Assessing and Improving Object-Oriented Design

Habilitation Thesis

dr. Radu Marinescu

May 2012

To Cristina and Mitzu

Acknowledgements

It is so unfair that this habilitation thesis must have a single author...because I owe so much to many people who have such a huge contribution to all my achievements.

Words are too poor to express my gratitude to George Ganea and Ioana Verebi. George, Ioana, I am absolutely sure that without having you close to me almost every day for the last five years *none* of my achievements would have existed. Thank you so much for sacrificing your time and energy just for the sake of us working together. Thank you for keeping alive my dream of having you as my first Ph.D. students, even through moments all hope seemed to vanish. Thank you for sharing my dreams, and for patiently listening to my endless talks about various crazy things. I just wish that our work together and friendship would never end.

I am very grateful for having the chance of meeting Marius Minea. Marius, it's such an amazing experience to plan, work, fight, laugh and cry together with you. Thank you for the infinite ways in which you supported and encouraged me over the last ten years. Your friendship is one of the most wonderful gifts I received so far.

Doru Gîrba is the living proof that physical distance can be bridged by a warm friendship and collaboration. Doru, when looking back at the years since we first met, I can't believe how amazing this experience has been. I am so deeply indebted to you for permanently stirring me up with your crazy dreams and wild ideas, and for being there for me every time when I was in need. I still hope that one day our friendship will become so powerful to change the laws of physics (or life?) and eliminate the kilometers that separate us.

I owe a lot of gratitude to Adrian Trifu for challenging me to start the Intooitus spin-off, which is certainly one of the most amazing experiences I've had so far. Adi, thank you for inviting me on this fabulous journey, thank you for the never-ending discussions that sometimes sharpened my arguments, and most of the times help me understand that I was simply wrong. Thank you for helping me look at things from a completely different perspective, and forgive me that I am often too stubborn and proud to admit that I need another point of view.

Working with the members of the LOOSE Research Group (LRG) in Timișoara is such an extraordinary experience. I want to thank Petru Mihancea, Dan Cosma, Mihai Balint and Richard Wettel for our fruitful work on various projects and for the days and nights that we wrote papers together. I so much hope that we will re-intensify this experience over the coming months and years. I am also deeply indebted to Daniel Rațiu, Mircea Trifu and for all the amazing years we spent together in the early days of LOOSE, but also for our more recent collaborations. After all these years I want to thank you one more time for your warm friendship and enthusiasm.

I also want to thank several diploma and master students for their exceptional work that contributed to some of my past achievements and/or currently ongoing projects, namely: Andrei Chiș, George Pucea, Caius Brindescu, Mihai Codoban, Adi Dozsa and Domi Tamaș-Selicean.

I would like to warmly thank my parents for all their love, for all their mental and financial support and for believing in me even when I didn't. I am so happy that you lived to see me reach this stage. Dad, you were right: balance and hard work are more important than being a genius; and, yes, one doesn't have to be a genius to achieve great things!

Cristina, how can I thank you for all the love, joy, balance and wisdom that you pour into me day by day? Without your constant encouragement I would have never finished this work, and without your love it wouldn't have been worth doing it. Mitzu, I thank you for all the times you phoned to my office to call me (urgently!) home so that we can play together, and for constantly asking me if I finished writing. I am so proud to finally tell you: I am done! Cristina and Mitzu, you are my most precious gift.

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

Timişoara,
April 30, 2012

Radu Marinescu

Summary

In a society characterized by frequent changes, software must evolve at the same pace. To be able to evolve and adapt to new requirements, software has to be prepared for changes, which in return require high design and implementation quality.

This habilitation thesis is the summary of the research that we performed during the last ten years on the assessment and improvement of design quality in object-oriented systems. This research direction has grown significantly over the last years due to the exponential increase of large-scale object-oriented systems. In such systems integration or bug fixing become so unpredictable that it becomes more cost-effective to rewrite the system. However, the cause of such situations is less visible: the internal quality of the system's design is declining; and duplicated code, overly complex methods or non-cohesive classes are just a few signs of this decline. These, and many others, are usually the symptoms of higher-level design problems, which are usually known as *design flaws*.

In software engineering, measurement is essential, as otherwise we risk losing control due to excessive complexity. Consequently, software metrics are the foundation of our research. Thus, we start the thesis by describing our approach to *defining metrics* in a way that is both accurate and easy to understand, and to establishing meaningful *metrics thresholds*. In the context of metrics we also introduce the *Overview Pyramid*, an integrated, metrics-based visualization aimed to characterize the complexity, coupling and inheritance usage in object-oriented system.

While metrics are needed for the assessment of software design, we argue that isolated metrics cannot serve this goal in a way that leads to improvement actions. Going from abnormal numbers to the recognition of design flaws is impossible because the interpretation of individual metrics is too fine grained to indicate real design flaws. To support the detection and location of design problems, we introduce the *detection strategy* technique for creating metrics-based rules that capture deviations from principles and heuristics of good design. Consequently, engineers can directly spot flawed design fragments rather than inferring the flaws from a large set of abnormal metric values. We have defined such rules for around 20 important flaws related to object-oriented design. In the thesis we also discuss various approaches for extending the set of detectable design flaws and for refining the accuracy of these detection techniques.

Detection strategies are a major step forward, but assessing the design of a system requires more than just a list of design flaw instances. Quality models are needed to get an overall assessment of design quality. We present our two approaches to quality models, both based on the idea of using detected design flaws as first-class entities in quality models. We describe the *Factor-Strategy* model which improves over traditional approaches by facilitating their *construction* and the *interpretation* as quality is related to violations of concrete design rules. More recent, we developed the idea further and built an assessment framework that exposes and quantifies symptoms of *design debt*. This framework contributes to increasing the visibility of design flaws that result from debt-incurring design changes.

Industrial systems are extremely large, and therefore the automation of assessment approaches is crucial. Therefore, we need to build scalable analysis tools. In this context, we present IPLASMA, which is an extensible integrated assessment framework that can be used for periodic code-reviews. We also introduce the INCODE plugin which is an instantiation of our vision to make assessment a *continuous* process, in which detecting, understanding and removing design problems will become part of the day to day activity of developers.

Assessing the design quality of software is certainly important; however, the ultimate goal is to *improve* the quality of the assessed system. Therefore, our main research focus has recently shifted towards bridging the gap between the detection and correction of design problems, by creating tool-supported techniques for describing and executing contextual, problem-driven *correction strategies*. Thus, we close this thesis by presenting our first research results in this new direction.

Our research so far has had a significant impact both in the international academic community, as well as in the software industry. Our publications have received over 250 citations in major software engineering journals and conferences, including 13 citations in the *IEEE Transactions on Software Engineering*, and 15 citations in the *IEEE International Conference on Software Engineering*. The quality assessment software industry has also acknowledged the relevance of these research achievements, by adopting our techniques in some of their CASE tools. In 2009 we received the *IBM John Backus Award*, offered by IBM in a worldwide competition to two mid-level career scientists, by a jury who included two *Turing Award* recipients. The award was granted for "*having done the most to improve programmer productivity*".

For the future, we plan to grow our research on several key directions. First we will continue the work on better connecting assessment and correction activities. We also plan to investigate how quality assessment can be efficiently applied to *hybrid software systems* which mix different programming languages or even paradigms. Another plan is to conduct a broad empirical validation of assessment techniques, by creating a comprehensive metrics benchmarking methodology, operationalized by proper tools. We aim to perform this validation on at least 10.000 projects. The analysis results will be used to calibrate the quality assessment techniques and to detect potential inconsistencies in quality models.

Summary (limba română)

Într-o societate caracterizată prin schimbări frecvente, sistemele software trebuie să evolueze în același ritm. Pentru a putea să evolueze, adaptându-se noilor cerințe, sistemele software trebuie să fie pregătite pentru schimbare, ceea ce implică o calitate foarte bună a proiectării și a implementării.

Această teză de abilitare reprezintă sinteza cercetării pe care am efectuat-o pe parcursul ultimilor zece ani în domeniul evaluării și îmbunătățirii calității proiectării în sisteme orientate-pe-obiecte. În ultimii ani, acestă direcție de cercetare a căpătat tot mai multă importanță odată cu creșterea exponențială a numărului de sisteme complexe, orientate-pe-obiecte. În astfel de sisteme integrarea sau depanarea tind să devină atât de impredictibile încât adesea devine mai eficientă rescrierea sistemului. Cauza unor astfel de situații este însă ascunsă: calitatea internă a proiectării se degradează, iar codul duplicat, metodele excesiv de complexe sau clasele necoezive sunt doar câteva dintre semnele acestui declin. Acestea, și multe altele, sunt adesea simptomele unor probleme de proiectare de nivel mai înalt, cunoscute sub numele de *carențe de proiectare*.

În ingineria software măsurarea este esențială întrucât altfel riscăm să pierdem controlul din cauza complexității excesive. În consecință, metricile software reprezintă fundamentul cercetării noastre. Așadar, această teză începe cu o descriere a abordărilor noastre legate de *definirea metricilor* într-un mod care să fie deopotrivă precis și ușor de înțeles, precum și cu abordarea legată de stabilirea unor *valori de prag* semnificative. În acest context al măsurării software-ului descriem și *Overview Pyramid*, o vizualizare integrată, bazată pe metrici, ce urmărește să caracterizeze în ansamblu un sistem orientat-pe-obiecte din perspectiva complexității, a cuplajului și a utilizării relației de moștenire.

Deși metricile sunt necesare pentru evaluarea proiectării, susținem că metricile folosite în izolare nu pot ajuta la evaluare într-un mod care să faciliteze corecția sistemului evaluat. Detectarea carențelor de proiectare pornind de la valori anormale de metrici este imposibil de realizat deoarece interpretarea metricilor individuale are o granularitate prea fină pentru a indica probleme reale de proiectare. Pentru a facilita detecția și localizarea problemelor de proiectare, am introdus tehnica *strategiilor de detecție* prin care se pot defini reguli bazate pe metrici ce descriu deviații de la principiile si regulile de bună proiectare. Astfel, inginerii pot detecta nemijlocit fragmente de cod afectate de carențe de proiectare în loc să fie nevoiți a le infera dintr-o mulțime de valori anormale de metrici. Am definit circa 20 de astfel de reguli referitoare la carențe de proiectare importante. În această teză sunt deasemenea discutate diferite abordări pentru extinderea setului de carențe detectabile și pentru rafinarea preciziei acestor tehnici de detecție.

Strategiile de detecție reprezintă un mare pas înainte, dar cu toate acestea evaluarea proiectării unui sistem necesită mai mult decât o simplă listă cu instanțe de carențe de proiectare detectate. Pentru o evaluare globală a calității proiectării sunt necesare modele de calitate. Vom prezenta cele două abordări proprii referitoare la modele de calitate, ambele centrate în jurul ideii de a folosi carențele de proiectare detectate ca entități primare ale acestor modele. Vom descrie modelul *Factor-Strategy* care aduce îmbunătățiri față de abordările tradiționale simplificând *construcția* și *interpretarea* modelelor, prin relaționarea calității direct cu încălcări ale unor reguli concrete de proiectare. Recent, am dezvoltat și mai mult această abordare și am construit un cadru conceptual prin care pot fi expuse și cuantificate simptome ale *restanțelor de proiectare*. Acest cadru conceptual contribuie la expunerea mai pronunțată a carențelor de proiectare rezultate din modificări nepotrivite ale sistemu-

lui.

Sistemele software industriale sunt extrem de mari și de aceea automatizarea tehnicilor de evaluare este esențială. De aceea este necesar să construim instrumente de analiză scalabile. În acest context, vom prezenta IPLASMA, un instrument integrat și extensibil de analiză, ce poate fi folosit pentru evaluări periodice ale codului. De asemenea, vom prezenta și INCODE un plugin reprezentativ pentru viziunea noastră de a transforma evaluarea calității într-un proces *continuu*, în care detecția, înțelegerea și corecția problemelor de proiectare să devină parte a activității cotidiene de programare.

Partea de evaluare a calității proiectării este cu siguranță importantă, dar scopul ultim este acela de a *îmbunătăți* calitatea sistemului evaluat. De aceea, cercetările noastre mai recente s-au refocalizat înspre conectarea fazelor de detecție și de corecție a carențelor de proiectare. Astfel, am început să dezvoltăm technici automatizate de descriere și execuție a unor *strategii de corecție* contextualizate și focalizate. Prin urmare, teza va fi concluzionată de prezentarea primelor rezultate obținute în această nouă direcție de cercetare.

Rezultatele de până acum ale cercetării noastre au avut un impact semnificativ atât în comunitatea academică internațională, precum și în rândurile industriei software. Publicațiile noastre au strâns mai mult de 250 citări în reviste și conferințe internaționale de prim rang în ingineria software, inclusiv 13 citări în *IEEE Transactions on Software Engineering* și alte 15 citări în *IEEE International Conference on Software Engineering*. Industria producătoare de instrumente de asigurare a calității software-ului a confirmat de asemenea relevanța rezultatelor obținute, adoptând unele din tehnicile pe care le-am dezvoltat. În 2009 am primit premiul *IBM John Backus*, oferit de către IBM într-o competiție mondială pentru doi cercetători care "*au făcut cel mai mult pentru îmbunătățirea productivității programatorilor*", din juriu făcând parte și doi câștigători ai premiului *Turing*.

Pe viitor ne propunem să dezvoltăm această cercetare în câteva direcții cheie. În primul rând vom continua să lucrăm în direcția unei punți mai solide între activitățile de evaluare și de corecție. Ne propunem de asemenea să investigăm cum tehnicile dezvoltate pot fi aplicate la *sisteme software hibride* ce mixează diferite limbaje sau chiar paradigme de programare. Un alt plan este acela de a pune la punct o amplă validare empirică a tehnicilor de evaluare, definind o metodologie complexă de analiză a valorilor metricilor, ce va fi operaționalizată prin unelte adecvate. Ne dorim să realizăm acestă validare pe o bază de cel puțin 10.000 de proiecte. Rezultatele acestei analize vor fi utilizate pentru a calibra metricile folosite in tehnicile de evaluare a calității și pentru a detecta potențiale incosistențe în modelele de calitate.

Contents

Part I

Achievements

Chapter 1

Introduction

This habilitation thesis is a summary of the research we performed in the area of quality assessment and improvement of object-oriented systems, over a period of almost ten years. The habilitation covers the research performed since the defense of the PhD thesis, in December 2002 and until this habilitation was printed. In fact we started this research 15 years ago, with a diploma thesis on software metrics. This chapter presents the context of this work in a nutshell, summarizes our contributions and proposes a roadmap of the document for the reader.

1.1 Context and Motivation

In an information technology society that is increasingly relying on software, software productivity and quality continue to fall short of expectations: software systems suffer from signs of aging [Par94] as they are adapted to changing requirements. The main reason for this problem is that activities of software maintenance and evolution are still undervalued in traditional software development processes. The only way to overcome or avoid the negative effects of aging in legacy software systems and to facilitate their smooth evolution is by providing engineers with a fully automated and integrated support for the entire process of software evolution.

The issue of object-oriented software evolution is an important matter in today's software industry, and it will definitely continue being a vital matter in tomorrow's software industry. The law of software entropy [LPR$^+$97] dictates that even when a system starts off in a well-designed state, requirements evolve and customers demand new functionality, frequently in ways the original design did not anticipate. A complete redesign may not be practical, and a system is bound to gradually lose its original clean structure and deform into an unmaintainable, rigid and hard to understand bowl of “object-oriented spaghetti” [WH, BMM98]. In order to postpone the software decay and at least partially save its technical and economical value a rigorous process to control the evolution of software is a must.

It is well known both that both in software engineering theory and also in practice, large-scale, complex applications that prove a poor design and implementation are very dangerous because of the delayed effect of these structural problems [DDN03]. The applications are going to run correctly a period of time, but their adaptation to new requirements is going to be unfeasible from the economical point of view. A late discovery of this problem can be very dangerous because rebuilding the application can be very expensive, while in the case of large enterprise applications it is virtually impossible [DM86]. This emphasizes the need to analyze the software from multiple points of view in order to detect on time the design and implementation problems that could inhibit or make very expensive the evolution of the system.

Like all human activities, the process of designing software is error prone and object-oriented design makes no exception. The flaws of the design structure have a strong negative impact on quality attributes such as flexibility or maintainability [FBB+99]. Thus, the identification and detection of these design problems is essential for the evaluation and improvement of software quality. The fact that the software industry is nowadays confronted with a big number of large-scale legacy systems written in an object-oriented language, which are monolithic, inflexible and hard to maintain shows that just knowing the syntax elements of an object-oriented language or the concepts involved in the object-oriented technology is far from being sufficient to produce good software. A good object-oriented design needs design rules and heuristics [JF88, Rie96] that must be known and used.

1.2 Research Roadmap

In this context, our research has been focused on the quality assessment and improvement of object-oriented systems. Figure 1.1 presents the main parts of our this research field and their relations. In fact, these are the phases of the general and well-established *reengineering and evolution life-cycle* [DDN03].

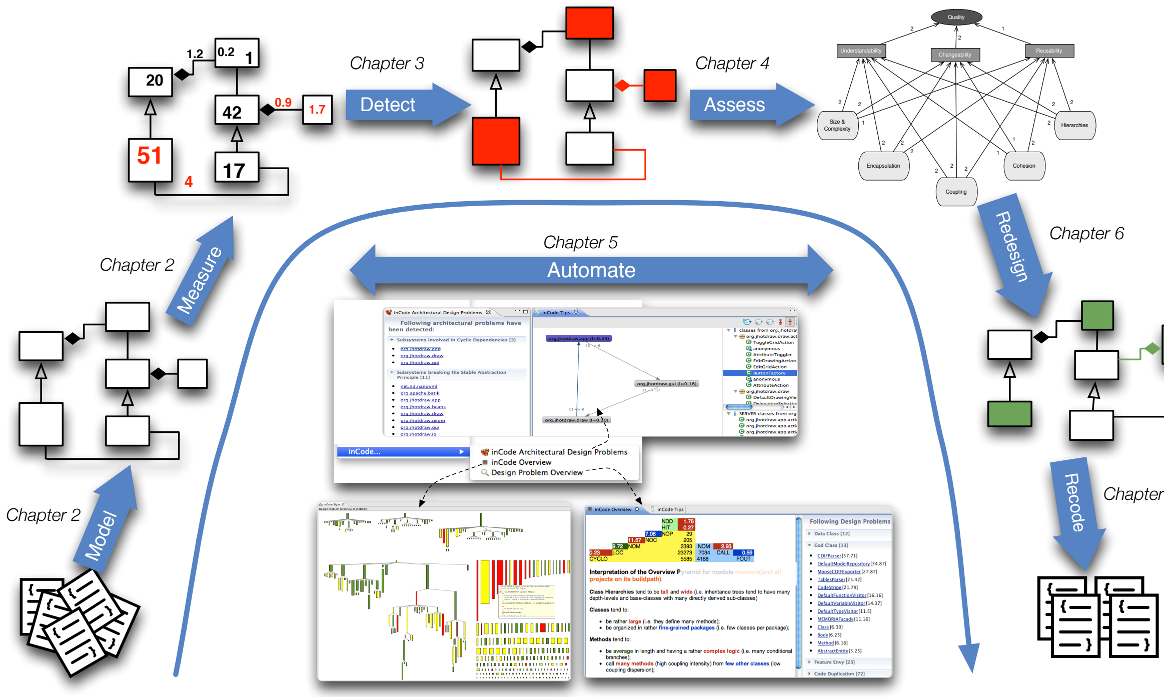


Figure 1.1: Roadmap of our research

This process starts from the source code of a *legacy system i.e.*, a software system that (i) one has *inherited* and which (ii) is *valuable*. Legacy systems usually show signs of *aging software* [Par94] *i.e.*, non-available original developers, lack of documentation, monolithic design, code bloat *etc.*. On of our main assumptions is that the only source of *reliable* information is the code itself. Documentation is most of the time inexistent [Par94], and sometimes even comments are missing or obsolete.

Model the Software The process of quality assessment and improvement starts with *abstracting* from the source and creating simplified representations of the code elements *i.e.*, extracting various design models, based on which further measurements and other assessment analyses can be performed. We present our contributions regarding modeling in Chapter 2, Section 2.2.

Measure the Design In order to assess and control quality one needs proper quantification means [DM86]. This is why the cornerstone of our research lies in software metrics. After introducing our modeling approach, the rest of Chapter 2 presents our contribution to two major problems in software measurement, namely: (i) finding means for an accurate and understandable definition of metrics (Section 2.3) and (ii) establishing meaningful metrics thresholds (Section 2.4). Last, but not least we introduce the *Overview Pyramid* (Section 2.5), which is an integrated, metrics-based visualization aimed to characterize the complexity, coupling and inheritance usage in object-oriented system.

Detect Design Flaws While it is clear that metrics should be employed for the evaluation of software design, the main question is: what exactly should we measure? For design quality assessment, we need to quantify how well a given design complies with a set of design principles, rules and heuristics. Chapter 3 introduces our novel approach of *detection strategies* (Section 3.2) that allows to quantify specific violations of such principles and rules. We illustrate the principle of *detection strategies* and show various techniques for enhancing and refining them by using historical information (Section 3.3). After describing a new technique for detecting fragmented code duplication (Section 3.4), the chapter is closed by presenting our approach for specifying and verifying project-specific (architectural) rules (Section 3.5).

Assess Design Quality Using *detection strategies* to raise the abstraction level in detecting design flaws is a significant step forward. However, in order to assess the design of a system one needs more than just a (long) list of design flaw instances. In this context, quality models are needed for getting an overall assessment of design quality. In Chapter 4 we present our two novel approaches to quality models, both based on the idea of building on using detected design flaws as first-class entities. First we discuss our earlier attempt, namely the *Factor-Strategy model* (Section 4.2). Then, in the second part of this chapter we present our very recent assessment framework that exposes and quantifies *symptoms of design debt*. The framework contributes to increasing the visibility of design flaws that result from debt-incurring design changes (Section 4.3).

Improve the Design Assessing the design quality of software is certainly important; however, the ultimate goal is to *improve* the quality of the assessed system. In Chapter 6 we close the circle and describe our most recent research on bridging the gap between the detection and correction of design flaws. There are two main contributions that we describe here: (i) the conceptual approach for describing higher-level *correction strategies* defined at the same abstraction level as *detection strategies* (Section 6.3); and (ii) the envisioned tool support for executing correction strategies (Section 6.4).

Automate the Assessment Legacy systems tend to be extremely large, up to 10-20 million lines of code, and therefore the scalability of the proposed approaches is crucial. Because of this we put a particular emphasis on validating our ideas by building scalable tools. In Chapter 5 we describe our two main approaches towards automating quality assessment: IPLASMA (Section 5.2) is a complex integrated assessment framework that can be used for standard assessments and code-reviews; on the other hand, more recently, we took a completely new approach on assessment tools by creating INCODE (Section 5.3) as an Eclipse plugin that transforms assessment into an agile, continuous process.

In order to provide a clear and coherent flow of our research, the chapters of this habilitation thesis follow the sequence described above (see Figure 1.1). Moreover, to ease the reading of this document, we follow a similar structure during every chapter, namely: (i) the *research problem*, together with a *condensed state of the art i.e.,* other approaches addressing the research problem, and then (ii) *our solutions to the problem*, emphasizing their novelty and scientific contribution.

1.3 Summary of Contributions and Impact

As already mentioned, our research has been focused on the assessment and restructuring of object-oriented software systems.

1.3.1 Major Contributions

In this research domain, we provided in the last 10 years a number of world-wide recognized contributions:

- *The detection strategy concept* [Mar04, Mar05, LM06] is a totally novel concept, which raises the level of abstraction in using software metrics. This approach made possible for the first time to define quantifiable design rules, so that these rules can be automatically checked on a certain project.
- *A methodology for detecting design flaws using metrics-based rules* [MM05, TM05, RDGM04]. As an application of the detection strategy concept, we proposed a suite of around 20 strategies which correspond to object-oriented design flaws that are intensely discussed in an informal manner in the literature. The major step forward consisted in making these design problems automatically detectable in large-scale software systems.
- *Adoption.* The detection strategy concept [Mar04, Mar05], and a suite of metrics defined earlier [Mar99] were integrated in one of *Borland*'s CASE tools. Furthermore, using the techniques he defined, we have provided consultancy for first ranking companies from Europe: Telelogic France, Sema Group, ABB, TogetherSoft, OCE Software, Alcatel, Siemens Automotive, in the fields of redesigning and quality assurance of software.
- *The Factor-Strategy quality model* [MR04]. Unlike the classic model (Factor-Criteria-Metric) – intensively used in different forms worldwide – this new model enables, for the first time, not only the assessment of the quality of a software system, but also the identification of the real causes that make a system to display a poor quality. This reduces considerably the necessarily amount of time spent on improving the quality of design.
- *The* IPLASMA *platform* [MMG05] that automates the quality assessment techniques for analyzing systems written in C++, Java or C#. Since 2004, IPLASMA has been used both as a consultancy tool, as well as a QA tool used directly by programmers in companies to perform code reviews on tens of projects. The tool has been used both in big corporations (Alcatel-Lucent, Continental, Siemens AG, Huawei) and local and European SMEs (Savatech, OCE Software, DeComp

Brussels etc). iPLASMA has been used to analyze extremely large-scale software for telecom (about 2 million lines of code) system belonging to Alcatel-Lucent (Timișoara). Furthermore, iPLASMA is currently in use in at least 5 European research groups (Software Composition Group at the University of Bern, University of Lugano, FZI Karlsruhe, TU Munich, University of Inssbruck).

- INCODE [MGV10] is the incarnation of a new way of performing quality assessment, in which design flaws are detected continuously as code is written. This increases the productivity of programmers in two ways: (i) by warning developers about the occurrence of a design problem as they appear, which is far more efficient than the classic code inspections; (ii) by providing contextualized explanations for each instance of a detected problem, instead of confronting programmers with just "dry" numbers (i.e., abnormal metrics values, which are hard to interpret) in a way that leads to real code/design improvements. The INCODE plugin was downloaded over more than 10.000 times (since May 2008) and is currently in daily use by programmers from more than 10 software companies in Europe.

1.3.2 Citations

Our publications on this topic have more than several hundreds of citations in the literature[1]. Many of these citations are in the top journals and conferences in the field; for example, our work has been cited 13 times in the *IEEE Transactions on Software Engineering*, 15 times in the *IEEE International Conference on Software Engineering*, and 35 times in the *IEEE International Conference on Software Maintenance*, which is the premier world-wide event in software maintenance.

Another evidence of the impact is the fact that the book *Object-Oriented Metrics in Practice* [LM06] has been sold so for in more than 1.000 copies, and it is among the top 5 most relevant hits on the "*object-oriented metrics*" search phrase, both on Google and Amazon.

1.3.3 Awards

In 2009 we received the *IBM John W. Backus Award*, which was offered by IBM in a worldwide competition to two mid-level career scientists who "*have done the most to improve programmer productivity, whether through languages, optimizations, tools, methodology or any other techniques*". The winners were decided by a jury that included *Turing Award winners* Frederic P. Brooks and Frances Allen, as well as well-known software engineering researchers like Grady Booch, Erich Gamma, David Harel.

In 2006, we received in a world-wide competition an *IBM Eclipse Innovation Award* for envisioning an enhancement of the Eclipse platform that would support programmers with a continuous detection of design problems means of detecting on-the-fly design and code problems (see Section 5.3).

On the educational side, we received in 2009 the *Apple Distinguished Educator*, and in 2007 the *Bologna Professor* awards.

[1]According to Scholar Google currently the number of citations is over 900, but this number does also include auto-citations. The number of citations in major journal and conferences is over 250, excluding auto-citations.

1.3.4 Community Service

As a result of the influence of our scientific achievements, we were invited to serve in over 30 program committees, including all the major conference in this research field: *International Conference on Software Engineering* (ICSE, NIER Track) *International Conference on Software Maintenance* (ICSM), *Working Conference on Reverse Engineering* (WCRE), *European Conference on Software Maintenance and Reengineering* (CSMR), *International Conference on Program Comprehension* (ICPC), *Model Driven Engineering Languages and Systems etc.*.

In 2010 we served as *General Chair* of the 26th *IEEE International Conference on Software Maintenance* (ICSM). The conference was organized in Timișoara and it was only the second time in over 25 years when this major event was hosted by an Eastern European city. In 2011 we served as co-chair of the *Doctoral Symposium* at WCRE, and in 2012 as chair of the *European Projects* track at CSMR. We were recently invited to join the Steering Committee of the CSMR and to serve as co-chair of the *doctoral symposium* at CSMR 2013. Last but not least, since 2011 we are in the editorial board of the *Journal of Software: Evolution and Process*.

Over the last ten years we were invited to give several talks at Top 500 universities like the Technical University of Karlsruhe, Technical University of Munich, University of Bern, and the University of Valladolid, and also at the IBM T.J. Watson Research Center (Hawthorne, NY, USA).

1.3.5 LOOSE: an Educational Innovation

In parallel with the research activity, we initiated an innovative educational (faculty) project in which students were organized in *virtual companies and teams*, and they were faced with a simulation of a software development process as it happens in the real-world. The project was shaped as a competition among these virtual companies, whereby the best teams and individuals received the *LOOSE Awards*. The project was an incredible success, with hundreds of students aiming to be involved in it over the 5 years since it got started. Later on, when these students got employed in local companies, they enthusiastically shared with their managers the huge influence this training had on their programming an software engineering skills. As a result, we received numerous requests from the managers of Siemens VDO, OCE Software, and Alcatel-Lucent to migrate this educational concept for the training of their own programmers. This has lead to a number of similar trainings that impacted around 100 programmers. Step-by-step this educational project had a huge impact on the programmers from many local software companies.

1.4 Research Environment and Funding

1.4.1 LOOSE Research Group

The work presented here is the result of a research effort lead at the LOOSE Research Group[2]. We co-founded LOOSE in 2002, against all odds, considering the fact that at the moment we were just about to finish the PhD, and software engineering research in Romania was rather low-profile. The vision of starting LOOSE has been to bring together young researchers from our faculty and outstanding senior students in computer science, in order to join the research experience of the former, with the student-hood creativity and enthusiasm of the latter. Over the years, we contributed

[2]http://loose.upt.ro

significantly in making LOOSE one of the leading European groups on software evolution and quality assurance.

LOOSE was founded aiming to set a right balance between performing state-of-the-art research, and adapting research achievements to the specific needs of software companies, especially to those of the many multinational IT companies (*e.g.,* Alcatel-Lucent, Siemens VDO Automotive, etc.) that have massively outsourced large segments of their software development to Eastern Europe. As a result of this vision, a significant number of excellent students were attracted to research. Some of them followed a research track, while others became influent people in software companies.

1.4.2 Research Grants

This work has been funded by Romanian and international grants and awards, that we accessed by competition as Principal Investigator, namely:

- *Continuous Assessment and Improvement of Code and Design Quality* - IBM John W. Backus Award from IBM T.J. Watson Research Center, 2009-2011
- *Detecting and Correcting Design Flaws using Eclipse* - IBM Eclipse Innovation Award from IBM T.J. Watson Research Center, 2007
- *Network of reengineering expertise (NOREX)* - Swiss National Science Foundation (IB7320-110997/2005), 2005-2008
- *Methods and instruments for continuous quality assessment in complex software systems* - CNCSIS National Grant (PCE-IDEI 357/2007), 2007-2010
- *Distributed environments for controlling and optimizing the evolution of software systems* - CNCSIS National Grant (CEEX 5880/2006), 2006-2008
- *Design quality assurance in enterprise software systems* - CNCSIS National Grant (CEEX 3147/2005), 2005-2007
- *Modeling, analysis and verification of software systems* - CNCSIS National Grant (27688/2005, A1/GR181/2006), 2005-2006
- *Integrated environment for software quality assurance* - CNCSIS National Grant (329840/2004), 2004-2005

Chapter 2

Measurement of Software Design

Metrics are a way to control quality [DM86]. In software engineering it is important and useful to measure systems, because 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.,* code that is cryptic, uncommented, badly structured, or dead). Consequently, software metrics are used to detect design problems as they quantify simple properties of design structures [HS96].

2.1 Problem Statement

Various software metrics have been defined to address the most important characteristics of good object-oriented design like complexity, cohesion, coupling and inheritance [LH93, CK94, BK95, HS96, ADB10]. In practice, appropriate tool support for metrics calculation is a must for performing quality assessments. Such tools are presented in [ARK05, Web05]. Unfortunately, many such tools do not go beyond the computation of a large number of metrics accompanied by the display of metrics in form of charts and by monitoring if some (oftentimes arbitrary) threshold values are not exceeded by the software system under scrutiny.

In the context of using metrics there are at least two major issues which must be addressed carefully: (i) devising empirical threshold values that signify abnormal characteristics of design entities [LK94], and (ii) using a rigorous approach for defining and using metrics [BDW99a] in order to avoid the inflation of useless and/or ill-defined metrics. Concerning thresholds, Lorenz and Kidd have worked on devising empirical threshold values which signify abnormal characteristics of design entities [LK94]. These thresholds were established based on the authors' experience with C++ and Smalltalk projects. Concerning the need for a rigorous approach for defining and using metrics, Briand *et al.* defined a unified framework for coupling measurement in object-oriented systems based on source model entities [BDW99a]. Based on these metrics they verified the coupling measurements at the file level using statistical methods and logical coupling information based on "ripple effects" [BDW99b]. Briand *et al.* described how coupling can be defined and measured based on dynamic analysis of systems [ABF04]. This recent study shows that some dynamic coupling measures are significant indicators of change proneness and that they complement existing coupling measures that are based on static analysis.

2.2 Design Models: the Foundation of Measurement

In general, a first mandatory step in computing design metrics is the transformation of source code into a more abstract representation form, in which only those elements are kept that are relevant in respect to the measurement activities to be performed. This first *abstraction step* and the result is usually called *meta-modeling*. Thus, a meta-model is a precise definition of the design entities and their types of interactions, used for defining and applying static analysis techniques. A meta-model has the main advantage of being easier to manipulate and understand than the code itself. All the metrics are defined as queries on the meta-model.

A good meta-model should capture only those types of entities that are relevant for the analysis, together with the properties of those entities and the relationships that exist between them. In a measurement context, the meta-model defines the boundaries of our measurement activities. By analyzing the different design entities that appear in object-oriented systems, we reached the conclusion that they belong to different categories and consequently have different compositions. The constructs and rules used to describe the meta-model are therefore distilled in the form of a *meta-meta-model* [KPF95].

In the context of describing design measurements in [LM06][1] we proposed a meta-meta-model in which we represent classes and operations as *design entities* that have *properties* and are in *relation* (*e.g.,* the visibility level of attributes) with other entities (*e.g.,* methods that access attributes). 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".

HAVING in Object-Oriented Design. In Figure 2.1 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.

USING in Object-Oriented Design In Figure 2.2 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

[1]This section is partially reproduced from [LM06], including all figures. ©Springer-Verlag Berlin Heidelberg 2006. Used by permission.

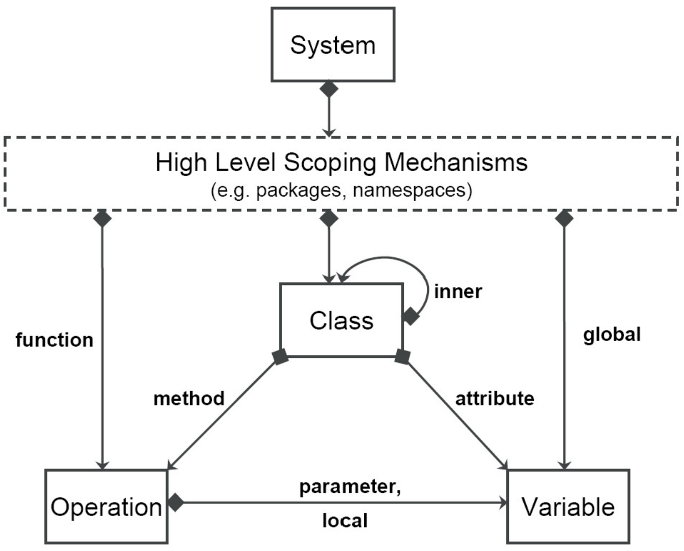


Figure 2.1: The HAVING relations.

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.

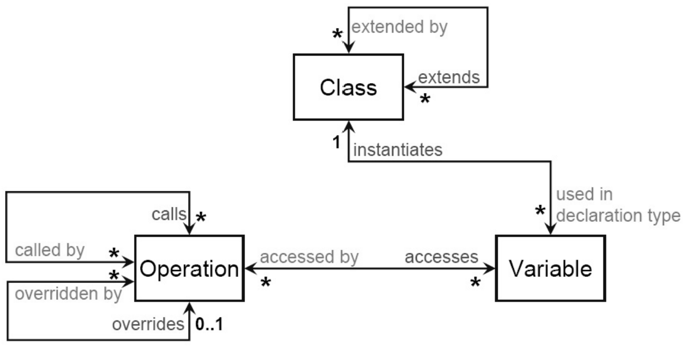


Figure 2.2: The USING relations.

BEING in Object-Oriented Design. One of the most frequently encountered dilemmas when reading any metric definition is: What is *really* counted? For example, a simple metric like *Number of Methods* seems straightforward to define. 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:

- *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?

Consequently, the HAVING-USING-BEING modeling approach allows for a systematic and unambiguous definition of metrics.

2.3 SAIL: Specification of Design Metrics

Design-related analyses can be implemented using almost any programming language (*e.g.,* Java). Unfortunately, almost all these implementations will be hard to reuse and understand and thus they hinder the correlation between the results of the implemented analyses. This happens because a single normal programming language does not provide all the proper mechanisms to ensure the easy implementations of many different analyses. Implementation approaches based on procedural or object-oriented programming languages are especially unsatisfactory for non-trivial combinations of model *navigation* and *filtering* conditions. On the other hand, approaches based on querying a repository have another major problem: they miss adequate mechanisms that would support a better modularization. This leads to analyses that consist of a monolithic query, which is very hard to maintain.

In this context we created [MMG05] SAIL (Static Analysis Interrogative Language) as a dedicated language for structural analyses, built on top of an earlier version of the meta-model presented in Section 2.2. The language provides a set of powerful mechanisms which facilitate a concise and natural expression of the implemented analyses. There are three major concepts in *SAIL*:

1. The *integration of a powerful query mechanism* (*i.e.,* the `select` statement) in a very simple structured programming language, which is syntactically very close to known programming languages like (C and Java). This brings a twofold advantage: the language adopts the key advantages of a query language, and it requires almost no learning effort for a programmer, due to its syntactic similarity with C and Java.
2. The *representation of the data model in* SAIL, although totally based on *data structures* can be used without any overhead both in imperative statements and in the query mechanism. This differentiates our approach from the *embedded SQL* approach because in *SAIL* the query mechanism is an intrinsic part of the language and thus it can work directly on the same data model as the rest of the language mechanisms (which belong to the world of imperative programming).
3. The *simple manipulation of collections* in SAIL proved to play an important role in simplifying the writing of code analyses. This is mainly because set operations an essential building stone in all non-trivial analyses.

Based on these concepts, the understandability and the changeability of the analyses implementation have been increased. On the other hand, it provides modularity mechanisms that allow us to reuse analyses and to compound them into more complex types of code inspections. In [MMG05] we validate the claim that the usage of SAIL would lead to a simplified expression of static analyses by comparing the size and complexity of implementation for a suite of over 40 object-oriented design metrics, most of them quite complex. These metrics were all implemented in

Java, SQL and respectively SAIL. The comparison has revealed that while SQL implementations usually need less lines of code to be implemented than Java or SAIL the complexity of each line highly exceeds those of found in the approaches based on structured/object-oriented programming. On the other hand comparing SAIL and Java implementations, the analyses written in SAIL prove to be significantly more concise (yet readable) than those implemented in Java. This supports the hypothesis that while keeping the shape of a procedural language, SAIL adds to it the conciseness of a query languages.

2.4 Determining Metric Thresholds

With any used metric 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. Thus, 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.

In [LM06] we addressed this issue by proposing a novel approach for determining thresholds using statistical information [2].

The problem 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.

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?

[2] This section is partially reproduced from [LM06], including all figures. ©Springer-Verlag Berlin Heidelberg 2006. Used by permission.

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 illustrate next our new approach for computing statistics-based thresholds by measuring a large number of Java and C++ systems. For illustrating the principle we consider the following three metrics:

1. Average *Number of Methods (NOM)* per class
2. Average *Lines of Code (LOC)* per method (operation)
3. Average *Cyclomatic Number (CYCLO)* [McC76] per line of code

These three metrics have three important characteristics, which makes the gathering of statistical data for them meaningful: (i) they are elementary metrics that address the key issues of size and complexity; (ii) they are independent of each other; (iii) 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 covers most projects.
- the *Lower* and respectively the *Higher* margins of this interval.
- the *Extreme high values*, *i.e.,* a value beyond which we have outliers.

We use two statistical means to find what the typical high and low values are: (i)*Average* (AVG), to determine the most typical value of the data set (*i.e.,* the central tendency); and (ii)*Standard deviation* (STDEV), to get a measure of how much the values in the data set are *spread.* [3].

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[4] and the threshold for very high values. These are:

- *Lower margin*: $AVG - STDEV$.
- *Higher margin*: $AVG + STDEV$.
- *Very high*: $(AVG + STDEV) \cdot 1.5$

In other words 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 Figure 2.3. These margins tell us now the meaning of *Low*, *High* and *Very High* for a given metric. Based on the information from Figure 2.3 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.

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.

[3]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 (i) a non-negative number, and (ii) 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

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

	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

Figure 2.3: Statistical thresholds based on 45 Java and 37 C++ systems computed for several well-known size and complexity 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 Figure 2.3 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, *i.e.*, LOC/Class and AMW (Average Method Weight) *a.k.a.* CYCLO/Method.

2.5 The Overview Pyramid

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

1. *Size and complexity.*

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 defined in [LM06] the *Overview Pyramid*[5], 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 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 template for presenting (and interpreting) system-level measurements in a unitary manner.

[5]This section is partially reproduced from [LM06], including all figures. ©Springer-Verlag Berlin Heidelberg 2006. Used by permission.

2.5.1 Components of the Overview Pyramid

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.

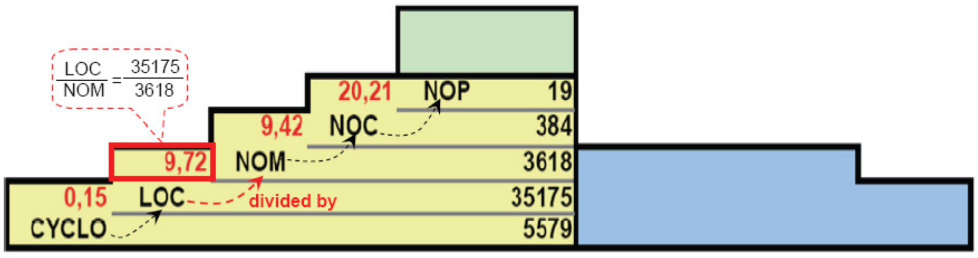


Figure 2.4: Size and complexity characterization.

The Left Part: System Size and Complexity The left side of the *Overview Pyramid* (Figure 2.4) provides information characterizing the *size* and *complexity* of the system. 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) [McC76]. We use the following metrics for the *size and complexity* side of the *Overview Pyramid*:

- *Number of Packages (NOP)*, *i.e.*, the number of high-level packaging mechanisms
- *Number of Classes (NOC)*, *i.e.*, the number of classes defined in the system
- *Number of Operations (NOM)*,*i.e.*, all user-defined operations, including methods or global functions.
- *Lines of Code (LOC)*, *i.e.*, the lines of all user-defined operations.
- *Cyclomatic Number (CYCLO)*, *i.e.*, the total number of possible program paths summed from all the operations in the system. It is the sum of the McCabe cyclomatic number [McC76] for all operations. We use this metric to quantify the intrinsic functional complexity of the system.

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 Figure 2.4). 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 Figure 2.4 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 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 operations in the system 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 The second part of the *Overview Pyramid* provides an overview with information about the level of coupling in the system, by means of operation invocations. The two direct metrics that we use are:

- *Number of Operation Calls (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.
- *Number of Called Classes (FANOUT), i.e.,* this is computed as a sum of the FANOUT [LK94] metric, namely classes from which operations call methods, for all user-defined operations. This metric provides raw information about how dispersed operation calls are in classes.

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.

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

1. *Average Number of Derived Classes (ANDC)*, *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.
2. *Average Hierarchy Height (AHH).* 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.

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).

2.5.2 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. As emphasized in Section 2.4 in order to ensure a reasonable level of objectiveness 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 Figure 2.5 normal, too small or too large? We need a reference point.

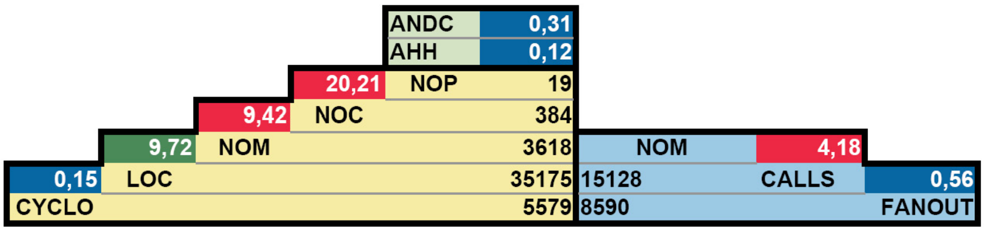


Figure 2.5: Using colors to interpret the *Overview Pyramid*. BLUE means a *low* value; GREEN means an *average* value; RED stands for a *high* value.

Based on the statistical thresholds described in the previous section (see Section 2.4) and using the same statistical base, we computed the *low*, *average* and *high* thresholds for all the proportions. As mentioned before, these metrics are collected from a statistical base of 45 Java projects and 37 C++ projects, of various sizes (from 20,000 up to 2,000,000 lines), and various application domains; projects are both open-source and commercial.

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 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*). Thus, we place a computed proportion in a *blue* rectangle to show that the value is close to the *low* threshold. Similarly, if a value is close to the *average* threshold it will be placed in a *green* rectangle; eventually, if the computed value is close to the *high* threshold, the number will be placed in a *red* rectangle.

Chapter 3

Detection of Design Flaws

There is no perfect software design. Like all human activities, the process of designing software is error prone and object-oriented design makes no exception. The flaws of the design structure, also known as *bad smells* or *code smells* [FBB$^+$99]) have a strong negative impact on quality attributes such as flexibility or maintainability. Thus, the identification, detection and correction of these design flaws is essential for the evaluation and improvement of software quality.

3.1 Problem Statement

Although metrics are essential in controlling the quality of a system, if taken in isolation they cannot serve the goal of controlling design quality [LM06]. Moreover, in spite of the fact that all major IDEs provide extensive quality assurance modules that use metrics to control design quality, in the last years it became more and more clear that there are several major problems when it comes to using metrics in practice [Mar04]: First, when metrics are used in *isolation* they are too fine grained to quantify comprehensively one investigated design aspect (*e.g.,* distribution of intelligence among classes). Thus, in most cases individual measurements do not provide relevant clues regarding the cause of a problem. In other words, a metric value may indicate an anomaly in the code but it leaves the engineer mostly clueless concerning the real cause of the anomaly. Thus, the *bottom-up approach i.e.,* going from abnormal numbers to the recognition of design flaws is impracticable because the symptoms captured by single metrics, even if perfectly interpreted, may occur in several flaws: the interpretation of individual metrics is *too fine-grained* to indicate a particular design problem. Second, as a consequence of the previous remark, in practice it is very hard to correlate an abnormal metric value with a concrete restructuring measure, that would improve the quality of design.

Beyond the issue of granularity of metrics, there is a second problem: it is impossible to establish an objective and general set of rules that would automatically lead to high-quality design. However, there is a common understanding of some general, high-level characteristics of a good design [Pre10]; notably, Coad and Yourdon identify four essential traits of a good object-oriented design, namely: low coupling, high cohesion, moderate complexity and proper encapsulation. As a consequence, over the last two decades, many authors were concerned with identifying and formulating design principles [M.88] [Lis87] [Mar02c], rules [CY91b] [M.88], 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 [BMM98] 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.

In order to detect design flaws, various detection techniques can be used. Some of these techniques are based on Prolog rules to describe structural anomalies [Ciu99], while others are metrics-based [Mun05, LM06, Tri08], or use a dedicated specification language [MGDLM10] that also allows taking into account correlation among flaws. In this section we present our approach on combining metrics in order to serve the identification and location of design problems and thus contribute to controlling design quality.

3.2 Detection Strategies: Rules for Detecting Design Flaws

We emphasized that a metric alone cannot answer all the questions about a system and therefore metrics must be combined to provide relevant information[1]. 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 then combine 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*, and is defined as *a composed logical condition, based on metrics, by which design fragments with specific properties are detected in the source-code* [Mar04, LM06].

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.

3.2.1 Defining a Detection Strategy

Starting from the informal description of design flaws [FBB$^+$99, Rie96, BMM98] we have defined a large set of detection strategies [Mar04, LM06, Tri08]. Each detection strategy was defined using the Goal-Question-Metric (GQM) methodology [BR94], which defines a three-level quantification model. The first, *conceptual* level defines the measurement goal, which in our case is to detect the presence of a particular design flaw. On the second level, the goal is refined in form of a set of *questions*, which in this particular case capture the specific traits of a design flaw. Finally, each question is associated with one or more *metrics* that answer the question in a measurable way.

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

[1]This section is partially reproduced from [LM06], including all figures. ©Springer-Verlag Berlin Heidelberg 2006. Used by permission.

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 respectively *lower/upper tail*. 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.

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 Section 2.4, 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.

3.2.2 Detection Strategies Exemplified

As mentioned before, *detection strategy* can be used to express in a quantitative manner deviations from a given set of *rules of good design*. While it is impossible to establish an objective and general set of such rules that would lead automatically to high-quality design if they would be applied, yet heuristic knowledge reflects and preserves the experience and quality goals of the developers.

Let us see now, based on the concrete example of the *God Class* [Rie96] design flaw, how design problems can be defined for a concrete design flaw. The entire process is summarized in Figure 3.1.

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 [Mar02c] or Fowler [FBB$^+$99] — that comprehensively define the design problem 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:

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 [BDW99a, 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:

- *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].

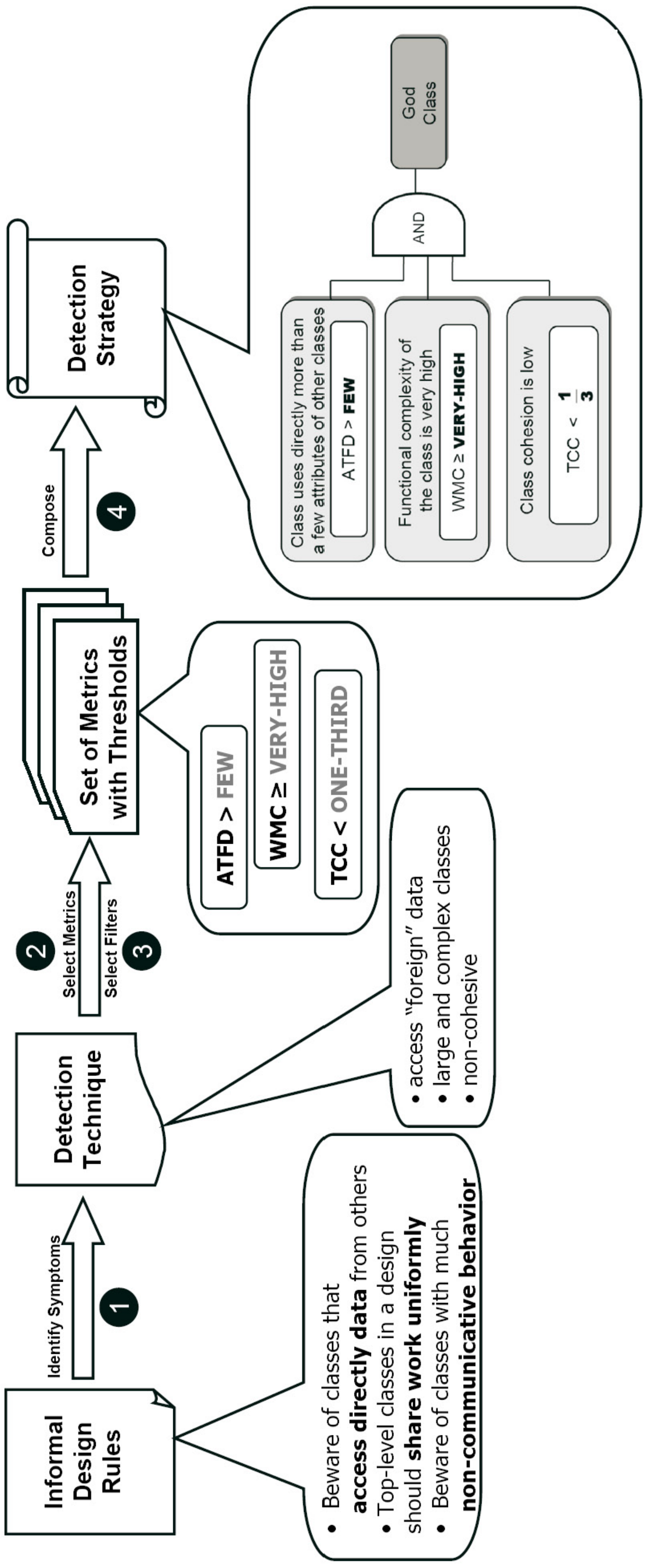

Informal Design Rules
Beware of classes that access directly data from others
Top-level classes in a design should share work uniformly
Beware of classes with much non-communicative behavior
1 Identify Symptoms
Detection Technique
access "foreign" data
large and complex classes
non-cohesive
2 Select Metrics
3 Select Filters
Set of Metrics with Thresholds
ATFD > FEW
WMC ≥ VERY-HIGH
TCC < ONE-THIRD
4 Compose
Detection Strategy
Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY-HIGH
Class cohesion is low
TCC < 1/3
AND
God Class

Figure 3.1: Process of transforming an informal design rule in a *detection strategy*.

- *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 [Mar04].

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 (i) pick-up a comparator and (ii) to set an adequate threshold.

In our concrete case, the first symptom is referring to *excessively high* class complexity; therefore 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 Section 3.2.1. 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. the *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. As this filter must capture non-cohesive classes, we decided to use the *one-third* threshold, 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 detection strategy for *God Class* can be expressed as follows [LM06]:

```
God Class = (WMC > VERY_HIGH) and (ATFD > FEW) and (TCC < ONE_THIRD)
```

3.2.3 The Issue of Thresholds

The *God Class* detection rule clearly shows that the most sensitive part of any metrics-based technique is the selection of concrete threshold values. In [MM05] we defined a novel method for establishing proper threshold values for *detection strategies*. The method is based on inferring the threshold values based on a set of reference examples, manually classified in *flawed* respectively *healthy* design entities (*e.g.,* classes, methods). More precisely, the tuning machine searches, based on a genetic algorithm, for those thresholds which maximize the number of correctly classified entities. In the aforementioned paper we also define a repeatable process for collecting examples, and discuss the encouraging and intriguing results while applying the approach on two concrete detection strategies that capture two well-known design flaws *i.e.,God Class* and *Data Class*.

3.2.4 Web of Correlated Detection Strategies

As a result of this approach we defined in [LM06] and in other subsequent publications [Mar12] almost 20 different detection strategies for design flaws ranging from fine-grained ones that affect methods (*e.g., Code Duplication*) to architectural flaws that occur at the level of subsystems (*e.g., Cyclic Dependencies* [Mar02c]).

Most of the times these design flaws do not appear in isolation. Therefore, when addressing design flaws we have to take into account also the most common correlations between the various disharmonies. For example, in Figure 3.2, we depict the *web of correlations* for the design flaws defined in [LM06]. The correlation enable us to propose adequate correction plans, like the one described in Section 6.3.

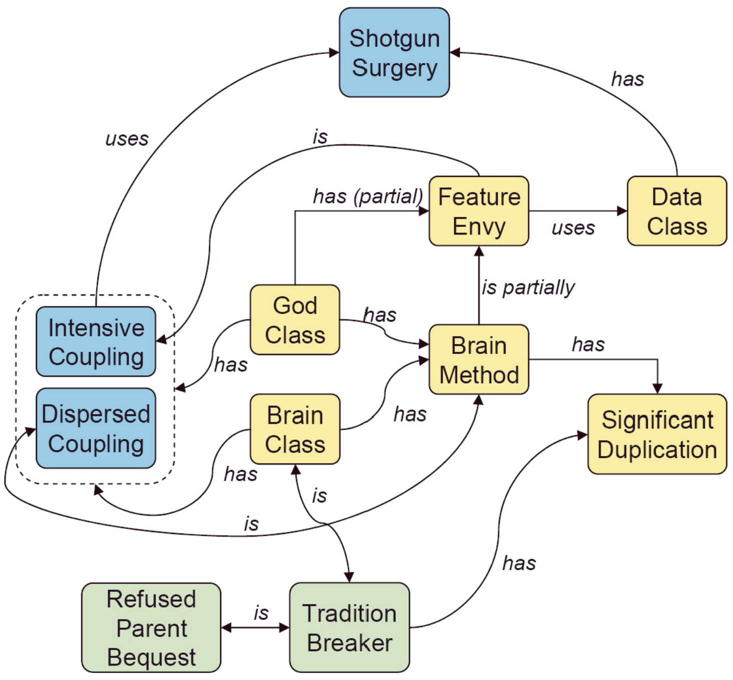


Figure 3.2: Disharmonies and their correlations.

3.3 History-Enriched Detection of Design Flaws

Design flaws are like human diseases — each of them evolves in a special way. Some diseases are hereditary, others are gained during the life-time. The hereditary diseases are there since we were born. If the physicians are given a history of our health status over time they can give the diagnostic in a more precise way. Moreover there are diseases with which our organism is accustomed/immune and thus represent no danger for our health and we don't even consider them to be diseases any more.

3.3.1 Refining Detection Rules

In [RDGM04] we propose an approach in which we use information about the evolution of a system to increase the accuracy of design flaws detection. We analyze the history of the suspects to see whether the flaw caused problems in the past. If in the past the flaw proved not to be harmful then it is less dangerous. For example, in many cases, the generated code needs no maintenance so the system which incorporates it can live a long and serene life no matter how the generated code appear in the sources (e.g., large classes or unreadable code).

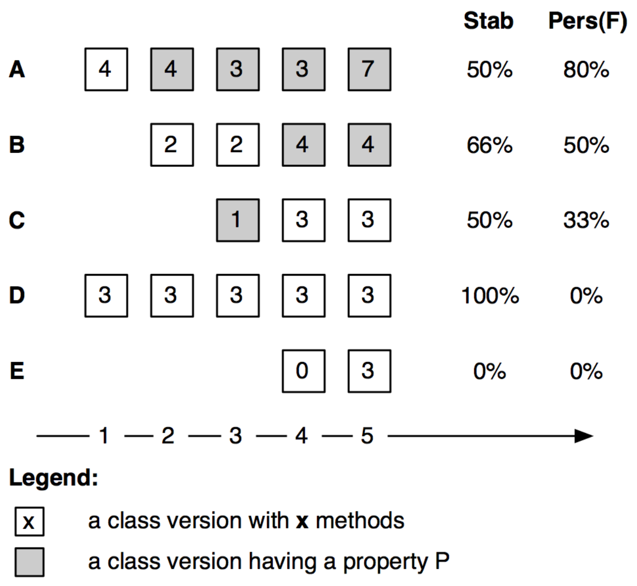


Figure 3.3: Examples of the computation of STAB and PERS

We refine the detection of design flaws by taking into consideration how *stable* the suspects were in the past and how long they have been suspected of being flawed (*persistency*). For each of these two aspects we define a metric, applied on a class history (see Figure 3.3):

- STAB is the stability metric, and it is defined as the number of versions in which a class was changed over the total number of versions. In this context a *change* is defined between two consecutive versions as an increase or decrease of the number of methods of a class.

- PERS is the persistency metric, and it is defined as the relative number (percentage) of versions in which the measured class was affected by a given design flaw.

By taking into account this historical information, we have now two ways of refining the *God Class* detection strategy (see Section 3.2): *Stable God Class* and *Persistent God Class*.

Stable God Classes can be detected as follows

```
Stable God Class = (God Class) and (Stab > 95%)
```

As we know *God Classes* are big and complex classes which encapsulate a great amount of system's knowledge. They are known to be a source of maintainability problems. However, not all God Classes raise problems for maintainers. The stable God Classes are a benign part of the God Class suspects because the system's evolution was not disturbed by their presence. For example, we found cases where they implemented a complex yet very well delimitated part of the system containing a strongly cohesive group of features (*e.g.,* an interface with a library). On the other hand, the changes of a system are driven by changes in its features. Whenever a class implements more features it is more likely to be changed. God Classes with a low stability were modified many times during their lifetime. Therefore, we can identify God Classes which raised maintenance problems during their life from the set of all God Classes identified within the system. The unstable God Classes malign sub-set of God Class suspects.

Persistent God Classes The persistent God Class are those classes which have been suspects for almost their entire life. Particularizing the reasons given above for persistent suspects in general, a class is usually born God Class because one of the following reasons:

1. It encapsulates some of the essential complexities of the modeled system. For example, it can address performance problems related to delegation or it can belong to a generated part of the system.
2. It is the result of a bad design because of the procedural way of regarding data and functionality there being an emphasis on the functional decomposition instead of data centric decomposition.

Persistent God Classes can be detected as follows

```
Persistent God Class = (God Class) and (Pers[GodClass] > 95%)
```

It is obvious that God Classes which are problematic belong only to the last category because in the first category the design problem can not be eliminated. God Class suspects which are not persistent, obtained the God Class status during their lifetime. We argue that *Persistent God Classes* are less dangerous than those which are not persistent. The former were designed to be large and important classes from the very beginning and thus are not so dangerous. The later more likely occur due to the accumulation of accidental complexity resulted from the repeated changes of requirements and they degrade the structure of the system.

3.3.2 Detecting History-Specific Flaws

Software systems need to change over time to cope with the new requirements. However, as requirements happen to crosscut the system's structure, changes will have to be made in multiple places. Research has been carried out to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes: to identify hidden architectural dependencies [GHJ98], to point developers to possible places that need change [ZWDZ04], or to use them as change predictors [HH04].

The detection is mostly based on mining versioning systems like CVS and in identifying pairs of changed entities. Entities are usually files and the change is determined through observing additions or deletions of lines of code. Also, changes are interpreted between pairs of entities. In [GDK$^+$07] we proposed a different approach, focused on identifying patterns of change that affect several entities in the same time. For this we use formal concept analysis [GW99], which is a technique that identifies sets of elements with common properties based on a given matrix that specifies the elements on the rows, properties on columns and the value of a field *(i, j)* is marked as true if the element *i* has property *j*.

To identify how entities changed in the same way, we use historical measurements to detect changes between two versions. For each history we identity each version in which a certain change condition is met. To use formal concept analysis, we use histories as elements, and "*changed in version j*" represents the *jth* property of the element. Furthermore, for building the matrix of changes, we make use of logical expressions which combine properties with thresholds and which run on two versions of the system to detect interesting entities. In this way, we can detect changes that take into account several properties.

This technique enabled the detection of new types of design flaws, that could not be detected otherwise. For example, *Shotgun Surgery* is a flaw that is indicated by the fact that every time we have to change a class, we also have to change a number of other classes [FBB$^+$99]. We would suspect a group of classes of such a bad smell, when they repeatedly keep their external behavior constant and change the implementation. We can detect this kind of change in a class in the versions in which the number of methods did not change, while the number of statements changed. Another design flaw detected by using this technique is *Parallel Inheritance i.e.,* classes which change their number of derived classes together [FBB$^+$99]. Such a characteristic is not necessary a bad smell, but gives indications of a hidden link between two hierarchies. For example, if we detect a main hierarchy and a test hierarchy as being parallel, it gives us indication that the tests were developed in parallel with the code.

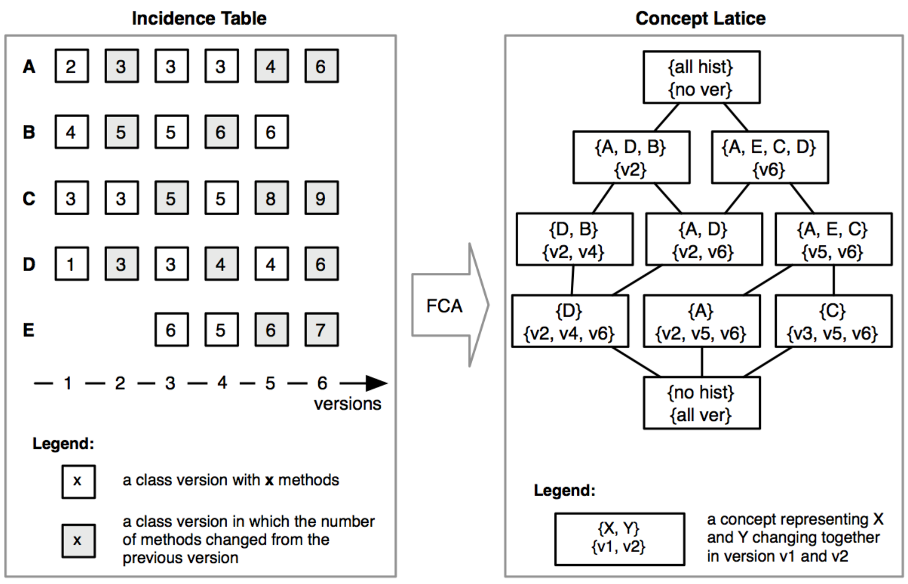


Figure 3.4: Example of applying formal concept analysis to group class histories based on the changes in number of methods. The Evolution Matrix on the left forms the incidence table where the property P_i of element X is given by "*history X changed in version i.*"

We depict in Figure 3.4 an exemplification of the approach. To the left, instead of a table, we use the notation of an Evolution Matrix [13] in which each square represents a class version and the number inside a square represents the number of methods in that particular class version. A grayed square shows a change in the number of methods of a class version as compared with the previous version. We use the matrix as an incidence table, where the histories are the elements and the properties are given by "*changed in version j*". Based on such a matrix we can build a concept lattice. To the right side of figure we show the concept lattice obtained from the Evolution Matrix on the left. Each concept in the lattice represents all the class histories which changed certain properties together in those particular versions. In the given example, class history A and D changed their number of methods in version 2 and version 6.

To identify a change we want to be able to take into account several properties, and not only one. For example, to detect *Parallel Inheritances* it is enough to just look at the number of children of classes; but, when we want to look for classes which need to change the internals of the methods in the same time without adding any new functionality, we need to look for classes which change their size, but not the number of methods. We encode this change detection in expressions consisting of logical combination of historical measurements. These expressions are applied at every version. In the example from Figure 2, we used as expression $E_i(NOM) > 0$ and we applied it on class histories.

We have started to apply this approach on several case studies (JBoss and ArgoUML) and the initial results have proven the feasibility of the approach.

3.4 Detection of Duplicated Code

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. In our view a case of duplication is considered significant if: (i) 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 and (ii) it is large enough.

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.

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.

In order to address this situation in [WM05] and [LM06] we proposed a novel, automated approach for recovering duplication blocks, by composing small isolated fragments of duplication into larger and more relevant duplication chains. Using a metaphor, our approach is similar to an archeologist that finds the ruins of an ancient village and will try to put all the pieces together for a better comprehension of the whole picture, rather than analyzing each artifact separately. In a similar manner, we try to recover a close representation of a duplicated block, before making decisions of refactoring that code.

The approach is based three low-level duplication metrics (see Figure 3.5):

1. *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.

2. *Line Bias (LB)*. When comparing two pieces of code 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

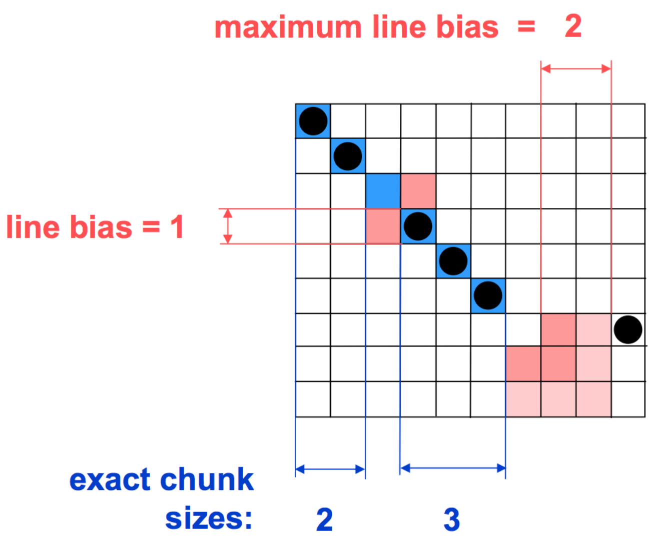


Figure 3.5: Metrics involved in detecting a duplication chain

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).

3. *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).

These three metrics can have a major influence on the detection process and results. Taking the example depicted as as scatterplot in Figure 3.5, let us assume that we are interested in duplications chains with SDC over 4 LOC and a maximum LB of 2. If we would set the minimum SEC to 3, then the first exact chunk starting from the upper-left corner would be overlooked because of that. Then the second exact chunk of length 3, because of the maximum LB of 2 will not be able to find a next exact chunk in its vicinity and it will also be missed because of the minimum SDC of 4. In this case, we would miss all the duplication information in this area. But if we would set a minimum SEC of 2, the first and second exact chunks will be merged because the LB between them is 2, which is acceptable for our example (maximum LB = 2). The chain would end here, because the next exact chunk is located at a distance of 3. However, the duplication chain has a length of 6, which qualifies it as a significant du- plication chain for the given parameters.

By performing various experiments we can draw the following conclusions:

- By using our approach we detect more clones (75% more,in this experiment) that a usual line-based clone detector.

- Some of these extra duplication chains are valuable results and can lead to refactorings.
- The recall of our approach is 89% under the strict conditions of the experiment described in [Bel02], but in a more loose context the recall could rise up to 95%.

3.5 Verification of Architectural Constraints

The size and complexity of software systems is constantly and abruptly increasing, as well as the size of the teams who develop them. Although software systems usually start with a clean design and an unitary architecture, preserving the design quality in the final product, especially its modularity and reusability, depends on the programmers' ability to understand, implement and maintain the initial architecture of the system. In other words, it depends on the ability to preserve a common vision about the high-level design i.e., to preserve the *architectural integrity*.

This is an essential problem, and it has led to a number of approaches that help maintaining architectural integrity by allowing for the specification and checking of architectural rules (constraints) in code. Unfortunately, these approaches are rarely used in practice because of their excessive complexity, lack of flexibility and absence of integration with the actual development environment. In [MG10] we propose a new, agile approach to defining and checking architectural constraints. The proposed solution consists, on one hand, of the INCODE.RULES *language* that offers a highly intuitive, yet flexible, means for defining architectural rules. On the other hand, INCODE.RULES is far more than a language specification: architectural rules can be automatically checked using the inCode.Rules interpreter, they can be easily edited using the full-fledged editor that we created. Both the interpreter and the editor are tightly integrated in the Eclipse IDE. Furthermore, INCODE.RULES has grown beyond being just a prototype as it has been already successfully applied on large-scale systems of over 1 MLOC. Thus, INCODE.RULES provides a more agile approach to architecture verification, as it brings the evolution of code and architecture closer than ever before.

INCODE.RULES supports two different sets of rules: (i) *usage rules* and (ii) *property rules*.

Usage rules. This category of rules are meant to provide the designer the ability to break-down the system into components, or modules, as well as to specify the usage relationships between the components. The beauty of this rule type is that one can define components that overlap, thus allowing the designer to specify more than one modularization view of the same system.

For instance, we might want to specify that *neither calls nor accesses* are made from package named `a.b` to package named `x.y`. This can be specified as follows in INCODE.RULES:

Listing 3.1: Composed Action Rule

```
package named "a.b" must not (call or access)
package named "x.y";
```

Property rules have another role: they allow the designer to enforce rules using filters and properties that are already defined in INCODE (see Section 5.3). In contrast to a *usage rules*, a *property rule* is asymmetric: it consists only of a subject and an action. While the subject is specified exactly as in *usage rules*, the action is specified differently, namely by the `have` keyword. Below is an example of a *property rule* stating that the system is not allowed to contain any classes that have the *Data Class* design flaw, namely classes that are "dumb" data holders without complex functionality, on which other classes strongly rely in terms of data-access [LM06]:

Listing 3.2: A Simple Property Rule

```
 classes must not have "Data Class";
```

As mentioned before, these properties (*e.g.,* "Data Class") are made available to `inCode.Rules` by the underlying software analysis infrastructure (*i.e.,* inCode). Properties are expressed as *property strings* and they correspond to a continuously growing set of quality assessment analyses defined in `inCode`. This means that `inCode.Rules` is also continuously enriching its *vocabulary* as it may use any of those externally available quality properties.

Furthermore, properties can be **composed** using the `or` and `and` composition operators, which consequently allows us to write more complex *property rules* like in the following example:

Listing 3.3: Composed Filter

```
classes must not have
   ( "Data Class" or "God Class" );
```

Exceptions. *Change* is an intrinsic property of software, and it would be foolish to think that a set of design decisions (let alone rules) will be valid and respected throughout the entire lifecycle of the system. This is the main reason why the `inCode.Rules` language supports the concept of **exceptions**. There is also a second reason for introducing exceptions: these language constructs increase the expressivity of the language, increasing thus their readability. For instance, consider a package `org.x` with four classes A, B, C and D. The design states that package `org.x` is not allowed to use package `org.y` except for class D. If exceptions did not exist we would have to write three rules to code the design, one for each class except class D. With exceptions we only need to write one rule and one exception.

Exceptions are an optional part of a rule and they appear after the rule definition. For example :

Listing 3.4: Exception

```
package named "org.x" must not use package named "org.y"
  except {
    class named "org.x.ThisClass"
    may use class named "org.y.ThatClass"
};
```

In conclusion, INCODE.RULES is an ADL that provides three main advantage:

1. *Simplicity* – as it is easy to write (and re-write) rules, not only by trained architects, but also for any developer. The language does not have a steep learning curve, as constraints are expressed in a way close to natural language. Therefore, it can serve not only as an input for the automatic checking of the architectural rules, but also as a proper architecture documentation.

2. *Flexibility* – as it supports the specification of architectural rules at various granularity levels (*i.e.,* it is possible to write rules both in terms of ”packages” as well as in terms of ”classes” and ”methods”. Furthermore, the language has flexibility not only in terms of the involved entities, but also in terms of the granularity at which *relations* can be defined.

3. *Integration* – as it is implemented as an Eclipse plugin, and thus is part of the Eclipse ecosystem, probably the most widely used environment for Java development. Thus, its implementation and tool support (editor, interpreter, rule checker etc) are tightly integrated with the IDE itself.

Chapter 4

Assessment of Design Quality

We measure because we want to assess and eventually improve the design quality of systems. Actually, our goals is to bridge the gap between how quality is perceived and how it is assessed internally through measurements at the design level. Without an assessment of product quality, speed of production is meaningless. This observation has led software engineers to develop models of quality whose measurements can be combined with those of productivity. Using *detection strategies* to raise the abstraction level in detecting design flaws is a significant step forward. However, for a global assessment one needs more than just a list of design flaw instances. In this context, *quality models* are needed for getting an overall assessment of design quality.

4.1 Problem Statement

The idea of aggregating basic quality indicators into a quality model is not new. The literature proposes a large number of quality models [Dro95, KLPN97, SBL01, BD02, DSP$^+$07] all based on the *Factor-Criteria-Metric* (FCM) de-compositional approach introduced by McCall [MRW76] and Boehm [BBK$^+$78]. FCM models describe complex quality aspects by breaking them down into more manageable criteria.

More recently, several European projects tried to address the issues of quality assessment by proposing new models, techniques and tools, namely:

- *QualOSS* [Con09b] proposes to build a methodology and the associated tools to benchmark the quality of open source software in order to assist companies in their strategic decision to integrate F/OSS. While the methodology developed in the project is valuable asset in evaluating maintainability, it is very much limited to the specificities of F/OSS software, which are very hard to be extended to non-F/OSS software.

- *SQO-OSS* [Con09c] aimed to implement a software quality-checking system that can be used by F/OSS software projects to gather information about the quality of their code and relate this information to other data sources such as issue-tracker, data and mailing-list archives. The most valuable result of this project is the resulting plugin-based platform for software quality analysis (*Alitheia*); however, the project could not provide a sufficiently solid validation of the efficiency and accuracy of the quality model for assessing maintainability.

- *FLOSSMETRICS* [Con09a] pursued a very ambitious and worthwhile goal, namely to construct, publish and analyze a large-scale database with information and metrics about F/OSS software development coming from several thousands of software projects, using existing methodologies, and tools already developed.

However, the source-code metrics collected have been rather simple metrics that do not evenly cover all design aspects, which are relevant in the context of a quality assessment

Unfortunately both the initial models and the more recent quality models proposed to assess maintainability have failed so far to establish a widely acceptable basis for quality assessment. We believe that one of the major problems with FCM models is the unclear decomposition criterion that leads to a "*somewhat arbitrary selection of characteristics*" [KP96]. Moreover, if the model is a fixed one, it will be hard to understand, as the quality principles that dictate the mappings remain implicit [MR04].

A notable attempt to create a widely acceptable basis has been the definition of the ISO 9126 standard [ISO91]. However, in spite of the rigorous definitions provided for the various quality factors and criteria, this standard – in fact, any standard – fails to properly address an essential problem, namely the model's operationalization. Thus, quality models are not very useful as long as they remain on a high level of abstraction, with no concrete quantifications and/or adequate tool support to conduct quality evaluations.

In conclusion, there are two major problems with current quality models:

1. There is a large gap between design principles and design metrics. Thus, there is still an important gap between what we measure and what is important in design.

2. There is a lack of relevant feedback link in quality models. We apply metrics we identify suspects, but the metric by itself does not provide enough information for a transforming the code so that it would improve quality. For example, what about a method with more than 1200 LOC? Should it be split, or could a part of it be factored out in a base-class, or should the method (or a part of it) be moved to another class? If the metric is considered in isolation it is hard to say. Thus, the developer is provided only with the problem and he or she must still empirically find the real cause and eventually look for a way to improve the design.

4.2 Factory-Strategy Quality Model

In [MR04] we propose a novel approach to the issue of quantifying the impact of object-oriented design on the high-level quality factors of software, like maintainability or portability. In order to bridge the gap between qualitative and quantitative statements relate to object-oriented design we propose a quality model that has two major characteristics:

- an easy and intuitive construction;

- a direct link at the design level to the cause(s) and location(s) of quality problems.

We are aware of the fact that high-level quality factors are influenced also by other criteria (*e.g.,* technologies involved, selection of algorithms or database schemas) than the design structure. Because of that, we focus our attention consciously on those aspects of quality that are heavily impacted by design problems, especially on maintainability. We would also like to emphasize that the approach is language-independent, while the toolkit proposed for automation currently supports the JAVA and C++ languages.

4.2.1 Limitations of Factor-Criteria-Metrics Models

As mentioned in the beginning, *Factor-Criteria-Metrics* (FCM) models are constructed in a tree-like fashion, where the upper branches hold important high-level *quality factors* related to software products, such as reliability and maintainability, which we would like to quantify. Each quality factor is composed of lower-level *criteria*, such as structuredness and conciseness. These criteria are easier to understand and measure than the factors themselves, thus actual metrics are proposed for them. The tree describes the relationships between factors and criteria, so we can measure the factors in terms of the dependent criteria measures (*e.g., structurednes* can be associated with a measure of class cohesion, one measuring the complexity of methods, and a third one measuring the coupling to other classes). This notion of divide-and-conquer has been implemented as a standard approach to measuring software quality [ISO91].

Although this approach is cited throughout the whole software engineering literature and is implemented in several commercial CASE tools it has two main drawbacks that limit its usability.

Drawback 1: Obscure mapping of quality criteria onto metrics. When analyzing different FCM models the first question that pops-up is: how are the quality criteria mapped to metrics? The answer to this question is essential because it affects the usability and reliability of the whole model. In the FCM approach this explicit mapping between quality criteria on one hand and rules and principles of design and coding on the other hand implicitly (and obscurely) contained in the mapping between the quality criteria and the quality metrics. Thus, the answer to the previous question is: quality criteria are mapped into metrics based on a set of rules and practices of good-design. But this mapping is "hidden" behind the arrows that link the quality criteria to the metrics, making it in most of the cases impossible to trace back. This observation reveals a first important drawback of the FCM quality models: if the model is a fixed one, it will be *hard to understand*, because we can only guess what are the rules and principles that dictated the mapping. In case of a "user-defined" model, the model is *hard to define* because when we mentally model quality we reason in terms of *explicit* design rules and heuristics, keeping the quality criteria *implicitly* contained in the rules.

Drawback 2: Poor capacity to map quality problems to causes. The interpretation of a quality model, must be done in terms of: *diagnosis i.e.,* what are the design problems that affect the quality of my software?, *location i.e.,* where are the problems located in the code? and *treatment i.e.,* what should be changed at the design level, to improve the quality?

When analyzing a software system using a FCM model, we get the quality status for the different factors that we are interested in (*e.g.,* the maintainability is quite poor, while portability stays at a fair level). We are also able to identify a set of design fragments that are supposed to be responsible for a poor status of a certain quality factor. Thus, FCM solves both the diagnosis and the location issues. But as soon as we arrive at the question concerning the treatment, we reached the limits of the FCM model, because the model doesn't help us find the *real causes* of the quality flaws detected by it. The cause of this is the fact that abnormal metric values – even if the metrics are provided with a proper interpretation model – are just *symptoms* of a design or implementation *disease* and not the disease itself. A treatment can only be imagined when knowing the *disease* not only a set of *symptoms*.

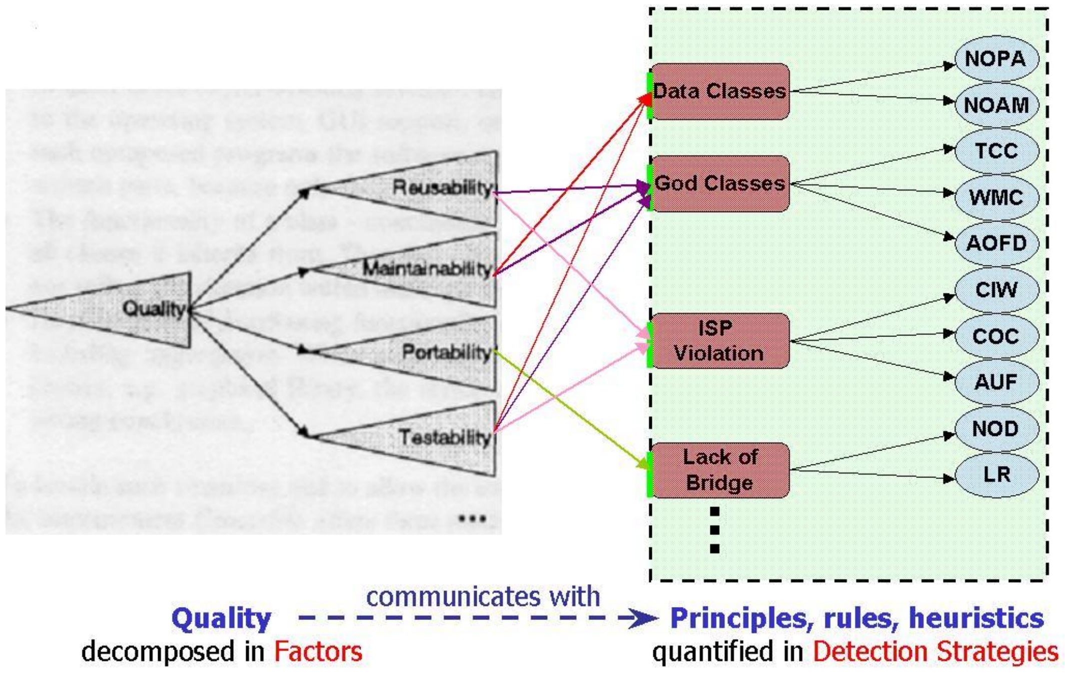


Figure 4.1: Factor-Strategy model: the Construction Principle. The acronyms that appear in the ovals on the right side represent metrics, and the arrows show which metric appears in which detection strategy (rectangles). The concrete metrics are not relevant for the understanding of this picture

4.2.2 Factor-Strategy Model: Construction Principle

Based on the *detection strategy* mechanism we proposed in [MR04] a new type of quality model, called *Factor-Strategy*(FS). This approach is intended to improve the FCM paradigm with respect to the two major drawbacks discussed in the previous section.

In Figure 4.1 we illustrate the concept of a Factor-Strategy model. FS models still use a decompositional approach, but after decomposing quality in factors, these factors are not anymore associated directly with a "bunch" of numbers, which proved to be of a low relevance for an engineer. Instead, quality factors are now expressed and evaluated in terms of detection strategies, which are the quantified expressions of the good-style design rules for the object-oriented paradigm.

Therefore we may state in more abstract terms that *in a Factor-Strategy model, quality is expressed in terms of principles, rules and guidelines of a programming paradigm.* The set of detection strategies defined in the context of a FS quality model encapsulate therefore the *knowledge-box of good design* for the given paradigm. The larger the knowledge-box, the more accurate the quality assessment is. In our case the detection strategies are defined for the object-oriented paradigm, and thus in the right side of Figure 4.1 we depicted a sample of a *knowledge-box* of object-oriented design. The knowledge-box, as such, is indispensable for any quality model. Although not visible at first sight, it is also present in the FCM models. The knowledge-box is not obvious in the FCM approach because of its *implicit character*, while it becomes *explicit* in the FS model.

4.2.3 Stepwise Construction Methodology

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

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

- The *computational aspect* must tackle the issue of quantifying the association *i.e.,* how the quality score for the higher-level element is to be computed from the quality scores of the lower-level elements associated with it.

Obviously, we must first define an association that "stands" semantically, and only then the focus must be set to finding the *association formula* that quantifies it. The association formula must reflect the *participation level* of each lower-level element within the higher-level aspect of quality. In constructing a Factor-Strategy model there are two association that must be done: the decomposition of the quality *goal* in quality *factors* and the association between these factors and detection strategies that can detect design flaws that affect the given quality factor.

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

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

Association of Factors with Detection Strategies Detection strategies used in FS models capture deviations of a design from design rules and guidelines. In [Mar04] we have described in detail the process of transforming informal design rules into detection strategies. The process of identifying the metrics needed to a quantify a given rule and the way the metrics are correlated to capture that precise aspect is done very much like in the *Goal-Question-Metric* approach [BR88]. The authors of such good-design rules implicitly or explicitly relate them to quality factors [Rie96, Mar02c, FBB+99], or to abstract design principles [M.88] (*e.g.,* abstraction, modularity, simplicity) that can be easily mapped to quality factors. Therefore, the *semantical aspect* of the association between quality factors and detection strategies becomes self-evident in the FS approach. This is one of the main advantages of the FS models over the FCM approach, where the correspondent association is subject to severe drawbacks. There are two actions related to the *computational aspect* of the association between factors and their set of strategies:

1. *Compute a quality score for each detection strategy.* We have defined two mechanisms for computing a quality score from the results of a detection strategy: first, a formula to compute the *raw-score* must be established (*i.e.,* based on the number of suspects); second, we need a *matrix of ranks* based on which the computed *raw-score* is transformed into a *quality score*, like a school grade. In other words, using this matrix we place the *raw-score* in the context of quality *i.e.,* the matrix of ranks tells us how good or bad a raw-score is with respect to the quality factor.

2. *Compute a quality score for the quality factor.* This score is also computed based on an algorithm that again involves two mechanisms: first, an *association formula* in which the operands are the *quality scores* computed for the strategies; second, a *matrix of ranks* that transforms the raw-score for the quality factor into a quality score.

4.2.4 A Factor-Strategy Model for Maintainability

The quality model that we are going to present below raises no claim of completeness. Moreover, we believe that a complete and universally acceptable quality model is impossible to define at least because of the following reasons: (i) there is no objective argument for adding or removing a component from the model; (ii) the *knowledge-box* used in the model – *i.e.,* the detection strategies defined for the model – is limited and we see no possibility of claiming completeness in this aspect. This concrete model illustrates the steps and mechanisms involved in the construction of a FS quality model. Thus, it will illustrate how metrics are encapsulated in detection strategies and how quality factors are associated with these strategies that quantify deviations from good design rules.

For describing FS quality models, we defined a simple description language, called QMDL (Quality Models Description Language). The decision to define QMDL as a variant of a description language used in connection with FCM quality models was deliberate, as we believe that this would simplify the understanding of both the commonalities and the differences between the FCM and the FS approach.

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

Listing 4.1: Maintainability Balanced

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

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

Listing 4.2: Maintainability Weighted

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

Associating Factors with Detection Strategies We briefly illustrate the process of association between a factor and a set of strategies using the *Stability* factor, which is in this model an aspect of maintainability. Stability is defined in ISO 9126 [ISO91] as the *"attributes of software that bear on the risk of unexpected effect of modifications"*. In conformity with the definitions of the design flaws that are detectable using the current set of detection strategies, and based on their impact on the desirable design properties we have selected the strategies associated with five of these flaws *i.e.,* those that affect directly stability. These are: *God Classes, Shotgun Surgery, Data Classes, God Method* and *Lack Of State* [Mar02b]. After the semantical association between the *Stability* factor and the strategies, we focus on the computational aspect, using the following sequence of steps:

- *Step 1: Choose a formula for computing the* raw-score *for each strategy.* In our case we have used for all the strategies the simplest formula, *i.e.,* the raw score is the *number of suspects* detected by that strategy.
- *Step 2: Choose an adequate* matrix of ranks *for each strategy.* We used three levels of tolerance in ranking the raw-scores for the strategies, and consequently we defined three matrices: a severe one (*SevereScoring*), a permissive one (*TolerantScoring*) and one in between the two (*MediumScoring*). For the design flaws that in our view had the highest impact on stability we applied the *SevereScoring* matrix of ranks.
- *Step 3: Define a formula for computing the* raw score for the factor. Because we intended to weight equally the five strategies when computing a score for stability, we used *average* function(*avg*). Throughout the model we applied the same function for computing the raw-scores for quality factors.
- *Step 4:Choose the* matrix of ranks *for computing the quality score for the factor.* The *raw-score* computed during the previous step must also be placed in a matrix of ranks in order to retrieve a normalized quality score.

For example, we believe the design flaws that affect stability in the highest degree are *ShotgunSurgery* and *God Classes*. Thus, while for the other design flaws we used the *MediumScoring* matrix, for the aforementioned two flaws we computed their quality score using the *SevereScoring* matrix, which is defined as follows:

Listing 4.3: Severe Scoring Matrix

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

Note that any *matrix of ranks* has three columns: the first two columns define the range (upper and lower limits) of the *raw score*, while the last column is the "grade". We chose a score (ranks) scale with the two limits being 1 (worst) and 10 (best) that corresponds to that range. In order to enhance our school-based intuitive understanding of the quality scores. Of course, this "intuition aid" has only a regional applicability; yet, the idea that stays behind the scale selection is reusable. . For example, the third line of the previous matrix is interpreted as follows: we grant a 7 mark for a system that has between 2 and 4 classes affected by the design flaw to which the matrix is attached (in this case *ShotgunSurgery* and *GodClasses*). The number of lines of such a matrix is dependent on the engineer who defines the model, and it could any value higher than 2 (*i.e.*, differentiate only between good and bad). Yet, we believe that in practice the number should not exceed 5.

Having reached the last step, we can now "reveal" how *Stability* is quantified (in QMDL):

Listing 4.4: Stability

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

The *Factor-Strategy* approach shows that the gap between qualitative and quantitative statements, concerning object-oriented software design can be bridged. While we used *detection strategies* as a higher-level mechanism for measurement interpretation, the FS quality model provides a goal-driven approach for applying the detection strategies for quality assessment.

The *Factor-Strategy* approach has two major improvements over the traditional approaches:

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

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

Besides the improvement of the quality assessment process, the detection strategies used in the context of a *Factor-Strategy* quality model proved to have a further applicability, at the conceptual level: for the first time a quality factor could be described in a concrete and sharp manner with respect to a given programming paradigm. This is achieved by describing the quality factors in terms of the detection strategies that capture design problems that affect the quality factor, within the given paradigm. We have also shown based on two versions of an industrial case study that the FS quality model is usable in practice and provides us with information that is not only relevant for quality assessment, but also for the further improvement of the system.

4.3 Assessing Technical Debt

Software systems must evolve continually to cope with requirements and environments that are permanently changing [Pre10]. Tough time to market constraints and fast emerging business opportunities require swift but profound changes of the original system. Therefore, in most software projects the focus is on immediate completion, on choosing a design or development approach that is effective in the short term, even at the price of an increased complexity and a higher overall development cost in the long term [McC07]. Cunningham introduced the term "technical debt" [Cun92] to describe this widely spread phenomenon, where *interest payments* represent the extra development effort that will be required in the future, due to the hasty, inappropriate design choices that are made today. Like in financial debt, there are two options: continuously paying the *interest*, or paying down the *principal*, by refactoring the design affected by flaws into a better one, and consequently gain by reducing the future interest payment [Fow09].

When debt is incurred, at first, there is a sense of rapid feature delivery, but later, integration, testing, and bug fixing become unpredictable and incomplete, to the point where eventually, the cost of adding features becomes so high that it exceeds the cost of writing the system from scratch [Ste11]. The cause of this unfortunate situation is usually less visible: the internal quality of the design is declining in a system [Leh96]; and duplicated code, overly complex methods, non-cohesive classes, long parameter lists, are just a few signs of this decline [FBB$^+$99]. These, and many others, are usually the symptoms of higher-level design problems, which are usually known as *design flaws* [Mar04], *design smells* [FBB$^+$99] or *anti-patterns* [BMM98].

Technical debt is not always bad. Many times the decision to incur debt is the correct one, especially when considering non-technical constraints [KTWW11]. However, most of the time, incurring debt has a negative impact on the quality of design; and, unlike its financial counterpart, technical debt starts by being *less visible*, and therefore easier to ignore [McC07].

In [Mar12][1]we defined a framework for assessing technical debt by exposing debt symptoms at the design level. The framework is based on detecting a set of relevant design flaws and measuring the negative impact of each detected flaw instance on the overall design quality.

The literature proposes various approaches [Ciu99, Mun05, LM06, Tri08, MGDLM10] to detect *design flaws* in object-oriented systems. Although these techniques are valuable for enabling the identification of individual problems, to our knowledge there are no publications proposing a framework to support the overall assessment of technical debt symptoms, by aggregating the results of individual design flaw detection techniques. Nevertheless, the idea of aggregating basic quality indicators into a quality model is not new. The literature proposes various models [Dro95, BD02, DSP$^+$07], all based on the *Factor-Criteria-Metric* (FCM) decompositional approach introduced by McCall and Boehm [Pre10]. Although the FCM approach is widely used, it has at least one significant limitation: the basic indicators in FCM are metrics, and therefore they are unable to expose the actual design flaws that lead to technical debt, as abnormal

[1]This section is partially reproduced from [Mar12], including the figure. Used by permission.

metric values are just symptoms of a design flaw, but not the flaw itself [MR04].

In [Mar12] we introduced framework that overcomes this limitation, as it builds on top of detected *instances of design flaws* instead of metrics. As a result, the framework provides additional benefits in three directions:

- *Assessment.* The framework helps assess the actual status of a design, by identifying the flaws that affect a system, and by measuring their negative impact on the overall design. Such an assessment is particularly relevant when a company acquires a software system from another company [KTWW11]. In such cases, it is essential to spot signs of a fragile design structure, which are usually the result of technical debt accumulated over time, by a team that was unwilling or incapable of paying down the "*principal*", namely to restructure the design.

- *Monitoring.* As opposed to financial debt, – where one knows from the beginning the *debt value* – when making a development decision, it is hardly possible to know beforehand its *actual impact*, even when knowing that the decision will incur technical debt. By quantifying debt symptoms, this framework makes it possible to continuously monitor the evolution of the system, during a period when such a critical design decision is implemented.

- *Restructuring.* Technical debt is reduced when a team decides to refactor the design [Fow09] instead of paying the additional costs of maintaining a flawed design fragment. By revealing the type, location and severity of design flaws this assessment framework allows a team to *prioritize* refactoring, and to perform it *systematically*.

4.3.1 Framework for Assessing Debt Symptoms

Building the assessment framework involves four steps: (i) *select* a set of relevant design flaws, (ii) *define* rules for the detection of each design flaw, (iii) *measure* the negative influence of each detected flaw instance; and eventually, (iv) *compute* an overall score that summarizes the design quality status of a system, by taking into account the influence of all detected flaws. For each of these steps, we distinguish in our description between two aspects: the conceptual side, and the decisions made for the actual implementation.

Select relevant design flaws It is impossible to establish an objective and general set of rules that would automatically lead to high-quality design. However, there is a common understanding of some general, high-level characteristics of a good design [Pre10]; notably, Coad and Yourdon identify four essential traits of a good object-oriented design, namely: low coupling, high cohesion, moderate complexity and proper encapsulation. Consequently, many authors have formulated design principles, rules, and heuristics [M.88, Mar02c, Rie96] that would help developers design better object-oriented systems. Additionally, the software refactoring community [BMM98, FBB$^+$99] is describing *design flaws*, as a means to highlight frequently encountered situations where the aforementioned design principles and rules are violated.

The framework can be used with any type of design flaws, ranging from fine-grained ones that affect methods (*e.g.,Code Duplication*) to architectural flaws that occur at the level of subsystems (*e.g.,Cyclic Dependencies* [Rie96]). The mixture of flaws that are included in an actual framework instantiation should reflect the assessment focus.

For the concrete instantiation of the framework we selected eight design flaws based on the following characteristics:

1. *Significant.* We include only design flaws that are well described in the literature and for which empirical studies or experience reports (*e.g.,* [MM11], [KDPG09]) indicate a high occurrence rate and a significant negative impact on the perceived quality of the analyzed systems (*e.g.,* high correlation with bugs).

2. *Balanced.* We select a minimal set of design flaws that cover in a balanced manner the aforementioned traits of good design identified by Coad and Yourdon [CY91a].

3. *Accurate.* We use only design flaws that proved in our earlier studies [LM06, Mar04] to be automatically detectable with a high level of accuracy (*i.e.,* good precision and recall). In fact, this is the main reason why we did not include any architectural flaws, as their detection accuracy requires a precise specification of the subsystem structure, which is often unavailable for systems such as the ones used for the case study.

These design flaws are detected automatically using detection strategies, as described in Section 3.2.

Measure the impact of design flaws Each design flaw affects to some extent the overall quality of a system, but not all have the same negative impact. So, in order to increase the accuracy of the assessment framework, the negative impact of each design flaw instance has to be correctly quantified. We take into account three factors:

1. *Influence (I_{flaw_type}).* This measures how strongly a type of design flaw affects the criteria of good design. We take into account the four aforementioned criteria identified by Coad and Yourdon [CY91a] and propose a three-level scale (HIGH, MEDIUM, LOW) to characterize the negative influence of a design flaw on each of the four criteria. The actual I^{flaw_type} values are the result of assigning numerical values to each of the three levels, and of computing a *weighted arithmetic mean* between the four criteria, where each weight represents the relative importance of a design criterion in a given assessment scenario.

2. *Granularity (G_{flaw_type}).* In general, a flaw that affects *methods* has a smaller impact on the overall quality than one affecting *subsystems.* Thus, we assign a weight to each design flaw according to the type of design entities (*e.g.,* class, method) that it affects.

3. *Severity ($S_{flaw\_instance}$).* The first two factors refer to design flaw *types*, and thus weigh equally all instances of the same flaw. However, not all cases are equal, and, therefore, we define for each flaw a *severity score* based on the most critical symptom of the flaw, measured by one or more metrics. To allow comparisons among design flaws, severity score have a lower limit of 1 (*low*) and upper limit of 10 (*high*).

Based on the three factors we compute the *Flaw Impact Score* (FIS) of a design flaw instance as follows:

$$FIS_{flaw\_instance} = I_{flaw\_type} \times G_{flaw\_type} \times S_{flaw\_instance}$$

Compute the overall score In order to get an overview of design debt in a system the various instances of anti-patterns must be aggregated.

$$DSI = \frac{\sum_{all\_flaw\_instances} FIS_{flaw\_instance}}{KLOC}$$

KLOC represents the number of thousands lines of code of the system. Scaling the *Debt Symptom Index* relatively to the code size of systems makes DSI values comparable among systems.

4.3.2 Experimental Remarks

We have used the framework to analyze 63 releases of two well-known Eclipse projects: the Java Development Toolkit (JDT) and the Eclipse Modeling Framework (EMF). The JDT project defines a full-featured Java IDE that provides a large number of views, editors, wizards, and refactoring tools [Fou11a]. The EMF project is a modeling framework and code generation facility for building applications based on a structured data model [Fou11b].

The results of this case study led us to several intriguing conclusions:

- Although our intuition says that a steep growth in code size would harm the design, the data from the two analyzed systems show something different: abrupt growths of code size are not necessarily increasing debt, nor does a smooth growth guarantee that technical debt is under control.

- Debt symptoms can have significant variations, and both systems show an alternation of decay and improvement phases. However, the starting point is very important: JDT started with a high level of debt symptoms and although reduced significantly, it stays above the values measured in the evolution of EMF.

- The most significant variations of DSI occur in the earlier releases, while later in the lifetime of the systems the DSI tends to stabilize completely. Furthermore, with one notable exception (JDT 3.3.1) the DSI suffers almost no variations during minor releases, which is a sign that none of the two development teams use the minor releases to perform refactorings.

- While some design flaws (*e.g.,Brain Method*, *Refused Parent Bequest*, *Code Duplication*) play a significant role in both systems, not all design flaws have the same negative impact in both systems. Moreover, the same flaw may have different evolution patterns in different systems.

- The diligence and the reactivity of refactoring actions show that the design flaws analyzed by our framework capture sensitive design characteristics, and that experienced developers considered them important enough to keep them under control.

- In the absence of a tool-supported framework, even experienced developers, such as those that work on JDT and EMF, have difficulties in addressing the symptoms of design debt in a systematic manner.

Assessing design symptoms of technical debt is *complex*. Therefore, the presented framework can provide both a coarse-grained perspective, to monitor the evolution of debt over time, and a more detailed perspective that enables locating and understanding individual flaws, which can lead in turn to a systematic refactoring. The proposed framework provides both. The framework achieves this by making design flaws *explicit*, which in return helps developers and quality engineers to connect them to design quality goals (*e.g.,* low coupling, tight encapsulation).

The provided assessment is *quantitative*, *informative* and *natural*, as it directly measures technical debt in terms of major design flaws that incur future evolution and maintenance costs. The framework is *useful*, because the case study showed that the framework characterizes quality aspects that the developers of the two systems have considered important. This is revealed by the fact that for all flaws, significant signs of refactoring actions have been noticed.

We have observed that the tool support offered by such a framework is *necessary*: without it, refactoring actions are not always coherent and systematic and their benefit can be neutralized by new instances of the same flaws. We are aware that debt symptoms can be also project-specific; for example, an application with no internationalization support that is acquired by a company for which internationalization is mandatory automatically incurs debt [KTWW11]. Assuming that detection rules for project-specific debt symptoms can be defined, the framework is flexible enough to accommodate them next to the general rules for design quality presented here. Furthermore, the framework is not aimed to measure or predict the actual effort or financial cost associated with technical debt [Fow09]. This is a highly challenging task, which needs to be addressed in the future.

Chapter 5

Automation of Design Assessment

Legacy systems tend to be extremely large, up to 10-20 million lines of code, and therefore the scalability of the proposed approaches is crucial. Thus, for performing quality assessments having the appropriate tools is a must. Because of this we put a particular emphasis on validating our ideas by building scalable tools.

5.1 Problem Statement

There are a number of commercial standalone tools like *Klocwork Insight* [Inc10], *Structure 101* [Sof10], *IBM Rational Quality Manager* [IBM10] as well as open-source tools like *PMD* [Cop05] or *Checkstyle* [Web10]. In spite of this apparent wealth of instruments for assessing maintenance issues, most of these tools don't go beyond the display of uncountable analysis results and metrics values and raising warning if some – oftentimes arbitrary – threshold values are exceeded. In the recent years countless code analysis tools support tasks as different as bug pattern identification [oM10, Cop05], clone detection [Ins10, Inc10] and dependency cycle analysis [Gmb10, Sof10].

All these tools, however, focus on very specific aspects of software quality and are, hence, not suited for a holistic quality assessment. Moreover, these tools provide sophisticated analysis techniques, but often fail to support quality engineers in interpreting the analysis results. In other words, such tools are providing low-level warnings that are not explicitly aggregated and connected to the high-level maintainability requirements (*i.e.*, he quality model).

To address this problem multiple dashboard tools like Sonar [Son10] or QALab [Con09b] are built on top of the specialized quality analysis tools, collecting, aggregating and visualizing data of metric calculators, and/or static analysis tools. Dashboards aim to provide a quality overview of a software system to monitor and control development activities. However, none of these tools establishes an explicit link between the specified quality requirements (*i.e.*, the quality model) and the actual quality characteristics of a software system (*i.e.*, the code analysis tools) [DHH$^+$11]. There are few notable tools that provide support for quality models, most of them the result of European or national research projects: SQUALE [LC09], Quamoco/ConQAT [DHH$^+$11] and Alitheia [Con09c]. SQUALE uses a fixed quality model, a fixed set of code analysis tools, and is limited to automated measure. Quamoco/ConQAT and Alitheia allow for project-specific customizations of quality models, mechanisms for a flexible configuration and integration of code analysis tools, as well as a seamless integration of results generated by manual analyses like inspections and reviews.

More recently a set of source code analysis environments have emerged, like MOOSE [DGN05], Sonar [Son10] or ConQAT [DPS05]. All of these are very mature, feature-rich tools which contain many innovative QA techniques. However, they are separated from the place where the code that they are analyzing is produced, namely the IDEs. Thus, there is an unfortunate separation between quality assessment and the actual development, and consequently a separation between the people who write the software and those who asses its quality. As a result, the feedback loop from code and reviews is very weak and inefficient, as developer will learn about the various design problems only from time to time (*i.e.,* after each code review), and usually only long after the code has been written. In result, this makes it very hard to solve all problems, because of the large number of problems that cumulate over time; and it's also hard to solve them efficiently because the context of that design fragment is probably long forgotten.

Because of this drawback, many of the aforementioned tools seek some form integration with the development process, either by providing means of integration in the build process [Inc10, Son10] and/or by integration in IDEs [Inc10, Cop05]. While the integration in the build process is definitely a step forward in the right direction — Sonar being a glorious [Son10] illustration of this category -– it still has the disadvantage that all problems are reported in a centralized manner and most of the times they lack providing the exact context of a design problem. Concerning the quality assessment tools that are integrated in IDEs, the biggest problem is that they level of integration is very shallow. Most of these are de facto standalone tools that lack any synergy [Zel07] with the other components of the development environment. In almost all cases the analyses have to be triggered by the developer and thus do not run continuously during development. In Eclipse one notable exception is the *Checkstyle* plug-in [Web10] that detects code-style violations by using a project builder, which means that whenever the build process is started the files are analyzed by *Checkstyle* as well. This plugin has the same continuous analysis approach like INCODE, however, the problems that it detects are more related to coding style (*e.g.,* code conventions) than to object-oriented design.

5.2 iPlasma: an Integrated Quality Assessment Platform

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-scale industrial projects of the size of millions of code lines (*e.g.,* Eclipse and Mozilla) [MMM^{+}05, LM06] [2].

Figure 5.1 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 us take a closer look at the layers of IPLASMA.

[1]Integrated PLAtform for Software Modeling and Analysis.

[2]This section is partially reproduced from [LM06], including all figures. ©Springer-Verlag Berlin Heidelberg 2006. Used by permission.

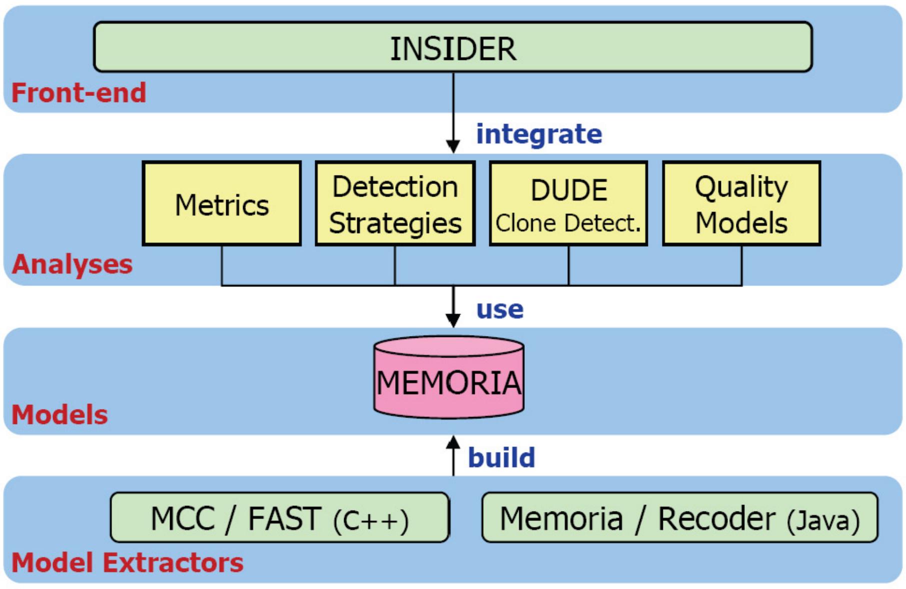


Figure 5.1: The layered structure of the IPLASMA quality assessment platform.

5.2.1 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 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[3] 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 of a MEMORIA model, the ASCII tables could be easily loaded in a RDBMS and interrogated in the form of SQL queries.

5.2.2 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. 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. In Chapter 3 we showed how *detection strategies* allow us to combine metrics in more

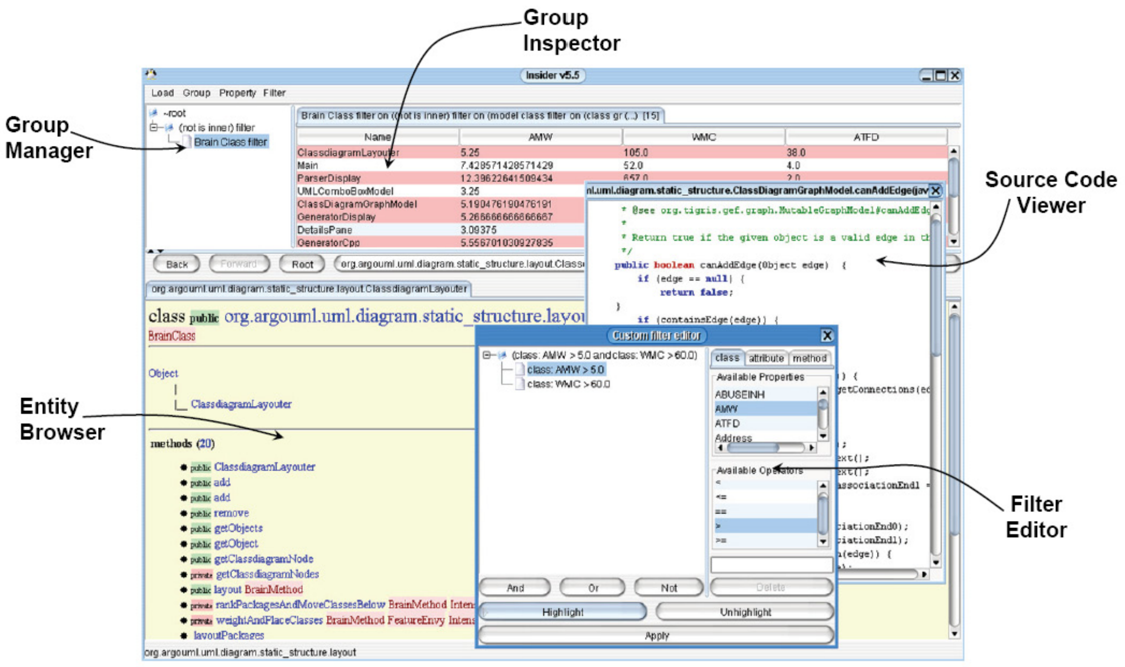
[3]See http://recoder.sourceforge.net/

complex rules for detecting design problems. In IPLASMA detection strategies can be implemented and adapted.

As we have seen in Section 3.4, 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).

5.2.3 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



Brain Class filter on ((not is inner) filter on (model class filter on (class gr (...) [15]

Name	AMW	WMC	ATFD
ClassdiagramLayouter	5.25	105.0	38.0
Main	7.428571428571429	52.0	4.0
ParserDisplay	12.39622641509434	657.0	2.0
UMLComboBoxModel	3.25		
ClassDiagramGraphModel	5.190476190476191		
GeneratorDisplay	5.266666666666667		
DetailsPane	3.09375		
GeneratorCpp	5.556701030927835		

nl.uml.diagram.static_structure.ClassDiagramGraphModel.canAddEdge(jav

```
     * @see org.tigris.gef.graph.MutableGraphModel#canAddEdg
     *
     * Return true if the given object is a valid edge in th
     */
    public boolean canAddEdge(Object edge)  {
        if (edge == null) {
            return false;
    }
        if (containsEdge(edge)) {
```

org.argouml.uml.diagram.static_structure.layout.ClassdiagramLayouter

class public org.argouml.uml.diagram.static_structure.layou
BrainClass

Object
ClassdiagramLayouter

methods (20)

- public ClassdiagramLayouter
- public add
- public add
- public remove
- public getObjects
- public getObject
- public getClassdiagramNode
- private getClassdiagramNodes
- public layout BrainMethod
- private rankPackagesAndMoveClassesBelow BrainMethod Inten
- private weightAndPlaceClasses BrainMethod FeatureEnvy Inten
- layoutPackages

(class: AMW > 5.0 and class: WMC > 60.0)
class: AMW > 5.0
class: WMC > 60.0

ABUSEINH
AMW
ATFD
Address

<
<=
==
>
>=

org.argouml.uml.diagram.static_structure.layout

Figure 5.2: 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 Figure 5.2) 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.

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 Figure 5.2), 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 Figure 5.2 the metrics are displayed for all the entities in the group.

- *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 Figure 5.2).

- *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.

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. More information about iPLASMA, including the possibility of downloading it, can be found at: http://loose.upt.ro/iplasma/

5.3 inCode: Continuous Quality Assessment

The typical usage scenario of a QA module/methodology is currently this: a developer, feeling that something is wrong with the design/code, is using the QA module provided by (or available for) her IDE to compute a suite of metrics; noticing some abnormal metric values, she must *infer* what the *real* design problem is from the informal description of the interpretation model of the metric. This is not easy at all, especially when the analysis occurs long after that code/design fragment has been created, and/or the code was written by someone else. But even after finding out what the problem is, correcting the design flaws moves the developer to another world, where she must compose the proper restructuring solution using the basic refactorings available in her IDE. This is again a challenging and painstaking operation. We believe that this process is so tedious because of two reasons: (i) metrics used to detect design flaws are only "detection atoms", and, therefore incapable of pointing out to relevant correction (restructuring) solutions; (ii) refactorings, as they are used know, are also only the "correction atoms", and therefore they don't represent the correction solution for all but non-trivial design problems.

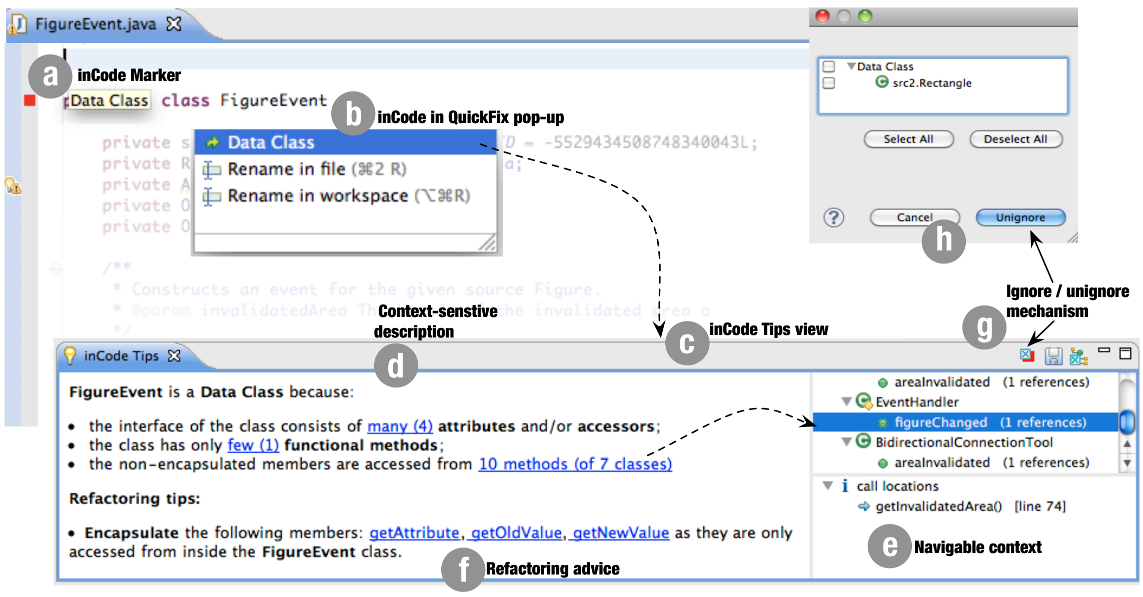
INCODE is an Eclipse plugin in which design problems are detected continuously. When the Eclipse workbench starts, INCODE also starts to analyze (in background) the source file currently active in the editor. When a design problem is detected, a red marker is placed on the ruler, next to the affected class or method (and also in the overview ruler on the right side of the editor). The inCode markers are similar to the markers used for compiler errors or warnings. The presence of these markers is very dynamic: as new code is written, or code is modified new markers may appear, or existing markers may disappear.

With INCODE we propose a novel type of QA tools, with the following (agile) “manifesto”:

- *Continuous detection and correction of design flaws*, instead of standalone, post-factum (or even “post-mortem”) code inspections;
- *Exposure of real design flaws*, instead of displaying abnormal metrics values
- *Contextual, problem-driven, tool-assisted restructuring strategies*, instead of a repetitive composition of restructuring solutions from atomic refactorings

Next we are going to briefly introduce the main features of INCODE *i.e.*, those that reveal its novelty.

5.3.1 Continuous assessment mode



```
class FigureEvent
private s                    D = -5529434508748340043L;
private R
private A
private O
private O
/**
 * Constructs an event for the given source Figure.
 * @param invalidatedArea The invalidated area o
 */
```

FigureEvent is a **Data Class** because:

- the interface of the class consists of many (4) **attributes** and/or **accessors**;
- the class has only few (1) **functional methods**;
- the non-encapsulated members are accessed from 10 methods (of 7 classes)

Refactoring tips:

- **Encapsulate** the following members: getAttribute, getOldValue, getNewValue as they are only accessed from inside the **FigureEvent** class.

```
areaInvalidated   (1 references)
▼ EventHandler
    figureChanged   (1 references)
▼ BidirectionalConnectionTool
    areaInvalidated   (1 references)
▼ i call locations
    getInvalidatedArea()   [line 74]
```

Figure 5.3: INCODE marks automatically, and updates continuously, code entities with design problems. The INCODE TIPS view can be used to better understand the causes, context and possible remedies for a particular instance of a design problem.

inCode Markers When the Eclipse workbench is started, INCODE begins to analyze in background the source file currently active in the editor. As seen in Figure 5.3, when a design problem is detected, INCODE places a *red marker* on the ruler (see[4] **a**), next to the affected class or method. As mentioned before, INCODE markers are similar to those used by Eclipse to indicate compiler errors or warnings. The presence of these markers is very dynamic: as new code is written, or code is modified new markers may appear, or existing markers may disappear.

[4]In this entire passage we will refer the various parts of Figure 5.3 by the corresponding labels depicted there

inCode Tips View By launching the *quick fix* for an INCODE marker (see **b**), the INCODE TIPS view (see **c**) is opened, providing a wealth of contextualized information on the particular cause (see **d**) of a detected problem. Although the design problems are detected using metrics-based rules [LM06], the description of the problem is in terms of design concepts, rather than numbers; in other words, the description is *hiding* the unnecessary complexity of working with software metrics. However, the description contains a set of hyperlinks that enable the developer to "zoom-in" in exploring in detail the relevant *context* of that problem (see **e**). Furthermore, the description may contain *significant correlations* of this problem with other design flaws detected by INCODE elsewhere in the project.

The other significant part of the INCODE TIPS view is the one that provides *concrete refactoring advices* (see **f**) *i.e.,* hints on how that particular problem instance can be corrected, by taking into account the *entire context of dependencies* of that class/method. Furthermore, if INCODE detects that in a given context a predefined Eclipse refactoring, or a composed restructuring (defined by INCODE) can be applied, the suggested code transformation can be triggered directly from the INCODE TIPS view. Currently, the set of refactorings that can be triggered automatically is limited, but we are working on extending both the *number of contexts* where INCODE can perform a design improvement, as well as the number of performable restructurings that involve the correlation of multiple atomic refactorings [Ver09].

Detected problems As mentioned before, we believe that metrics used in isolation cannot help in detecting *real* design problems [Mar04]. Therefore, in INCODE we use *detection strategies* to quantify design problems [LM06]. Currently INCODE detects four well-known design problems related to an improper distribution of intelligence among classes, namely *God Class*, *Data Class*, *Feature Envy* and *Code Duplication*. Thus, while the detection is based on object-oriented metrics, developers do not have to interact directly with metrics. Instead, they can reason about the quality of their design at the conceptual level that is more convenient for them.

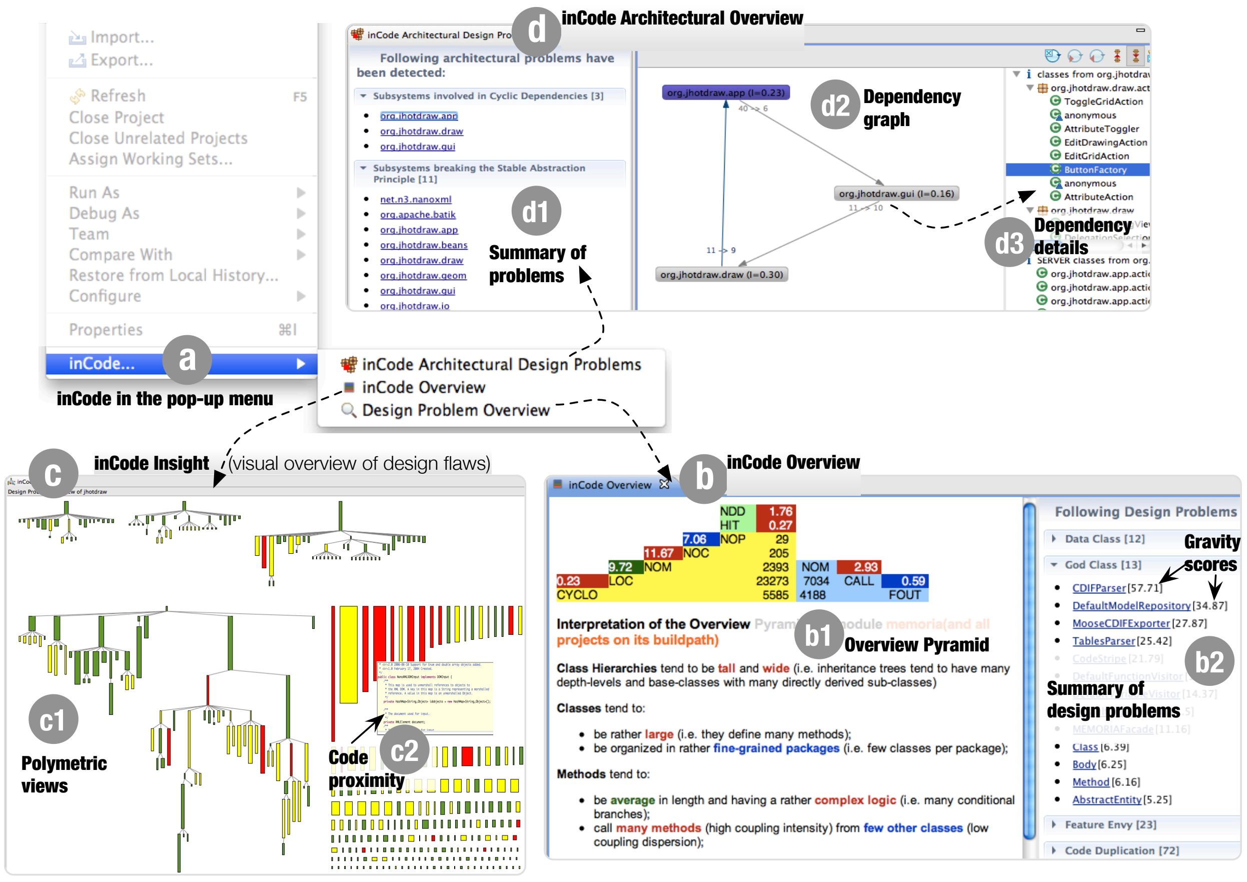
5.3.2 Global assessment mode

While the continuous assessment mode is what makes INCODE outstanding, we also need a way to "zoom-out" in order to see the global design quality picture for a system. A case where this is obviously needed, is the *first time* when we first approach an existing system; this is oftentimes the case, as we rarely start writing code from the scratch. Thus, an initial, global assessment is mandatory, before diving in the continuous mode. Furthermore, although the continuous assessment mode is aimed to distribute the responsibility of quality assessment among developers, from time to time, it is still needed to get a clear *overview* of the quality of a system.

INCODE addresses this need by providing a set of three views that reveal the overall quality of a system, namely (see[5] Figure 5.4): INCODE OVERVIEW (see **b**) and the INCODE INSIGHT view (see **c**), for summarizing the quality aspects of *detailed design*; and the INCODE ARCHITECTURAL OVERVIEW (see **d**) for summarizing the *high-level design* (architectural) issues. In terms of user interactions, these views are reached via the pop-up menus associated with the *Package Explorer* and *Outline* (see **a**) of Eclipse.

In this context it is worth mentioning that these overview analyses can be executed not only on the entire system, but also on a package level. Furthermore, INCODE

[5]In this entire passage we will refer the various parts of Figure 5.4 by the corresponding labels depicted there.



Following architectural problems have been detected:

Subsystems involved in Cyclic Dependencies [3]

- org.jhotdraw.app
- org.jhotdraw.draw
- org.jhotdraw.gui

Subsystems breaking the Stable Abstraction Principle [11]

- net.n3.nanoxml
- org.apache.batik
- org.jhotdraw.app
- org.jhotdraw.beans
- org.jhotdraw.draw
- org.jhotdraw.geom
- org.jhotdraw.gui
- org.jhotdraw.io

```
NDD   1.76
HIT   0.27
7.06 NOP   29
11.67 NOC   205
9.72 NOM   2393   NOM   2.93
0.23 LOC   23273   7034   CALL   0.59
CYCLO   5585   4188   FOUT
```

Interpretation of the Overview Pyramid of module memoria(and all projects on its buildpath)

Class Hierarchies tend to be tall and wide (i.e. inheritance trees tend to have many depth-levels and base-classes with many directly derived sub-classes)

Classes tend to:

- be rather large (i.e. they define many methods);
- be organized in rather fine-grained packages (i.e. few classes per package);

Methods tend to:

- be average in length and having a rather complex logic (i.e. many conditional branches);
- call many methods (high coupling intensity) from few other classes (low coupling dispersion);

Following Design Problems

Data Class [12]

God Class [13]

- CDIFParser [57.71]
- DefaultModelRepository [34.87]
- MooseCDIFExporter [27.87]
- TablesParser [25.42]
- CodeStripe [21.79]
- DefaultFunctionVisitor
- DefaultVariableVisitor [14.37]
- MEMORIAFacade [11.16]
- Class [6.39]
- Body [6.25]
- Method [6.16]
- AbstractEntity [5.25]

Feature Envy [23]

Code Duplication [72]

Figure 5.4: Global Assessment. The INCODEOVERVIEW and INCODEINSIGHT should be used as a starting point for a closer inspection. Architectural flaws can also be inspected using a dedicated view

supports the analysis of an entire Eclipse workspace (consisting of one or more Eclipse projects), as well as the analysis of a single project, both in isolation, and by taking into account its dependencies on other projects contained in the same workspace.

inCode Overview This view has two components: (i) the *Overview Pyramid* (see **b1**) that captures the key characteristics of the system/package in terms of complexity, coupling and shape of class hierarchies [LM06]; and (ii) a categorized list of detected design problems (see **b2**). From here, the problematic classes and methods can be inspected closer in order to understand for each cases the particular causes and the suggested correction steps. Due to the close integration with Eclipse, at any moment, the engineer has also direct access to the source-code of the problematic code fragment.

When it comes to low-level design problems, for large systems their number can be overwhelming. Therefore in INCODE we use for each of the four design problems a *gravity score* that indicate for each detected instance how severe the symptoms of the design problem are. As the detection of the design problem is based on metrics [LM06], the *gravity scores* are computed based on how much the various key measures are beyond the thresholds used in the detection rule. For example, a *Data Class* instance with 40 public attributes has a (significantly) higher gravity score than other instance with only 6 public attributes. In order to make the gravity scores comparable, each factor that enters that gravity score is normalized. Thus, the categorized lists of design problems are sorted descending based on the gravity score (see **b2**). This allows developers to prioritize the inspection and (hopefully) the correction of the most critical design fragments.

inCode Insight It is not always true that a picture is worth a thousand words, but it is certainly true that *well-designed* software visualizations can be efficient means to support developers in assessing quality, especially for understanding problems in context. Therefore, in INCODE we created the INCODEINSIGHT view (see **c**) which can display polymetric views [LD03]. In INCODE INSIGHT visualizations can be *explored*: developers can double-click figures to get the corresponding program entities (*e.g.,* class, method) opened in the editor; alternatively code can be viewed without losing the visual context, by means of the *hover code viewers* (see **c2**) that it supports.

The support of INCODE for polymetric visualizations is quite wide, but there is one visualization which proved to be especially useful in getting an overview of design problems: the *Design Flaws View*, which is an adaptation of the *System Complexity View* [LD03], in which classes are represented as rectangles, with the number of attributes determining the width, and the number of methods setting the rectangle's length. The point of our adaptation is the color: a rectangle is *green* if INCODE has not detected any design problems, it is *yellow* if only method-level problems have been detected, and it is *red* if class-level problems have been identified. The advantage of this view is that it allows an engineer to see how design problems are spread in the system, and especially to which extent problems are "clustered" in the various class hierarchies.

inCode Architectural Overview INCODE complements the detection of flaws at the level of *detailed design* with support for quality assessment at the architectural level. This is done by automatically detecting and providing significant contextual information about (see **d2** and **d3**) four well-known architectural problems, related to subsystem dependencies [Mar02c]. While architectural problems are also described using the INCODE TIPS view, there is one significant element that supports the understanding of all these problems: the visual representation of the package/subsystem-dependency graph (see **d2**). The graph reveals the various cycles in which a package is involved, but it also annotates the graph with additional information about the strength of a dependency, by displaying for each package its *Instability Factor* and for each dependency the number of *client* classes and the number of *server/provider* classes that are actually causing the dependency. All this information, provides a visual support for the understanding and solving of these four architectural problems. Furthermore, although these problems are considered to be "high-level" ones, INCODE helps exploring the problems down to the methods and classes that cause the dependencies (see **d3**).

We believe that INCODE and other future tools that will take the same approach will not only improve design and code; they will do the same with designers and developers because by treating the whole issue of design flaw detection in terms of violations of design principles, heuristics and best practices, engineers will be constantly confronted with these good design rules and will understand the real causes of a design problem rather than its symptoms (as revealed by plain metrics). Consequently, they will learn how to avoid them in the future. Furthermore, the controlled correction (restructuring) process will teach developers how to solve various design problems in a context-sensitive manner.

Chapter 6

Improvement of Software Design

There is no perfect software design. Like all human activities, the process of designing software is error prone and object-oriented design makes no exception. The flaws of the design structure, also known as “bad smells” [FBB$^+$99] have a strong negative impact on quality attributes such as flexibility or maintainability. Thus, not only the identification and detection of design flaws is important but also the correction of these flaws is essential for the improvement of software quality.

6.1 Problem Statement

In [CCI90], the authors define *restructuring* as *”the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior (functionality and semantics).”*. Thus, program restructuring is the transformation of the source code of a program that preserves its semantics and external behavior [Opd92]. The term *behavior preserving* means that program should produce the same externally observable behavior for any legal input after the refactoring was applied as it did before.

In [Rob99], the author argues that if timing constraints are part of the behavior, it becomes very difficult to argue behavior preservation. Other non-functional properties of a system, such as memory usage would be very hard to maintain in a refactoring. For this reason, in this diploma we do not consider the case of programs for which timing constraints or non-functional requirements are part of their specification. In order to be broadly used by practitioners, refactorings have to guarantee that they preserve the behavior of the system. One of the reasons that the structure of a system degrades is is the fear of touching something that works, even if the structure is no longer appropriate. Therefore, a tool supported approach to the refactoring process has to guarantee that its changes don’t alter the behavior of the system. But it is theoretically impossible for a refactoring technique to encompass all the programs that exhibit the same behavior. Existing work in the field of refactoring has either relied on a semi-formal demonstration of behavior preservation [Opd92], or no demonstration has been given at all [FBB$^+$99]. The refactorings proposed in [FBB$^+$99] and [Ker05] are behavior preserving more because of good engineering practice than because of any formal proof.

Composite Refactorings Even the simplest refactorings can be hard to implement. For example encapsulating a field is not just changing its modifier into *private*, but there can be references to that field and their scope is the whole system, thus these references have to be updated too. Thus, a methodology of describing high-level refactorings based on ***composition*** has been proposed in [OCN99], and reemphasized by Kerievsky [Ker05].

Composite refactorings are also good candidates for automation. Therefore, in our work [LM06, Ver09] we use this approach of *composing* a suite of code transformations into a high-level refactoring. The use of composite refactorings has several advantages [MD03]: (i) they are better at capturing the design intent of the software change introduced by the refactoring, and thus it becomes easier to understand how the software has been refactored; (ii) using composite refactorings represents a performance gain as the tool has to check the preconditions only once for the composite refactoring, rather than for each primitive transformation in the refactoring sequence; (iii) composite refactorings weaken the behavior preservation for their constituents. The primitive transformations in a sequence don't have to preserve the behavior, as long as the net effect of their composition is behavior preserving.

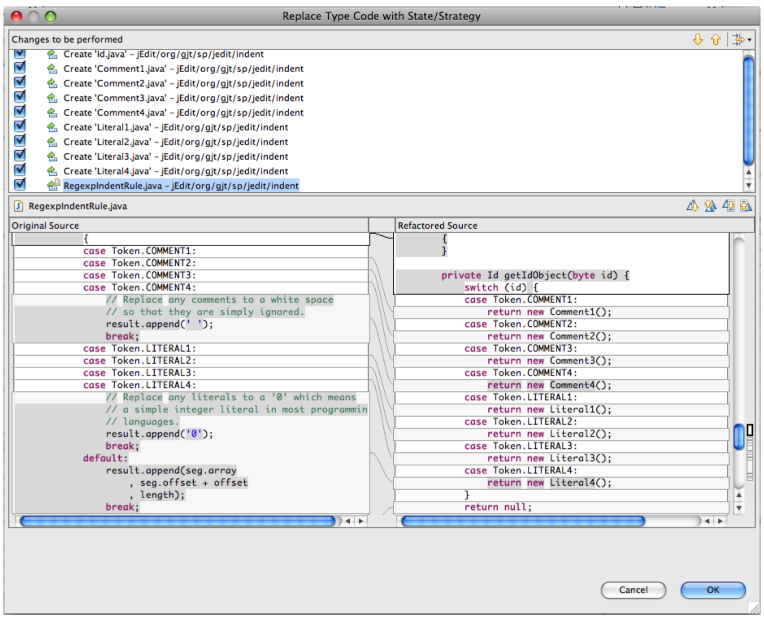
Refactoring Tools Today, a wide range of tools are available to automate various aspects of refactoring. Depending on the tool and the kind of support that is provided, the degree of automation varies. Tools as the *Refactoring Browser*[RBJ97] and *XRefactory* [Vit03] support a semi-automatic approach. A fully automated tool for restructuring inheritance hierarchies and refactoring methods in SELF programs is *Guru* [Moo96]. Given a collection of Self objects, *Guru* produces an equivalent set of objects in which there is no duplication of methods or certain types of expressions.

There is also a tendency of integrating refactoring tools directly into software development environments. This is the case for *Eclipse*, *IntelliJ IDEA*, *Borland JBuilder*, etc. These tools apply a refactoring upon request of the user. There is much less support for detecting where and when a refactoring can be applied. [SSL01] proposes to do this by means of metrics, while [KEGN01] indicates where refactorings might be applicable by automatically detecting program invariants using the *Daikon* tool.

A fully automated tool for identifying and removing bad smells in code is *jDeodorant* [plu]. The methodology used by this tool consists of two parts: the first deals with the identification of type-checking bad smells. The second concerns their elimination by applying appropriate refactorings. Currently, jDeodorant identifies three bad smells: Feature Envy, Type Checking and Long Method. Feature Envy is corrected with a Move Method, Type Checking is corrected either by employing *Replace Conditional with Polymorphism* or *Replace Type Code with State/Strategy*, while Long Method is corrected using Extract Method. After analyzing a system, the plugin presents a view which for example for Type Checking contains information about the Suggested refactoring type, the Abstract Method Name that will be created, the number of System Level Occurrences, the number of Class-Level Occurrences and the Average number of statements per case. The refactorings that jDeodorant proposes are tightly integrated in the Eclipse platform, but they do not favor composition. An example refactoring supported is depicted in Figure 6.1.

The refactorings offered by jDeodorant are high-level refactorings that do all the work that is needed in just one single big step, thus no decisions can be made at runtime and smaller transformations cannot be reused. But the fact that the plugin bridges the gap between problem detection and correction is highly valuable.

The Gap Between Flaw Detection and Correction In spite of this progress, restructuring large object oriented systems still remains a predominantly manual, time-consuming, risky process, that involves extensive (and costly) human expertise (both domain-specific and technical). In other words, restructuring is still an art, rather than engineering, an aspect which is underlined by studies [Fea05, Pre10] showing



```
Create 'Id.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Comment1.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Comment2.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Comment3.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Comment4.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Literal1.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Literal2.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Literal3.java' – jEdit/org/gjt/sp/jedit/indent
Create 'Literal4.java' – jEdit/org/gjt/sp/jedit/indent
RegexpIndentRule.java – jEdit/org/gjt/sp/jedit/indent
```

```java
            {
            case Token.COMMENT1:
            case Token.COMMENT2:
            case Token.COMMENT3:
            case Token.COMMENT4:
                // Replace any comments to a white space
                // so that they are simply ignored.
                result.append(' ');
                break;
            case Token.LITERAL1:
            case Token.LITERAL2:
            case Token.LITERAL3:
            case Token.LITERAL4:
                // Replace any literals to a '0' which means
                // a simple integer literal in most programmin
                // languages.
                result.append('0');
                break;
            default:
                result.append(seg.array
                    , seg.offset + offset
                    , length);
                break;
```

```java
        {
        }

        private Id getIdObject(byte id) {
            switch (id) {
            case Token.COMMENT1:
                return new Comment1();
            case Token.COMMENT2:
                return new Comment2();
            case Token.COMMENT3:
                return new Comment3();
            case Token.COMMENT4:
                return new Comment4();
            case Token.LITERAL1:
                return new Literal1();
            case Token.LITERAL2:
                return new Literal2();
            case Token.LITERAL3:
                return new Literal3();
            case Token.LITERAL4:
                return new Literal4();
            }
            return null;
```

Figure 6.1: Refactoring Wizard for the Replace Type Code with State/Strategy

that approximately 50% of maintainer time is spent exclusively on understanding the code.

We believe that the major cause for this situation is the large gap that exists in current IDEs between the automatic *detection* of significant design problems (*i.e.*, the *quality assessment* modules) and the automatic (or at least tool-supported) *correction* of those problems (i.e. the *refactoring/restructuring* modules). Detection and correction of design flaws are unfortunately still two separated worlds.

The typical usage scenario of a quality assessment module (and/or methodology) is currently this: a developer, feeling that something is wrong with the design/code, is using the QA module provided by (or available for) her IDE to compute a suite of metrics; noticing some abnormal metric values, she must *infer* what the *real* design problem is from the informal description of the interpretation model of metrics. This is not easy at all, especially when the analysis occurs long after that code/design fragment has been created, and/or the code was written by someone else. But even after finding out what the problem is, correcting the design flaws moves the developer to another world, where she must compose the proper restructuring solution using the basic refactorings available in her IDE. This is again a challenging and painstaking operation. We believe that this process is so tedious because of two reasons: (i) metrics used to detect design flaws are only "detection atoms", and, therefore incapable of pointing out to relevant correction (restructuring) solutions; (ii) refactorings, as they are used now, are also only the "correction atoms", and therefore they do not represent the correction solution for all but non-trivial design problems.

6.2 Correction Plans

As mentioned earlier, after a problematic design fragment is detected, the next step is to restructure the system, by removing the detected flaw. In most cases, it is not enough to apply a single basic refactoring (that is usually available via a refactoring plugin, *e.g., Move Method*, *Push-Up Field* [FBB+99]; instead, an entire *sequence* of such refactorings is needed. These complex restructurings depend on two main factors: (i) the design flaw itself and (ii) further contextual information concerning the state of the system.

These contextualized correction solution must be be defined for each design flaw that needs to be removed. In this context, in [TM05] we define *correction plans* as "*precisely defined procedures that describes a sequence of operations (basic refactorings) that need to be carried out in order to eliminate an instance of a given design problem*". These *correction plans* can be best imagined as an activity diagram, or a logical schema, composed mainly of decision and action blocks (see Figure 6.2).

In [LM06] we took a first step by defining concrete *correction plans* for improving object-oriented design with respect to its three main aspects: *Complexity* (*identity flaws*), *Coupling* (*collaboration flaws*) and the use of *Inheritance* (*classification flaws*). Each of these are described in terms of the design flaws defined in [LM06][1].

Correction Plan for Complexity Flaws We will illustrate this principle of *correction plans*, by using the case of recovering from *Complexity* design flaws. The first step in doing this is to identify the "intelligence magnets", *i.e.*, those classes that tend to accumulate much more functionality than an abstraction should normally have. In terms of *detection strategies*, this means to make a blacklist containing all classes affected by the *God Class* or by the *Brain Class* flaws. For each of the classes in the blacklist one has to find the *flawed methods*. A method is considered flawed if at least one of the following is true: (i) it is a *Brain Method*; (ii) it contains duplicated code; (iii) it accesses attributes from other classes, either directly or by means of accessor methods.

Assuming that for a class in the blacklist we have gathered its flawed methods, then in order to improve the design we have to follow the roadmap described in Figure 6.2, and explained briefly below.

Action 1: Remove duplication. The first thing to be done is to check if a method contains portions of *Code Duplication* and remove that duplication. Because we analyze the class from the perspective of complexity flaws 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* and *God Class* 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 modeled 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 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 (Access To Foreign Data) [LM06]

[1]This section is partially reproduced from [LM06], including all figures. ©Springer-Verlag Berlin Heidelberg 2006. Used by permission.

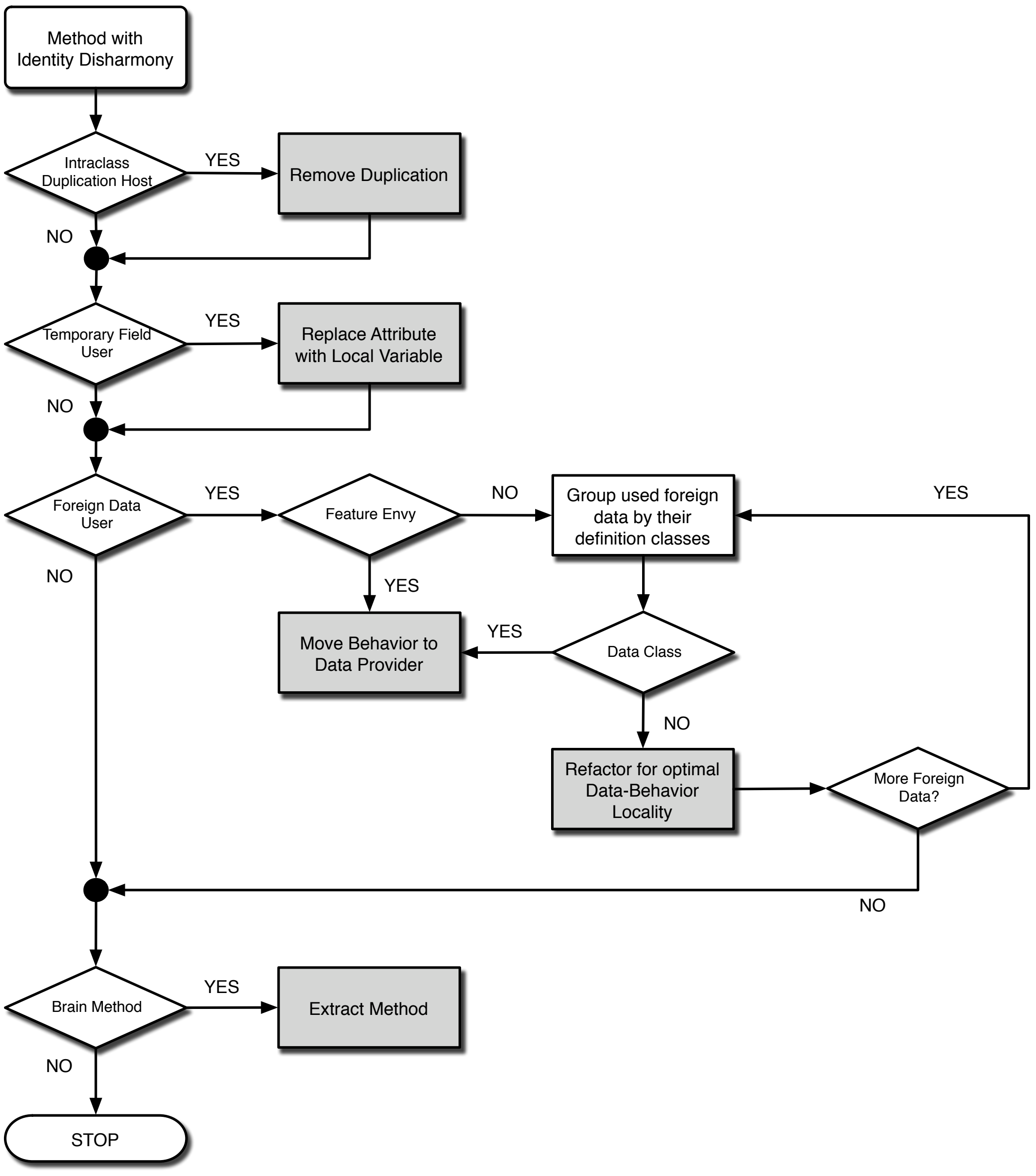

Method with Identity Disharmony
Intraclass Duplication Host
YES
Remove Duplication
NO
Temporary Field User
YES
Replace Attribute with Local Variable
NO
Foreign Data User
YES
Feature Envy
NO
Group used foreign data by their definition classes
YES
YES
Move Behavior to Data Provider
YES
Data Class
NO
Refactor for optimal Data-Behavior Locality
More Foreign Data?
NO
NO
Brain Method
YES
Extract Method
NO
STOP

Figure 6.2: Restructuring in order to remove complexity flaws

is at least one. In a simplistic way we can say that refactoring in this case requires one of these two actions:

1. *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* and the class that provides the attributes is a *Data Class*. In this case the method was simply misplaced, and needs to be moved to the *Data Class*. 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. As a rule of thumb that we often do this: if the class that provides the accessed attributes is "lightweight" (*i.e.,Data Class* or close to it) try to extract fragments of functionality from the "heavyweight" class and move them to the "lightweight" one.

2. *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* and *God Class*. 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 Methods.* If you reached this step while inspecting a method that was initially reported as a *Brain Method*, first look if this is still the case after proceeding with *Action 1* and *Action 3*. Sometimes, removing duplication and refactoring a method for better data-behavior locality solves the case of the *Brain Method*.

6.3 Correction Strategies

Methodologically, *correction plans* are a significant step forward in the process of performing complex restructurings. However, they are not specified sharp enough to enable their *automation*. Therefore, in the diploma thesis of Verebi [Ver09] we made a first step in moving beyond correction plans, and proposing tool-supported high-level correction solutions for individual flaws *i.e., correction strategies*. In concrete terms, we defined a *correction strategy* for a well-known design flaw, namely *Feature Envy* [FBB+99]. This restructuring extension plugs into the mechanisms of INCODE for detecting and signaling a *Feature Envy* and suggests the appropriate correction strategy (see Section 5.3). As we will see in Section 6.4 the integration with INCODE is not accidental, as our goal is to connect at the level of tools the detection and correction of design flaws.

6.3.1 Detection Strategy

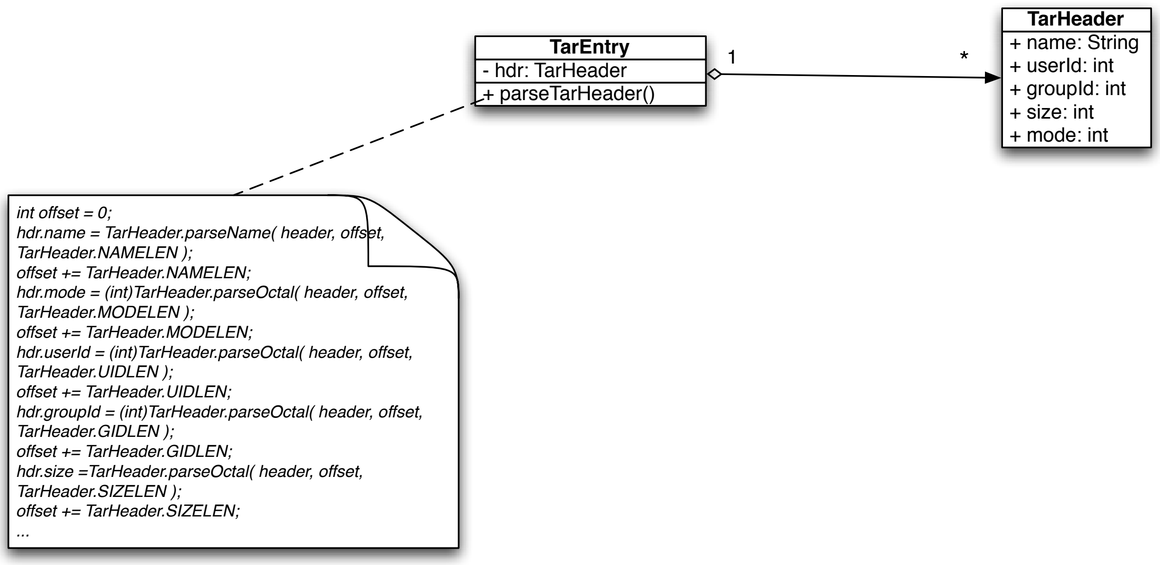
Feature Envy is a sign of violating the principle of grouping behavior and data together and refers to a method that seems more interested in a class other than the one it actually belongs to [FBB+99]. Te focus of the envy is the data. But this does not mean that the class is only using the data directly, but also through accessor methods. We are not interested in methods that use data from a a large number of classes, but methods that are interested in few (one or two) classes. This is because this is a suggestion that the affected method is *misplaced*, and not that it acts as a controller [Rie96] or that it is a *brain method* [LM06].

The detection process counts the number of attributes belonging to external classes, accessed either directly, or through accessor method. The *detection strategy* proposed in [LM06] requires that these three conditions are met:

1. The method uses more than a few attributes of other classes. This translates into a query on the ATFD and then a comparison with a predefined threshold.

2. The number of attributes accessed from an external class is in a much greater proportion than the number of attributes that the method accesses from its owning class. This rule is quantified by the LAA (Locality of Attribute Accesses) metric.

3. The last requirement is that there are very few classes into which the method is interested in. This requires to calculate the FDP (Foreign Data Providers) metric and compare it to a predefined threshold.



```
int offset = 0;
hdr.name = TarHeader.parseName( header, offset,
TarHeader.NAMELEN );
offset += TarHeader.NAMELEN;
hdr.mode = (int)TarHeader.parseOctal( header, offset,
TarHeader.MODELEN );
offset += TarHeader.MODELEN;
hdr.userId = (int)TarHeader.parseOctal( header, offset,
TarHeader.UIDLEN );
offset += TarHeader.UIDLEN;
hdr.groupId = (int)TarHeader.parseOctal( header, offset,
TarHeader.GIDLEN );
offset += TarHeader.GIDLEN;
hdr.size =TarHeader.parseOctal( header, offset,
TarHeader.SIZELEN );
offset += TarHeader.SIZELEN;
...
```

Figure 6.3: Feature Envy example

The example depicted in Figure 6.3 illustrates two classes, namely *TarEntry* and *TarHeader*. The method *parseTarHeader()* **envies** the data from TarHeader and uses this data directly to provide a functionality that should have been available in the *TarHeader* class.

6.3.2 Suggested Correction Plans

Feature Envy problems can be corrected in three ways [FBB$^+$99]: *(i)* by moving the method to the class that it envies, *(ii)* by extracting only a fragment of the method and then moving it to the class that it envies and *(iii)* by moving one or more attributes to the class that the feature envy method belongs to. The correct application of any of these methods in a system clearly improves its design quality, without altering its external behavior. These three methods to correct a *feature envy* are in accordance to one of these two principles: *move behavior close to data* or *move data close to behavior*.

The solution that we defined is to move a method to a class that it envies. The solution of moving a field has not been considered because usually fields have stronger conceptual binding to the classes in which they were initially placed [TC].

To move the method to one of the classes that it envies, only one *precondition* must be met: there must be *one and only one envied class*. This is because if there is more than one class, an accurate decision of where to move the method cannot be made. Even if moving a class to one of the envied classes will remove the feature envy smell of the class, an assessment that the overall structure of the system has improved cannot be made accurately. Thus, the proposal to move the method will be made only if there is exactly one envied class. Oftentimes, the class that the feature envy method depends upon is a class with next to no functionality, sometimes even a *data class*. If this is the case, moving the method will rebalance the overall distribution of responsibilities in the system and will improve the data-behavior locality [LM06].

The target class in which the method is to be *moved* is the envied class. Once the refactoring has successfully been executed, not only our method is no longer a *feature envy*, but it is possible that another problem is solved too: if the envied class was a *data class*, and the refactoring identified opportunities when it can encapsulate some of its members after the method was moved (members that were used only by the moved method), it is possible that the class would no longer be a data class.

6.3.3 Implementation of the Correction Strategy

INCODE accepts correction proposals from other plugins that connect to it through an *extension point*, namely *inCode.correctionStrategy*. This extension point has three attributes:

1. the class that executes the restructuring (this class must implement the *ICorrectionStrategy* interface),
2. an identifier for the design problem that it fixes, so that INCODE can do the mapping between the two, and
3. an identifier of the correction plan.

For example, the *Move Method* correction strategy is depicted in Listing 6.1. In this case, the correction proposal is implemented in class *MoveMethodCorrectionStrategy*, the identifier of the design problem is `Feature Envy` and the identifier of the correction strategy is `Move Method`.

Listing 6.1: The extension point for the Move Method correction strategy

```
<extension point="inCode.correctionStrategy">
   <correctionStrategy
      class="com.intooitus.refactoring.featureEnvy.MoveMethodCorrectionStrategy"
      message="Feature Envy"
      name="Move Method">
   </correctionStrategy>
</extension>
```

The interface *ICorrectionStrategy* consists of four methods:

- RefactoringStatus execute()

  This method executes the refactoring and returns a *RefactoringStatus* which can be used to find the outcome of the operation. Additionally, the RefactoringStatus class manages the problem severity. Problem severities are ordered as follows: *OK* < *INFO* < *WARNING* < *ERROR* < *FATAL*. If the status does not have an entry, then the default status severity is returned (*i.e., OK*).

- void setEntity(AbstractEntityInterface entity)

  This method tells the correction strategy which is the entity on which it will execute. A setter method is needed because extension points are registered statically, and we cannot know at the moment they are registered which is the problematic entity.

- RefactoringStatus checkPreconditions()

  This method will check that the preconditions specified in the previous section for each design flaw are met in order to proceed with the refactoring. This methods returns a status that can be later checked to see if the refactoring can be performed (using RefactoringStatus#isOK()).

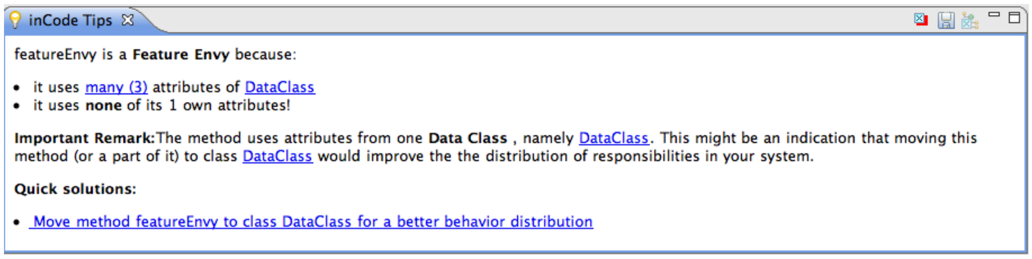
- String getMessage() This method returns the message that will appear in *Tips View*. INCODE will display this message and if the preconditions are met, this message can be selected by the user and thus the refactoring is triggered.

Next, we will describe the implementation of a concrete correction strategy, namely *Move Method* to correct a *Feature Envy*. The code of the specified correction strategy is the one in listing 6.2. In the remainder of this section we will explain this class in detail.

Listing 6.2: Move Method correction strategy

```java
public class MoveMethodCorrectionStrategy implements ICorrectionStrategy {

  public static final String MOVE_METHOD = "Move method";
  private AbstractEntityInterface entity;
  private GroupEntity foreignDataProvidersGroup;
  private FeatureEnvyRefactoring refactoring;

  public void setEntity(AbstractEntityInterface entity){
    this.entity = entity;
    foreignDataProvidersGroup = entity.getGroup("FDPG");
  }

  public RefactoringStatus execute(){
    new RefactoringAlgorithmWizard(refactoring,
                "Move Method Correction Strategy");
    return refactoring.execute();
  }

  public RefactoringStatus checkPreconditions() {
    refactoring = new FeatureEnvyRefactoring(entity);
    IPrimaryElement initialConditions = refactoring.checkInitialConditions();
    return initialConditions.execute();
  }

  public String getMessage() {
    return MOVE_METHOD+" "+entity.getName()+" to class "+
        foreignDataProvidersGroup.getElementAt(0).getName()+" for a better
        behavior distribution";
  }
}
```

Method *checkPreconditions()* creates the object that executes the actual refactoring. This object subclasses the interface *IRefactoring* that we've implemented, which means it can check initial conditions (through IRefactoring#checkInitialConditions()), it can create a refactoring algorithm (through IRefactoring#createRefactoringAlgorithm()) and can execute that refactoring (IRefactoring#execute()). After creating the refactoring object, it will delegate the initial conditions check to the *checkInitialConditions()* method of this object. This method is the one that actually checks if the refactoring can be safely executed. If the outcome of this check is *valid*, then the message returned from the method *getMessage()* will appear in Tips View. This will look like in figure 6.4.



featureEnvy is a **Feature Envy** because:

- it uses many (3) attributes of DataClass
- it uses **none** of its 1 own attributes!

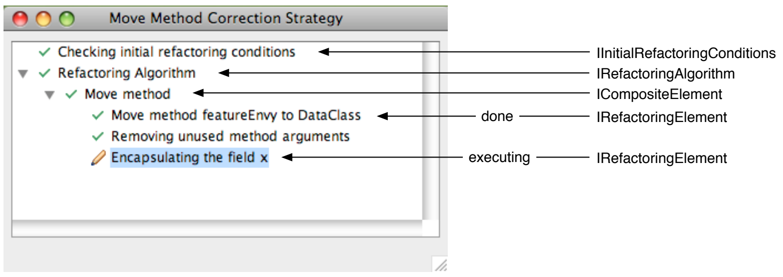
Important Remark: The method uses attributes from one **Data Class** , namely DataClass. This might be an indication that moving this method (or a part of it) to class DataClass would improve the the distribution of responsibilities in your system.

Quick solutions:

- Move method featureEnvy to class DataClass for a better behavior distribution

Figure 6.4: Tips View for a Feature Envy

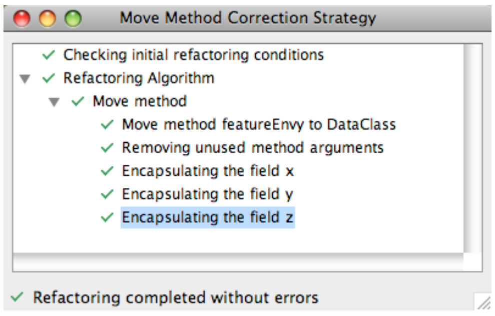
When the user chooses to start the refactoring, he can click on the link that is available in Tips View. This is the moment when the *execute()* method is called. This method opens an application window to show the user the progress of the refactoring. This wizard is consistent with the Eclipse user interface guidelines [LEHP06] and looks the same for any restructuring that we've implemented. An example wizard is depicted in figure 6.5. This screenshot is taken during the execution of the refactoring.



✓ Checking initial refactoring conditions
▼ ✓ Refactoring Algorithm
▼ ✓ Move method
✓ Move method featureEnvy to DataClass
✓ Removing unused method arguments
Encapsulating the field x

Figure 6.5: Refactoring wizard during execution

As it can be seen from the picture, the refactoring process is based on object composition, as a way to treat uniformly the execution process: a refactoring algorithm (IRefactoringAlgorithm) consists of several composite elements (ICompositeElement), which in turn are made up of a series of refactoring elements (IRefactoringElement). When the *execute()* method is called, this delegates execution to its components and so on. As it can be seen from the figure, the user knows at each moment what is executing. A solution that would show the user the next steps that are to be executed in advance could not be implemented as there are decisions involved in executing a refactoring and they are made at runtime. After the refactoring is done executing, it will show the status of the refactoring in the status of the wizard.



✓ Checking initial refactoring conditions
▼ ✓ Refactoring Algorithm
▼ ✓ Move method
✓ Move method featureEnvy to DataClass
✓ Removing unused method arguments
✓ Encapsulating the field x
✓ Encapsulating the field y
✓ Encapsulating the field z

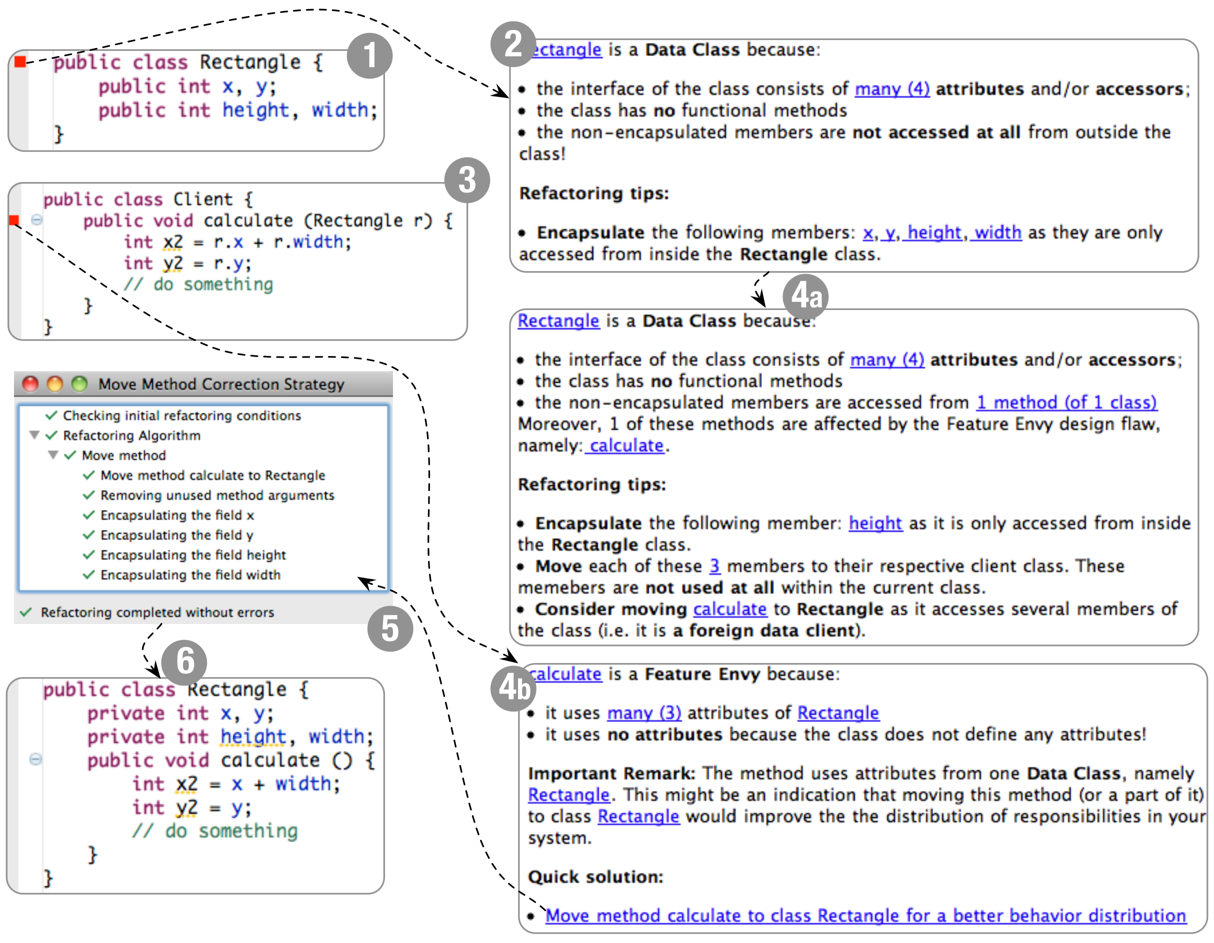
✓ Refactoring completed without errors

Figure 6.6: Refactoring wizard after the refactoring is done executing

6.4 Continuous Detection and Correction by Example

The works of Kerievsky [Ker05] and Feathers [Fea05] point out imperatively towards focus our attention on higher-level, composite restructuring activities, that need to be supported by tools. Thus, as mentioned in the previous section, as part of INCODE (see Section 5.3) we developed a module that complements the problem detection features with an essential correction component. This component is envisioned to act as a "wizard" that guides the developer through a contextualized *correction strategies* that will be defined for each design flaw that needs to be removed.

In this section we will illustrate by a simple yet significant example the capacity to automate in a continuous manner the detection of design problems, as well as its context-sensitive support for automatic restructuring.



```
public class Rectangle {
    public int x, y;
    public int height, width;
}
```

ectangle is a **Data Class** because:

- the interface of the class consists of many (4) **attributes** and/or **accessors**;
- the class has **no** functional methods
- the non-encapsulated members are **not accessed at all** from outside the class!

Refactoring tips:

- **Encapsulate** the following members: x, y, height, width as they are only accessed from inside the **Rectangle** class.

```
public class Client {
    public void calculate (Rectangle r) {
        int x2 = r.x + r.width;
        int y2 = r.y;
        // do something
    }
}
```

Rectangle is a **Data Class** because:

- the interface of the class consists of many (4) **attributes** and/or **accessors**;
- the class has **no** functional methods
- the non-encapsulated members are accessed from 1 method (of 1 class)

Moreover, 1 of these methods are affected by the Feature Envy design flaw, namely: calculate.

Refactoring tips:

- **Encapsulate** the following member: height as it is only accessed from inside the **Rectangle** class.
- **Move** each of these 3 members to their respective client class. These memebers are **not used at all** within the current class.
- **Consider moving** calculate to **Rectangle** as it accesses several members of the class (i.e. it is **a foreign data client**).

calculate is a **Feature Envy** because:

- it uses many (3) attributes of Rectangle
- it uses **no attributes** because the class does not define any attributes!

Important Remark: The method uses attributes from one **Data Class**, namely Rectangle. This might be an indication that moving this method (or a part of it) to class Rectangle would improve the the distribution of responsibilities in your system.

Quick solution:

- Move method calculate to class Rectangle for a better behavior distribution

Move Method Correction Strategy

✓ Checking initial refactoring conditions
✓ Refactoring Algorithm
  ✓ Move method
    ✓ Move method calculate to Rectangle
    ✓ Removing unused method arguments
    ✓ Encapsulating the field x
    ✓ Encapsulating the field y
    ✓ Encapsulating the field height
    ✓ Encapsulating the field width

✓ Refactoring completed without errors

```
public class Rectangle {
    private int x, y;
    private int height, width;
    public void calculate () {
        int x2 = x + width;
        int y2 = y;
        // do something
    }
}
```

Figure 6.7: The various steps of using INCODE for design assessment, going from writing a piece of code to the eventual restructuring.

The scenarios starts with Lisa, a typical developer, beginning to write the Rectangle class (see Figure 6.7[2] - **Step 1**) in the editor window of Eclipse. The very moment she finishes writing the 4 lines of the Rectangle class and saves the file, a red square – which is an INCODE *marker* – appears on the left side ruler of her editor, on the line where the class definition starts. This notifies her that INCODE has detected a potential design problem. By reading the code, you might have noticed that Rectangle is a class that defines four public attributes, opening the gates for breaking encapsulation. This problem is known in the literature as *Data Class* design flaw [FBB+99, Rie96]. So, every time Lisa is changing the file, on save INCODE executes its metrics-based detection rules (see Section 3.2) and displays a red marker for each each design flaw that has been detected in that source file.

By noticing the red INCODE marker, Lisa wants to find out what the problem is. Because INCODE markers behave exactly like the standard Eclipse ones which signalize compiler errors or warnings so it comes natural to Lisa to click on it (**Step 2**) and find out that indeed the *Data Class* problem has been detected.

[2]The sequence of steps in our example are summarized in form of numerical labels (which we are going to refer to in the following as **Step X**)

Next, when Lisa decides to find out more about the problem, the INCODE TIPS view is opened and she can read a detailed description of what the problem is. One important thing here is that the description is not a presentation of what *Data Class* means in *general*, but an explanation of why Rectangle, *in particular*, is reported by INCODE as a *Data Class* (see text box next to **Step 2**). By reading the text you will also notice that INCODE TIPS includes a section of *Refactoring Tips*, which in this particular case advises Lisa to encapsulate the four attributes of Rectangle because they have no reason to be declared public (as no one is using them from outside the class). Again, the noteworthy aspect here is the *context-sensitive nature* of the refactoring advice.

Finally, there in another thing about this description that needs to be emphasized: while both the detection of the problem and the additional description are based on a significant amount of metrics and further dependency analysis, Lisa is not required to have an understanding of these in order to use INCODE. Lisa does not even need to know exactly what a *Data Class* is. We believe that this is an essential trait for any QA tool in order to facilitate a wide adoption by developers: it has to hide the complexity of the powerful analysis techniques that it uses under a presentation that is easy to understand by developers. It is our tools that have to learn the language of developers, not vice-versa.

Now, assume that after reading the description of the problem, Lisa decides to momentarily ignore it and move on, writing class Client that uses the data members exposed by Rectangle (**Step 3**). Again, a red marker appears left to the line where the definition of method calculate starts. This is because calculate is affected by the *Feature Envy* design problem [FBB+99], and INCODE detects it based on the detection strategy defined in [LM06].

The first thing that Lisa may notice is that the description of the Rectangle *Data Class* has been updated (**Step 4a**), and if Lisa didn't close the INCODE TIPS view, the update occurs automatically; the description now remarks the usage of Client.calculate using the data of Rectangle. But, the most remarkable change is the one that occurs in the *Refactoring Tips* part, as now, by the fact that three public members of Rectangle are used from a *single* external class, INCODE can give different refactoring advices (see description in **Step 4a**); the initial refactoring suggestion is maintained for the height data member, but for the others, due to the new usage context, there are two options: moving the three data members to the Client class or vice-versa, move the calculate method to the Rectangle class.

Beyond the update of the Rectangle problem description, there is one even more interesting thing for Lisa to see: the description of the *Feature Envy* problem detected for calculate (**Step 4b**). Apart from the characteristics already emphasized while presenting the description of the Rectangle *Data Class* (*i.e.*, context-sensitivity, continuous update, refactoring advices), there are two additional aspects of INCODE TIPS revealed here: (i) the description contains a remark on the fact that the "foreign" data used by calculate are defined in a class that has been detected as *Data Class*; (ii) it has an additional refactoring section, called *Quick Solution* indicating that INCODE has detected an actual refactoring that Lisa can launch in order to solve the *Feature Envy* problem. In other words, INCODE TIPS supports the design improvement by (i) providing information about meaningful *correlations* detected between various design flaws and (ii) by identifying cases where "clear-cut" restructuring solution exist and consequently by providing support for automating the code transformation process.

If Lisa decides to solve the problem by moving calculate to Rectangle, she selects the link below *Quick Solution* which, in result, will start the restructuring process (**Step 5**). As seen in Figure 6.7 the restructuring process consists of a sequence of basic refactorings, which lead to a significantly better solution (**Step 6**) than a simple restructuring like the built-in *Move Method* refactoring in Eclipse.

In conclusion, with INCODE we took the first steps towards bridging the existing gap between the detection and the correction of design problems by:

- creating tool-supported techniques for describing and executing contextual, problem-driven restructuring strategies, that should relieve developers from manually composing restructuring solutions based on atomic refactorings;
- synchronizing the detection and correction activities, so that most design problems become convenient to correct immediately after they are detected.

We believe that this approach will not only reduce significantly design degradation and maintenance costs, but it will also improve continuously the design skills of object-oriented programmers.

Part II

Future Plans

Chapter 7

Design Assessment and Improvement: The Challenges

We plan to grow our research on several key directions. First we will continue the work on better connecting assessment and correction activities. We also plan to investigate how quality assessment can be efficiently applied to *hybrid software systems* which mix different programming languages or even paradigms. Another plan is to conduct a broad empirical validation of assessment techniques, by creating a comprehensive metrics benchmarking methodology, operationalized by proper tools. We aim to perform this validation on at least 10.000 projects. The analysis results will be used to calibrate the quality assessment techniques and to detect potential inconsistencies in quality models.

7.1 Closing the Gap Between Flaw Detection and Correction

7.1.1 Impact Analysis of Correction Strategies

The direct modification of the source code is an expensive task, especially due to the fact that a software system is not composed only from its source code and when a modification is made on the source code, all the other parts of the system should be updated, too (*e.g.,* tests, diagrams, documentation). Additionally, after all the modifications have been done, a new quality assessment is required in order to find out if, and at which extent, the restructurings have really improved the quality of the system.

Most of the times design flaws don't exist in isolation (see Figure 3.2) and therefore, correcting one problem might have a positive or/and negative impact on other parts of the system. For example, correcting the case of a method affected by *Feature Envy* might also lead to the disappearance of a *Data Class*, from which initially the method was directly accessing data, as discussed in Section 6.3.3. This example is a rather happy case. But it might also happen that a modification, usually an expensive one, will not have a major impact over the global design quality of the system, or even worsen it; and thus, the entire process of assessment and improvement needs to be iterated over and over again.

In this context, our novel idea is to add build a tool-supported impact analysis technique that would *simulate a correction strategy* on the model of the system before committing the restructuring solution. This way the impact of the change can be (automatically) assessed at a lower price. If the change has a major positive impact, it can be performed on the source code, otherwise it will remain as a possible unfulfilled change. Thus, it will become possible to estimate the impact of a change over the system.

In this context we envision the following usage scenario: the developer decides to correct a particular design problem, that was revealed by the *detection component* of the INCODE (or a similar tool). Consequently, she activates the *correction component* and executes a correction strategy of her choice (see Section 6.3). At the end of this correction process the *impact analysis component* will re-assess the overall design quality of the system, in terms of detectable design flaws For example: How many further design flaws were eliminated as a side-effect of applying the correction plan/strategy? Did new design flaws occur as a consequence of applying the correction plan/strategy? How many? The results of this differential impact analysis are presented to the developer who will decide if she wants to commit (keep) the new version of the system resulted by applying the correction plan. Depending on the decision of the engineer, the restructuring changes become permanent, or the system is rolled back to its original state.

7.1.2 Extending Correction Strategies

As shown in Section 6.3, automating design flaw correction and detection is a feasible task. Although so far we did define only a small set of such *correction strategies*, these are far from being trivial. But there is still a lot more work to be done to improve them and to encompass even the findings mentioned in Section 6.3. Using the approach presented there, new design flaws and correction strategies can be created.

The next logical step would be to define a language that allows anyone to define a design flaw correction and detection. This language should be as highly independent of the programming language in which design flaws are to be corrected. This language should provide means to ensure behavior preservation through the mechanism of preconditions and postconditions.

Another possible area of investigation would be to extend these restructurings into another programming language, most importantly procedural programming. As there is already a large existing set of design rules and guidelines for these languages, as well as a set of refactorings that can be applied, this transition is highly feasible.

Additionally, restructurings could be integrated with visualization techniques [LM06], that allow the user to see how the structure of the class will look after the refactoring was applied. Also a mechanism that asks the user input about the design intent and context and based on that decides which path to follow can be implemented.

7.2 Extending and Refining Detection Rules

7.2.1 Extending Detection Rules for Multi-Paradigm Systems

Among the very complex software systems that the industry is confronted with, over the last years a new type of application has emerged *i.e., enterprise applications*. An enterprise application is a software product that manipulates lots of persistent data and that interacts a lot with the user through a vast and complex user interface [Fow05]. As a reflection of our society, enterprise applications are characterized by a huge amount of *heterogeneity*: various implementation languages used in the same project, multiple programming paradigms, multiples coexisting technologies *etc.*. As we rely more and more on such systems, assuring their quality is a crucial concern. Yet, due to their intrinsic heterogeneity the current quality assurance techniques (*i.e.,* techniques for detecting design flaws in object-oriented systems) are necessary but not sufficient. A novel layer of dedicated quality assurance techniques is needed in order to address properly the multiple aspects of heterogeneity in design and implementation.

Fowler [Fow05] states that there are different sorts of software systems each of them having its own category of problems and complexities. In order to be well designed, an enterprise application must fulfill some specific design rules. Different authors [Fow05, Noc03, Mar02a] have proposed in recent years such rules. For example, we know that an enterprise software system consists of 3 layers: data- source layer, the business logic layer and the presentation layer [Fow05]. For example, a fundamental design rule for such systems states that the data layer and the logic layer must not be depended on the presentation layer. Moreover, the logic layer must be loosely coupled with the data layer in order to be able to replace the support that ensures the date persistency (*e.g.,* changing a relational database with an object-oriented one). The detection high coupling between the logic and data layers needs some advanced structural analysis (*e.g.,* detecting lack of polymorphism).

As we have seen in Chapter 3, the usual process of quality assessment for object-oriented systems starts from a set of quality assurance (QA) design rules and heuristics [Rie96, Mar02c], which are first transformed into a set of quantifiable rules, which can be then applied on the analyzed project, more precisely on the part of the project which is designed in an object-oriented manner. The result is a set of design fragments that are affected by design problems and that need to be restructured in order to improve the quality of the system.

The main assumption of the aforementioned approach is that software systems are *homogenous* (*i.e.,* that they can be analyzed using only object-oriented QA rules). But, such an approach ignores the relation to the persistency layer (which could be for example a RDBMS) and makes no distinction between the user-interface and the business domain part of an application. But addressing quality assurance in this manner is less and less feasible as software applications become more heterogeneous. Summarizing, we can state that these drawbacks are caused by the intrinsic heterogeneity of very complex systems (including enterprise applications). Enterprise applications encapsulate different technologies, different paradigms and must fulfill specific design rules and heuristics depending on the concrete type of the application. All these aspects must be considered when the quality of the design and implementation are evaluated. Unfortunately, almost none of the analyses techniques take into consideration this type of heterogeneity. In other words they unilaterally treat the analyzed system. Because of this reason the current quality estimation techniques strongly needs improvement.

In Figure 7.1 we depict our vision of a generic design quality assessment approach for an enterprise software system. The approach takes into consideration the heterogeneity of such systems and the consequent need to reflect this characteristic in the techniques used for quality assessment.

The first important distinction is that each of three layers of an enterprise application (*i.e.,* the presentation, the domain and the data-source layer) must be addressed by a distinct set of specialized QA rules. These design rules exist for both the design of user-interfaces and also for the design of databases. Based on these rules, problematic design fragments can be identified for the presentation and the data-source layer. Yet, none of the three layers exist in isolation; consequently there is a significant amount of the system's complexity involved in the relation between these layers. This brings us to the first two issues (marked in Figure 7.1 with the red bullets numbered 1 and 2):

- *Problem 1*: What are the proper QA design rules that specify the relation between

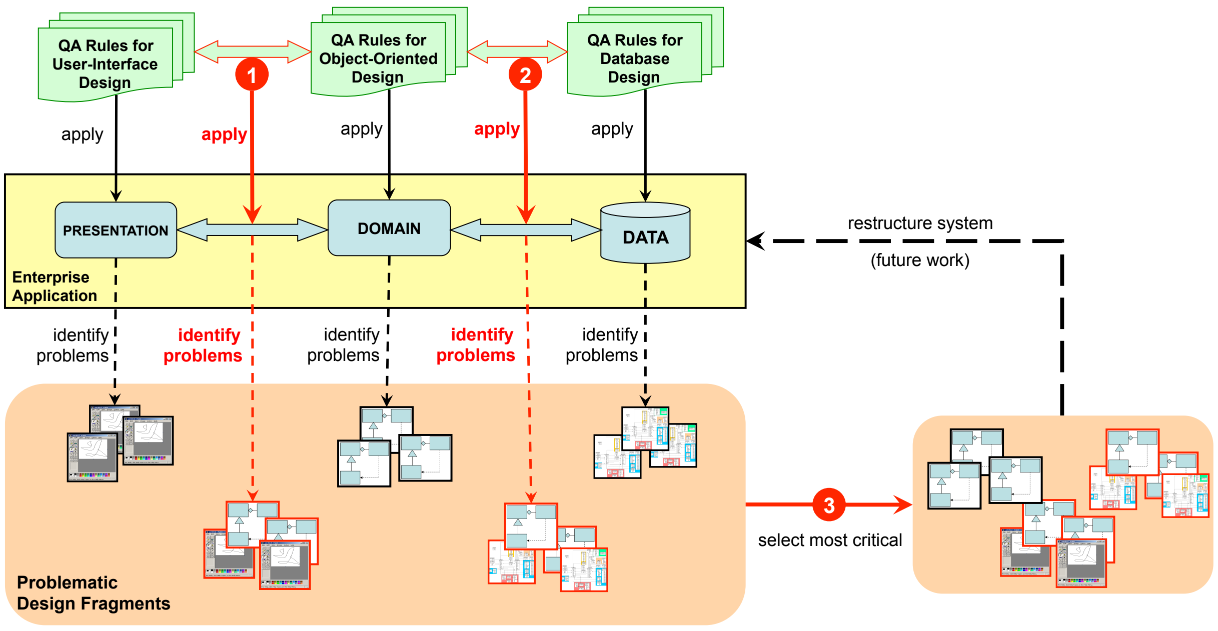


Figure 7.1: Quality assessment approach for an enterprise software system. The three red numbered bullets mark the main problems

the presentation and the domain layer? How can these rules be made quantifiable? What is the proper tools support needed to detect these design problems automatically?

- *Problem 2*: What are the proper QA design rules that specify the relation between the domain and the data-source layer, especially if the data-source layer is based on a relational database model? How can we overcome the paradigm shift? What is the proper tool support needed to detect these design problems automatically?

As mentioned before, quality assessment is not a goal in itself. The real goal is to improve the quality of the system by restructuring all the identified problems. But assuming that for each layer a set of design fragments (*i.e.*, classes, methods, relations among database tables) are detected as containing design problems, the number of such problematic fragments may increase significantly compared to the more simplistic approach of pure object-oriented code. Consequently, it is highly probable that it will not be economically feasible to address all the identified problems. This brings us to the third issue that we want to address (marked in Figure 7.1 with the red bullet numbered 3):

- *Problem 3*: How to define efficient (i.e., what must be restructured first?) and cost-effective (i.e., what is the optimal restructuring option?) plans for improving the quality of the design? Thus, a strong methodology and tool support is required in order to quickly asses the quality of a possible restructuring solution in terms of human effort, time, cost and quality amelioration.

Over the last years we have partially addressed these issues within the LOOSE Research Group, especially by addressing the first 2 problems for enterprise systems [Mar07a, Mar07b] as well as for the specificities of distributed systems [CM08, CM07] and, last but not least, for Lisp systems [DGM08]. However, the issue of specific detection rules for multi-paradigm systems remains open; and, additionally, the third problem (*i.e.*, restructuring) has not yet been addressed.

7.2.2 Defect-Based Refinement of Detection Techniques

There are two major dimensions related to quality assurance in complex software systems: (i) detection of *defects* (bugs) and (ii) detection and correction of *design flaws*. In the past, significant research effort has been spent in each of these two areas, but there is a total *lack of correlation* between these two dimensions: detection of coding flaws is only studied ignoring the issue of design flaws and vice-versa. This is because the communities addressing these two issues are working separately with few connection points. Furthermore, there is a *lack of correction focus*; for example, if the code changes due to refactoring/redesign/bug fixing, how do the original test suites have to be adapted? Or vice-versa: if a bug is removed, resulting in a non-negligible change to the code, how do we know that the change did not alter the design structure (*i.e.,* introduced a new instance of a design flaw)? And, last but not least there is the issue of *problems' prioritization i.e.,* having a large list of design or coding flaws, where do one start correcting them?

The integration of techniques related to design and coding flaw detection is an important novelty. Traditionally targeted by different research communities, we plan to take advantage of expertise in both areas in order explore their interaction and synergies, increasing the efficiency of their joint use. Consequently, in this respect we have a twofold future goal:

1. *Study the correlation between design flaws and defects (bugs) i.e.,* to investigate correlations between the presence of design and code flaws in terms of their number, type, and evolution in time. This study will involve an investigation of (i) *numerical correlations* (*i.e.,* number of design flaws vs. number of code flaws), (ii) *typological correlations* (which design flaws correlated with which code flaws), and (iii) historical correlations (*i.e.,* analyze the system versioning history to detect correlations between the presence of design and code flaws, such as simultaneous appearance and disappearance, or weaker *de-phased patterns* of correlation). For this study we plan use a significant number of large-scale complex software systems, as this will allow us to investigate the correlation on various programming languages, and thus strengthen the conclusions of the study.

2. *Investigate potential correlation means between defects and design flaws for focusing assessment activities.* More exactly we will investigate how the dominant presence of design flaws can focus the testing process; and vice-versa, how the presence of a significant number of code flaws can trigger a redesign that would improve code and thus reduce the the number of future code flaws. This focalization can occur in two directions: (i) given the presence of many design flaws in a design fragment (*e.g.,* package or class hierarchy), this should focus a more thorough (fine-grained) verification (*e.g.,* test generation) of that code portion; or, (ii) given the presence of many code flaws in a design fragment, raise the priority of building and executing a *correction strategy* for that part of the system, as such a restructuring may improve code understanding (*e.g.,* complexity reduction of a method by replacing large conditional blocks with a polymorphic method call) and thus reduce the future risk of code flaws for that design fragment. As part of our research methodology we plan to integrate at the tool level the testing and the design assessment tools developed in the project and define a connection module that would ensure the bidirectional focalization of the testing and design assessment and correction as described above.

7.3 Calibration of Metrics for Quality Assessment

In order to grow the confidence of using metrics the issue of thresholds has to be addressed with more rigor than it was addressed in the past [LM06]. Our main goal in this direction is to provide a broad empirical validation of assessment techniques, by creating a comprehensive metrics benchmarking methodology and by operationalizing it using adequate tools. This will create a set of tools for automatically collecting and analyzing a very large number (several thousands) of systems. The analysis results are used to calibrate the quality assessment techniques and to detect potential inconsistencies in tools that implement the quality models.

7.3.1 Methodology for tool calibration and validation

The starting point is to define a concrete and comprehensive set of calibration and validation goals, based on the quality assessment methods defined so far, and on the tools that support them. The methodology will cover the following aspects:

- Validate the consistency and completeness of the analysis models extracted from the various artifacts of projects. The tools involved in assessment rely on complex analysis techniques, which in turn rely on proper fact (model) extractors. As these must be usable for a large variety of software systems, it is essential to make sure that the tools, and especially their model extractors, are validated on a large number of extremely heterogeneous systems. In concrete terms, we will define a set of *checksum heuristic indicators* that would reveal severe abnormalities and/or inconsistencies in the extracted models.

- Define selection criteria for the projects used for calibration and validation, namely to define criteria (*e.g.,* programming language, maturity, use of a particular communication middleware or technology, application domain, etc.) for selecting the actual projects on which the calibration and validation will take place. Because calibration and validation need a large number of systems, our goal is to select, according to the defined criteria, several thousand projects by mining several of the most popular open-source repositories (e.g. *SourceForge*, *GitHub*). Furthermore, by actively seeking collaborations with the industry we will include in the investigated projects' set any commercial software system that we can get access to.

- Defining the calibration procedure. The goal of calibration is to makes sure that the quality assessment techniques are appropriate for assessing state-of-the-art systems, namely that it is neither too strict nor too lenient for existing systems. The procedure will specify how to spot inappropriateness patterns when applying quality models on the selected set of projects used for calibration. Furthermore, the procedure will also provide the input needed for performing the refinements of the weights and thresholds of the quality indicators used in these models.

7.3.2 Tools for calibration and validation

Performing such a large-scale calibration and validation is only possible if the methodology is automated to a large extent. Consequently, we will be focused on creating a set of tools that will automatically mine open-source repositories, select and grab projects according to the criteria defined in the methodology, feed them into the tools and collect the results. These tools will be used in three steps:

- First, tools will analyze the entire project base and will report any abnormalities in the *checksum heuristic indicators* defined as part of the methodology. These abnormalities that reveal potential inconsistencies in the analysis models have to be then addressed (either by improving the tools or by removing the results from the outlier system) before the actual validation can take place.
- Next, the tools will analyze half of all the projects selected for calibration and validation (in the range of thousands), and for each quality certification model it will provide the results that will indicate and guide the needed refinement, if such a refinement is necessary.
- Eventually, the same tools will perform the validation by analyzing the other half of the selected projects

After the quality assessment tools have been updated to incorporate the calibrated quality models, in this task we validate these models by analyzing the other half of the project set, using again the tools implemented for this purpose. During this quantitative analysis we will again observe if the adjustments performed during calibration led to appropriate results on this new set of analyzed projects. However, this time the statistical, quantitative analysis is doubled by a qualitative validation assessment. Concretely, we will pick randomly three sample systems: one among those that were ranked among the best by the quality model, one among the worst ranked and an average ranked system. These three systems will be manually inspected in order to validate the appropriateness of the classification provided by our quality models.

Part III

References

Bibliography

[ABF04] E. Arisholm, L.C. Briand, and A. Føyen. Dynamic Coupling Measurement for Object-Oriented Software. *Transactions on Software Engineering*, 30(8), 2004.

[ADB10] J. Al Dallal and L.C. Briand. An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 52(12), 2010.

[ARK05] J. Alghamdi, R. Rufai, and S. Khan. OOMeter: A Software Quality Assurance Tool. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2005.

[BBK$^+$78] B. Boehm, J. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt. *Characteristics of Software Quality*. North Holland, 1978.

[BD02] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1), 2002.

[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.

[BDW99a] L.C. Briand, J.W. Daly, and J.K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1), 1999.

[BDW99b] L.C. Briand, J.W. Daly, and J.K. Wüst. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Engineering (ICSE 1999)*. IEEE Computer Society Press, 1999.

[Bel02] S. Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. *Master's Thesis, Institut fur Softwaretechnologie, Universitat Stuttgart, Stuttgart, Germany*, 2002.

[BK95] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Symposium on Software Reusability(ACM)*. ACM Press, 1995.

[BMM98] W.H. Brown, R.C. Malveau, and T.J. Mowbray. *AntiPatterns: Refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[BR88] V. Basili and H.D. Rombach. The TAME project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6), 1988.

[BR94] V. Basili and H. Rombach. Goal Question Metric paradigm. *Encyclopedia of Software Engineering*, 1, 1994.

[CCI90] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1), 1990.

[Ciu99] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS USA*. IEEE Computer Society Press, 1999.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994.

[CM07] D.C. Cosma and R. Marinescu. Distributable features view: Visualizing the structural characteristics of distributed software systems. In *Visualizing Software for Understanding and Analysis(VISSOFT)*. IEEE Computer Society Press, 2007.

[CM08] D.C. Cosma and R. Marinescu. Understanding the impact of distribution in object-oriented distributed systems using structural program dependencies. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 103–112. IEEE Computer Society, 2008.

[Con09a] FLOSSMETRICS Consortium. The flossmetrics fp6 ec project. *http://www.flossmetrics.org/*, 2009.

[Con09b] QualOSS Consortium. The qualoss fp6 ec project. *http://www.qualoss.org/*, 2009.

[Con09c] SQO-OSS Consortium. The sqo-oss fp6 ec project. *http://www.sqo-oss.org/*, 2009.

[Cop05] T. Copeland. Pmd applied, 2005.

[Cun92] W. Cunningham. The WyCash portfolio management system. In *ACM SIGPLAN OOPS Messenger*, volume 4, 1992.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice Hall, London, 2nd edition, 1991.

[CY91b] Peter Coad and Edward Yourdon. *Object Oriented Design*. Prentice-Hall, 1991.

[DDN03] S. Demeyer, S. Ducasse, and O.M. Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufmann, 2003.

[DGM08] Adrian Dozsa, Tudor Gîrba, and Radu Marinescu. How Lisp systems look different. In *European Conference on Software Maintenance and Re-Engineering (CSMR)*, pages 223–232. IEEE Computer Society Press, 2008.

[DGN05] Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, 2005.

[DHH+11] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner. The quamoco tool chain for quality modeling and assessment. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE Computer Society, 2011.

[DM86] T. De Marco. *Controlling Software Projects: Management, Measurement, and Estimates*. Springer-Verlag, 1986.

[DPS05] F. Deissenboeck, M. Pizka, and T. Seifert. Tool support for continuous quality assessment. In *13th IEEE International Workshop on Software Technology and Engineering Practice*. IEEE, 2005.

[Dro95] R.G. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2), 1995.

[DSP+07] F. Deissenboeck, S.Wagner, M. Pizka, S. Teuchert, and J.F. Girard. An activity-based quality model for maintainability. In *IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 2007.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Fea05] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.

[Fou11a] Eclipse Foundation. Java Development Tools. *http://eclipse.org/jdt*, 2001-2011.

[Fou11b] Eclipse Foundation. Eclipse Modeling Framework. *http://www.eclipse.org/modeling/emf*, 2003-2011.

[Fow05] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2005.

[Fow09] M. Fowler. Technical debt. *Fowler's Bliki, http://martinfowler.com/bliki/TechnicalDebt.html*, 2009.

[FP96] N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, 1996.

[GDK+07] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and D. Raţiu. Using concept analysis to detect co-change patterns. In *International Workshop on Principles of Software Evolution (IWPSE)*. ACM Press, 2007.

[GHJ98] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 1998.

[Gmb10] Odysseus Software GmbH. Stan - structure analysis for java. *http://stan4j.com/*, 2010.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[HH04] A. Hassan and R. Holt. Predicting change propagation in software systems. In *IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2004.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[IBM10] IBM. Rational quality manager. *http://www-01.ibm.com/software/rational/*, 2010.

[Inc10] Klocwork Inc. Klocwork insight. *http://www.klocwork.com/products/insight*, 2010.

[Ins10] Instantiations. Codepro analytix. *http://www2.instantiations.com/codepro/analytix*, 2010.

[ISO91] International Standard Organization. ISO 9126 - quality characteristics and guidelines for their use. *Brussels*, 1991.

[JF88] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[KDPG09] F. Khomh, M. Di Penta, and Y.G. Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Working Conference on Reverse Engineering*. IEEE, 2009.

[KEGN01] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance(ICSM)*. IEEE Computer Society Press, 2001.

[Ker05] J. Kerievsky. *Refactoring to Patterns*. Pearson Education, 2005.

[KLPN97] B. Kitchenham, S. Linkman, A. Pasquini, and V. Nanni. The squid approach to defining a quality model. *Software Quality Journal*, 6(3), 1997.

[KP96] B. Kitchenham and S.L. Pfleeger. Software quality: the elusive target [special issues section]. *IEEE Transactions on Software Engineering*, 13(1), 1996.

[KPF95] B. Kitchenham, S.L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12), 1995.

[KTWW11] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams. An enterprise perspective on technical debt. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011.

[Lak96] J. Lakos. *Large Scale C++ Software Design*. Addison Wesley, 1996.

[LC09] JL Letouzey and T. Coq. The sqale models for assessing the quality of software source code. *DNV Paris, white paper (September 2009)*, 2009.

[LD03] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering(TSE)*, 29(9), 2003.

[Leh96] M. Lehman. Laws of software evolution revisited. *Software process technology*, pages 108–124, 1996.

[LEHP06] J. Li, N. Edgar, K. Haaland, and K. Peter. *Eclipse User Interface Guidelines*. Addison-Wesley Professional, 2006.

[LH93] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. *International Software Metrics Symposium.*, 1993.

[Lis87] B. Liskov. Data Abstraction and Hierarchy. In *International Conference on Object Oriented Programming, Systems, Languages and Applications(OOPSLA)*. ACM Press, 1987.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[LM06] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[LPR$^+$97] M. Lehman, D. Perry, J. Ramil, W. Turski, and P. Wernick. Metrics and laws of software evolution–the nineties view. In *International Software Metrics Symposium(METRICS)*. IEEE Computer Society Press, 1997.

[LR89] K.J. Lieberherr and A.J. Riel. Contributions to teaching object oriented design and programming. In *International Conference on Object Oriented Programming, Systems, Languages and Applications(OOPSLA)*. ACM Press, 1989.

[M.88] Bertrand M. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[Mar99] R. Marinescu. A multi-layered system of metrics for the measurement of reuse by inheritance. In *Technology of Object-Oriented Languages and Systems(TOOLS)*. IEEE Computer Society Press, 1999.

[Mar02a] F. Marinescu. *Ejb Design Patterns: Advanced Patterns, Processes, and Idioms with Poster*. John Wiley & Sons, Inc., 2002.

[Mar02b] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timişoara, 2002.

[Mar02c] R.C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[Mar04] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *International Conference on Software Maintenance(ICSM)*. IEEE Computer Society Press, 2004.

[Mar05] R. Marinescu. Measurement and quality in object-oriented design. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, 2005.

[Mar07a] C. Marinescu. Discovering the objectual meaning of foreign key constraints in enterprise applications. In *Working Conference on Reverse Engineering(WCRE)*. IEEE Computer Society Press, 2007.

[Mar07b] C. Marinescu. Identification of relational discrepancies between database schemas and source-code in enterprise applications. In *Symbolic and Numeric Algorithms for Scientific Computing(SYNASC)*. IEEE Computer Society Press, 2007.

[Mar12] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5 (to appear)), 2012.

[McC76] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[McC07] S. McConnell. Technical debt. *10x Software Development, http://bit.ly/McConnell07TechnicalDebt*, 2007.

[MD03] T. Mens and A. Van Deursen. Refactoring: Emerging trends and open problems, 2003.

[MG10] R. Marinescu and G. Ganea. inCode.Rules: An agile approach for defining and checking architectural constraints. In *International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE Computer Society Press, 2010.

[MGDLM10] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 2010.

[MGV10] R. Marinescu, G. Ganea, and I. Verebi. inCode: Continuous quality assessment and improvement. In *Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society Press, 2010.

[MM05] P. Mihancea and R. Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *Conference on Software Maintenance (CSMR)*. IEEE Computer Society Press, 2005.

[MM11] R. Marinescu and C. Marinescu. Are the clients of flawed classes (also) defect prone? In *Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society Press, 2011.

[MMG05] C. Marinescu, R. Marinescu, and T. Gîrba. Towards a simplified implementation of object-oriented design metrics. In *Symposium on Software Metrics (METRICS)*. IEEE Computer Society Press, 2005.

[MMM$^+$05] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 77–80. IEEE Computer Society, 2005.

[Moo96] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Object-Oriented Programming Systems, Languages, and Applications(OOPSLA)*. ACM Press, 1996.

[MR04] R. Marinescu and D. Raţiu. Quantifying the quality of object-oriented design: the Factor-Strategy model. In *Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2004.

[MRW76] J. McCall, P. Richards, and G. Walters. *Factors in Software Quality*. NTIS Springfield, 1976.

[Mun05] M.J. Munro. Product metrics for automatic identification of bad smell design problems in java source-code. In *Symposium on Software Metrics (METRICS)*. IEEE Computer Society Press, 2005.

[Noc03] C. Nock. *Data access patterns: database interactions in object-oriented applications*. Prentice Hall Professional Technical Reference, 2003.

[OCN99] M. Ó Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *International Conference on Software Maintenance(ICSM)*. IEEE Computer Society Press, 1999.

[oM10] University of Maryland. Findbugs. *http://findbugs.sourceforge.net/*, 2010.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Par94] D.L. Parnas. Software aging. In *International Conference on Software Engineering(ICSE)*. IEEE Computer Society Press, 1994.

[plu] JDeodorant Eclipse plugin. http://www.jdeodorant.com/.

[Pre10] R.S. Pressman. *Software Engineering: A Practitioner's Approach, 7th Edition*. McGraw-Hill, 2010.

[RBJ97] D. Roberts, J. Brant, and R. Johnson. Theory and practice of object systems. *Theor. Pract. Object Syst.*, 3(4), 1997.

[RDGM04] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society Press, 2004.

[Rie96] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[Rob99] D.B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[SBL01] H.A. Sahraoui, M. Boukadoum, and H. Lounis. Building quality estimation models with fuzzy threshold values. *L'objet*, 17(4), 2001.

[Sof10] Headway Software. Structure 101. *http://www.headwaysoftware.com*, 2010.

[Son10] SonarSource. Sonar. *http://www.sonarsource.org*, 2010.

[SSL01] Frank S., F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Conference on Software Maintenance and Reengineering(CSMR)*. IEEE Computer Society Press, 2001.

[Ste11] C. Sterling. *Managing Software Debt*. Addison-Wesley, 2011.

[TC] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3).

[TM05] A. Trifu and R. Marinescu. Diagnosing design problems in object oriented systems. In *Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2005.

[Tri08] A. Trifu. *Towards Automated Restructuring of Object Oriented Systems*. Kit Scientific Publishing, 2008.

[Ver09] I. Verebi. Automation of complex design restructurings. Master's thesis, Politehnica University of Timisoara, 2009.

[Vit03] M. Vittek. Refactoring browser with preprocessor. In *Conference on Software Maintenance and Reengineering(CSMR)*. IEEE Computer Society Press, 2003.

[Web05] SDMetrics Website. Sdmetrics, 2005.

[Web10] Checkstyle Website. The checkstyle plugin for eclipse. *http://eclipse-cs.sourceforge.net/*, 2010.

[WH] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, (12).

[WM05] R. Wettel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE Computer Society Press, 2005.

[ZWDZ04] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 2004.