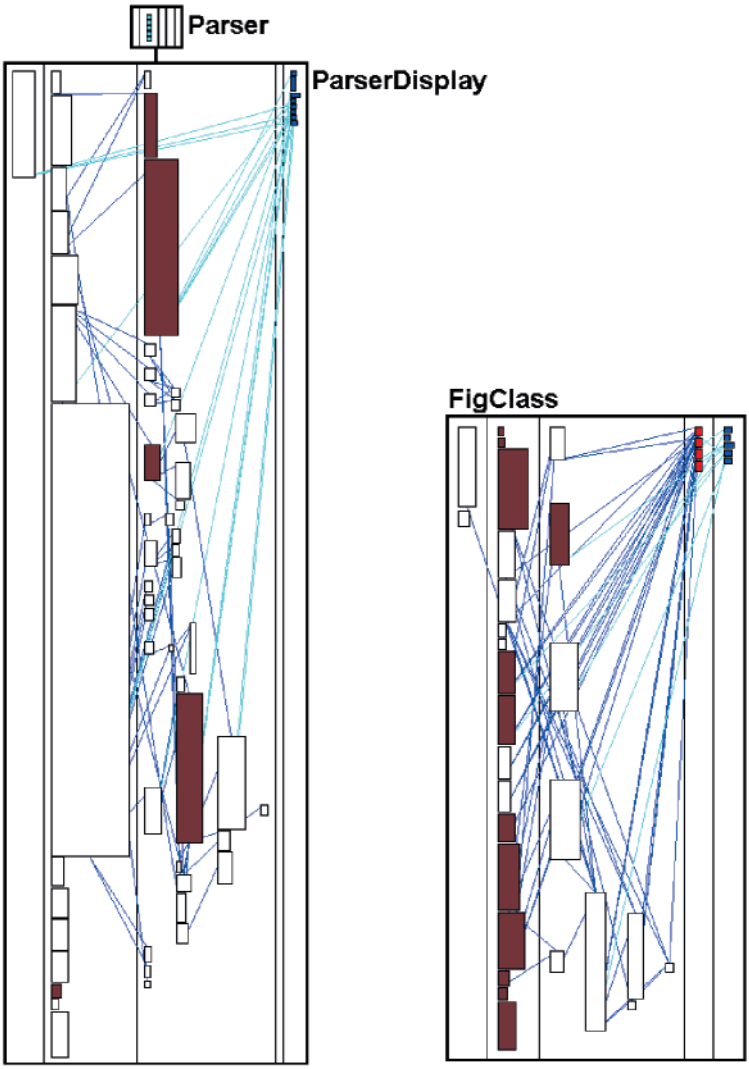


Figure 6.5 *(on page 124)*

The class ClassDiagramLayouter is intensively coupled especially with ClassDiagramNode. The red classes are non-model classes, i.e., belong to the Java library. The classes have been laid out according to the invocation sequence: above ClassDiagramLayouter are all classes that use it, while below it are all classes whose methods get used.

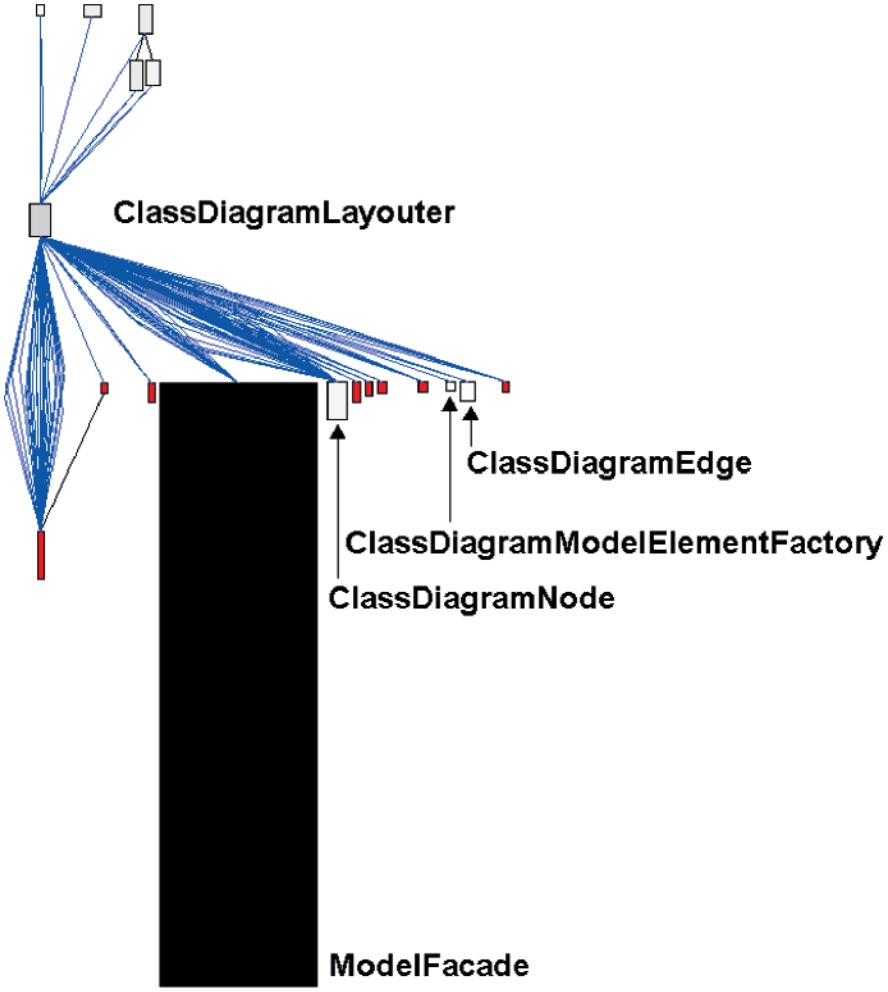


Figure 6.6 *(on page 125)*

The class ClassDiagramLayouter is intensively coupling with a few classes, especially ClassDiagramNode.

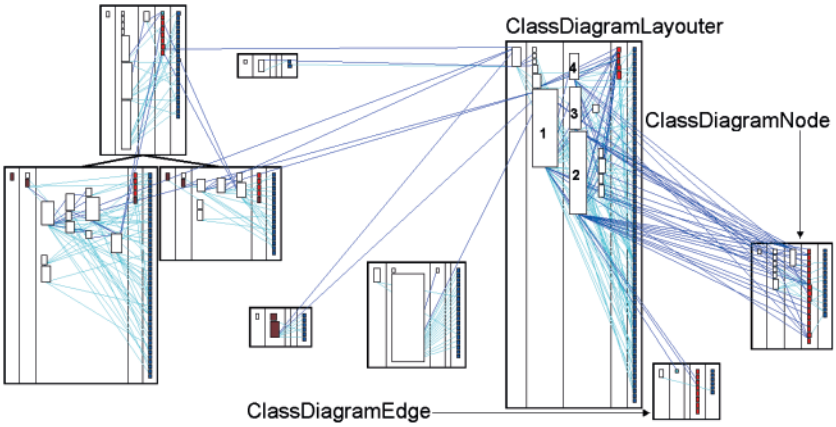


Figure 6.11 *(on page 129)*

The class ActionOpenProject is coupled with many classes. The red classes are non-model classes, i.e., belong to the Java library. The blue edges represent invocations.

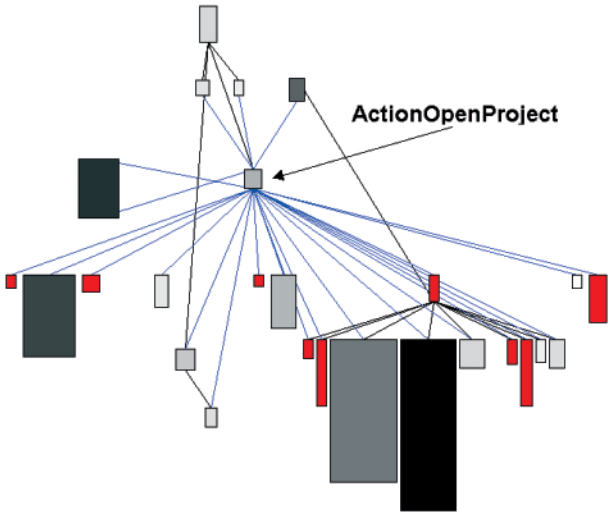


Figure 6.12 *(on page 130)*

The class Modeller is coupled with many classes and suffers itself from many other problems.

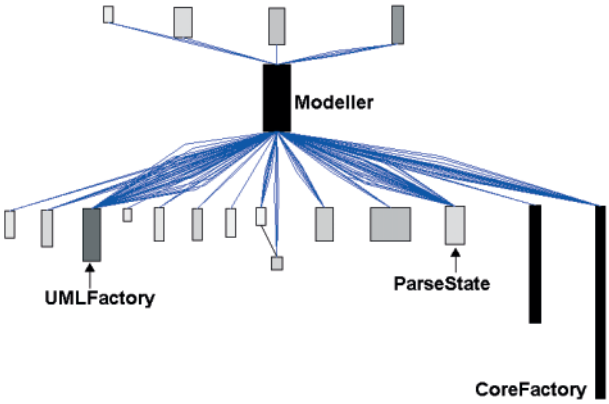


Figure 6.15 *(on page 135)*

Project provides an impressive example of a class with several methods affected by Shotgun Surgery(133). Due to these methods, Project is coupled with 131 classes (ModelFacade has been elided from the screenshot). Furthermore, the class has cyclic invocation dependencies with ProjectBrowser and CoreFactory. In the figure, the classes above Project depend on it, while Project itself depends on (i.e., invokes methods of) the classes below it.

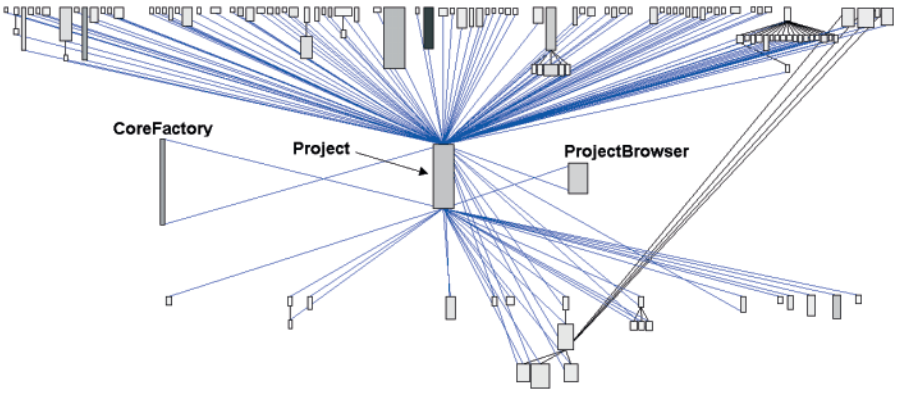


Figure 7.1 *(on page 143)*

Correlation web of classification disharmonies.

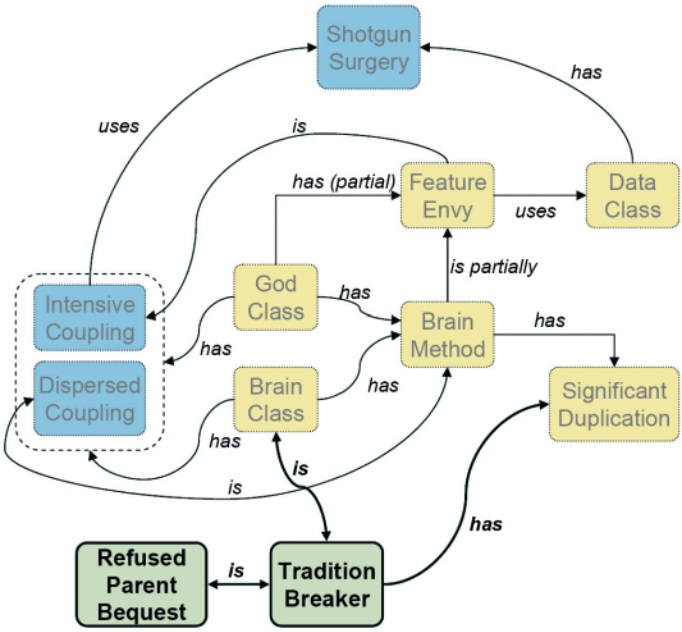


Figure 7.5 *(on page 148)*

A *Class Blueprint* view of the PerspectiveSupport hierarchy.

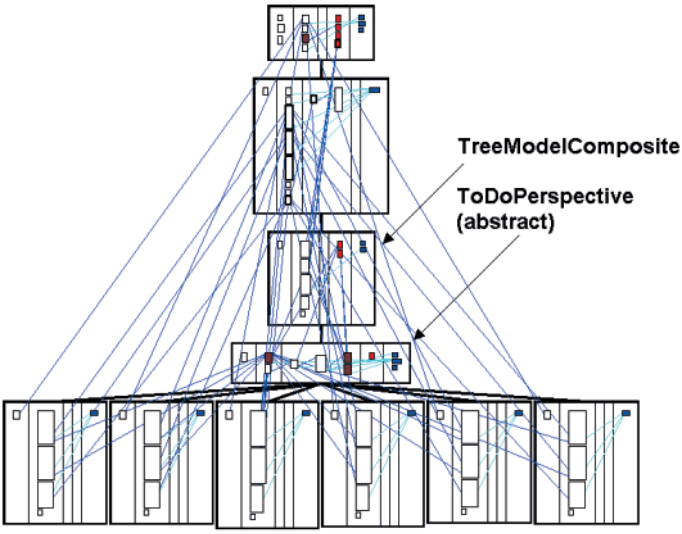
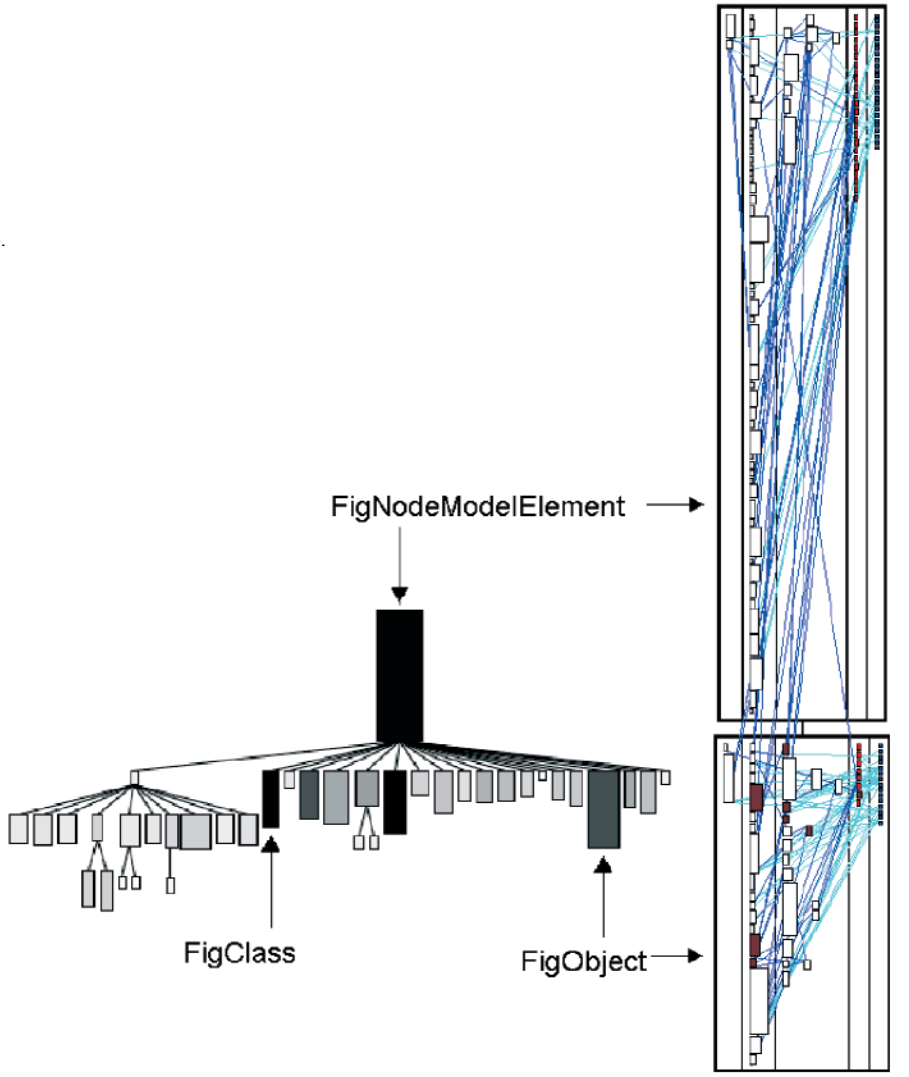


Figure 7.10 *(on page 156)*

A *System Complexity* view of the FigNodeModelElement hierarchy.



References

ABW98. Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

Aré03. Gabriela Arévalo. X-Ray views on a class using concept analysis. In *Proceedings of WOOR 2003 (4th International Workshop on Object-Oriented Reengineering)*, pages 76–80. University of Antwerp, July 2003.

BDW98. Lionel C. Briand, John Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(2), 1998.

BDW99. Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

Bec97. Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

Bec00. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

Ber74. Jacques Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

Bin99. Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

BK95. J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, April 1995.

BMMM98. William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

Boe88. Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

BR88. V. Basili and D. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

BS97. David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison Wesley, 1997.

CK94. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

Coo99. Alan Cooper. *The Inmates are running the Asylum*. SAMS, 1999.

CY91. Peter Coad and Edward Yourdon. *Object Oriented Design*. Prentice-Hall, 1991.

DDN02. Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

Dek02. Uri Dekel. Applications of concept lattices to code inspection and review. Technical report, Department of Computer Science, Technion, 2002.

DGLD05. Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

DL05. Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, January 2005.

DRW00. Alastair Dunsmore, Marc Roper, and Murray Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

DTD01. Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

Fav01. Jean-Marie Favre. Gsee: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 233–244. IEEE, May 2001.

FBB+99. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

Fea05. Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.

FP96. Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

GLD05. Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

HS96. Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

JF88. Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

Ker04. Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2004.

Kru04. Philippe Kruchten. *The Rational Unified Process*. Addison-Wesley, third edition, 2004.

Lak96. John Lakos. *Large Scale C++ Software Design*. Addison Wesley, 1996.

Lan03a. Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

Lan03b. Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

LB85. Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

LD01. Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

LD03. Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

LD05. Michele Lanza and Stéphane Ducasse. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, Milano, 2005.

LDGP05. Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM Press, 2005.

LH89. K. Lieberherr and I. Holland. Assuring a good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

Lie96. Karl J. Lieberherr. *Adaptative Object-Oriented Software: The Demeter Method*. PWS Publishing, 1996.

Lis87. Barbara Liskov. Data Abstraction and Hierarchy. In *Proceedings OOPSLA '87*, page addendum, December 1987.

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

LN95. Danny B. Lange and Yuichi Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 342–357. ACM Press, 1995.

LP90. Wilf LaLonde and John Pugh. *Inside Smalltalk: Volume 1*. Prentice Hall, 1990.

LPLS96. David Littman, Jeannine Pinto, Stan Letovsky, and Elliot Soloway. Mental Models and Software Maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98. Ablex Publishing Corporation, 1996.

LR89. Karl J. Lieberherr and Arthur J. Riel. Contributions to teaching object oriented design and programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 11–22, October 1989.

LW93a. Barbara Liskov and Jeannette M. Wing. Family values: A behavioral notion of subtyping. CMU-CS-93-187, Carnegie Mellon University, July 1993.

LW93b. Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag.

MADSM01. C. Best M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

Mar02a. Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timişoara, 2002.

Mar02b. Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

McC76. T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

Mey88a. Bertrand Meyer. Disciplined exceptions. Tr-ei-22/ex, Interactive Software Engineering, Goleta, CA, 1988.

Mey88b. Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

MGM02. Ludger Martin, Anke Giesl, and Johannes Martin. Dynamic component program visualization. In *Proceedings of WCRE 2002 (Working Conference on Reverse Engineering)*, 2002.

Pin97. Steven Pinker. *How the Mind Works*. W. W. Norton, 1997.

RDGM04. Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 223–232, Los Alamitos CA, 2004. IEEE Computer Society.

Rie96. Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

SDBP98. John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

Sto98. Margaret-Anne D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, December 1998.

SWM97. Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

Tuf90. Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

Tuf01. Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

vMV96. A. von Mayrhauser and A.M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, June 1996.

War00. Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.

WBM03. Rebecca Wirfs-Brock and Alan McKean. *Object Design — Roles, Responsibilities and Collaborations*. Addison-Wesley, 2003.

WBW89. Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings OOPSLA '89*, pages 71–76, October 1989. ACM SIGPLAN Notices, volume 24, number 10.

WH92. Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

Wys05. Christoph Wysseier. Interactive 3-D visualization of feature-traces. MSc. thesis, University of Berne, Switzerland, November 2005.

Index