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