Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Modularity Improvements with Aspect-Oriented Programming

Fernando Sérgio Bryton Dias Marques

Dissertação apresentada para a obtenção do Grau de Mestre em Engenharia Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(2008)

This dissertation was prepared under the supervision of
Professor Fernando Brito e Abreu,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

Measure what is measurable, and make measurable what is not so.
Galileo Galilei (1564 - 1642)

[This page was intentionally left blank]

Summary

Aspect-Oriented Programming is a topic of growing interest, since its presentation in 1997, and significant modularity improvement has, since then, been claimed as one of its main benefits, both by researchers and practitioners.

The importance of modularity for maintenance and reuse which, consequently, are crucial for software development time and costs, emphasizes this programming paradigm, and strongly suggests its adoption by academia and industry. However, most of the claims on those benefits are, so far, qualitative, and only a small subset of them has quantitative grounds; therefore, any decision regarding where to adopt or not this paradigm is not yet sufficiently grounded.

This dissertation presents a modularity quantitative evaluation of the Gang-of-Four Design Patterns functionally equivalent implementations, in Java and AspectJ, by Hannemann and Kiczales and, within this context, draws conclusions about the modularity improvements obtained with Aspect-Oriented Programming, which are expected to contribute to decisions on where this paradigm should or not be adopted. This evaluation is made with the Paradigm Independent Metrics (PIMETRICS), defined according to the Meta-Model Driven Measurement (M2DM) technique, and grounded at the Paradigm Independent Meta-model (PIMETA) which implements the Paradigm Independent Modularity Factors (PIFACTORS).

[This page was intentionally left blank]

Sumário

A Programação Orientada a Aspectos é um tópico cujo interesse tem vindo a aumentar, desde a sua apresentação em 1997. Os benefícios que este paradigma introduz na modularidade têm sido reconhecidos, desde então, como uma das suas principais vantagens.

A importância da modularidade na manutenção e reutilização e, consequentemente, no tempo de desenvolvimento do software e respectivo custo, relevam assim a importância deste paradigma e recomendam a sua adopção. Contudo, a maioria das avaliações efectuadas sobre os benefícios deste paradigma têm sido, até à data, na sua maioria de natureza qualitativa, resumido-se as de natureza quantitativa a um grupo muito restrito. Por conseguinte, a decisão sobre a adopção ou não deste paradigma ainda não está suficientemente consolidada.

Esta dissertação apresenta uma avaliação quantitativa das implementações funcionalmente equivalentes, em Java e AspectJ, dos padrões de desenho do *Gang-of-Four* , desenvolvidas por Hannemann e Kiczales, e neste contexto, retira conclusões sobre os benefícios da modularidade obtidos com a Programação Orientada a Aspectos, que se espera venham a contribuir para a decisão sobre a adopção ou não deste paradigma. Esta avaliação é feita com métricas independentes do paradigma(PIMETRICS), definidas de acordo com a técnica *Meta-model Driven Measurement*(M2DM), e baseadas num meta-modelo independente do paradigma para avaliação da modularidade(PIMETA) e em factores de modularidade também independentes do paradigma(PIFACTORS).

[This page was intentionally left blank]

Contents

1 Introduction **1**

1.1 Introduction . . . 2
1.2 Modularity . . . 4
1.3 Coupling . . . 5
1.4 Cohesion . . . 5
1.5 Aspect-Oriented Programming . . . 6
1.6 Meta-Model Driven Measurement . . . 7
1.7 Contributions of this Thesis . . . 8
1.8 Outline of the Dissertation . . . 9

2 PIMETA: Paradigm Independent Meta-model **11**

2.1 Introduction . . . 12
2.2 Paradigm Independent Meta-model . . . 12
2.2.1 PIMETA description . . . 12
2.2.2 PIMETA well-formedness rules . . . 14
2.3 PIMETA instantiation with OOP and AOP . . . 16
2.3.1 PIMETA instantiation with OOP/Java . . . 16
2.3.2 PIMETA instantiation with AOP/AspectJ . . . 17
2.4 The Observer Design Pattern in Java and AspectJ . . . 17
2.5 PIMETA instantiation with the Observer versions . . . 18
2.5.1 PIMETA instantiation with the Java version of the Observer . . . 18
2.5.2 PIMETA instantiation with the AspectJ version of the Observer . 22

3 PIFACTORS: Paradigm Independent Modularity Factors **23**

3.1 Introduction . . . 24
3.2 Taxonomy for dependency topologies . . . 25
3.2.1 Role . . . 25
3.2.2 Coverage . . . 26
3.2.3 Commitment . . . 27
3.2.4 Clustering . . . 29
3.2.5 Density . . . 31
3.2.6 Type . . . 32

4 PIMETRICS: Paradigm Independent Metrics **35**
4.1 Introduction 36
4.2 Definitions 37
4.3 Formalizations 43
4.4 Metrics collection 50
4.5 Exemplification 51
4.6 Validation 56

5 GoF Design Patterns Modularity **59**
5.1 Introduction 60
5.2 The sample 60
5.3 Descriptive statistics 63
5.4 EDA: Exploratory Data Analysis 65
5.4.1 Outlier analysis 65
5.4.2 Testing distribution adherence 67
5.5 Hypotheses identification and testing 69
5.5.1 Metrics discriminative power 69
5.5.2 Overall coupling changes 71
5.5.3 Core modular features coupling changes 73
5.5.4 Overall cohesion changes 75
5.5.5 Core modular features cohesion changes 76
5.5.6 Overall modularity changes 78

6 Related Work **81**
6.1 Introduction 82
6.2 PIFACTORS 82
6.3 PIMETRICS 83
6.4 AOP Modularity Improvements Evaluation 84

7 Conclusions and Future work **87**
7.1 Conclusions 88
7.2 Threats to validity 89
7.3 Future work 90
7.3.1 Quantitative Modularity Evaluation 90
7.3.2 Refactoring 91

A Java Feature and Dependency Types **93**

B AspectJ Feature and Dependency Types **97**

C PIMETRICS primitives **103**

List of Figures

1.1 Code scattering and tangling . . . 7
1.2 Meta-model Driven Measurement Technique . . . 8

2.1 PIMETA Meta-class diagram . . . 13
2.2 PIMETA well-formedness rules in OCL . . . 14
2.3 PIMETA well-formedness rules auxiliary operations in OCL . . . 15
2.4 PIMETA instantiation extract for Java . . . 16
2.5 PIMETA instantiation extract for AspectJ . . . 17
2.6 Observer - Java version class diagram . . . 18
2.7 Observer - AspectJ version class diagram . . . 19
2.8 Screen class source code from the Observer pattern in Java . . . 20
2.9 PIMETA instantiation with the *Screen* class of the Observer in Java . . . 21
2.10 Screen class source code from the Observer pattern in AspectJ . . . 22
2.11 PIMETA instantiation with the Screen class of the Observer in AspectJ . 22

3.1 Aspectized Screen class dependencies graph . . . 24
3.2 Source Coverage example . . . 26
3.3 Target Coverage example . . . 27
3.4 Source Commitment example . . . 28
3.5 Target Commitment example . . . 28
3.6 Source Clustering example . . . 30
3.7 Target Clustering example . . . 30
3.8 Source Density example . . . 31
3.9 Target Density example . . . 32

4.1 Summary of the metrics for features . . . 41
4.2 Metrics collection process . . . 50
4.3 Results of the absolute metrics from goal G1 . . . 52
4.4 Results of the absolute metrics from goal G1 (cont.) . . . 52
4.5 Results of the ratio metrics from goal G1 . . . 52
4.6 Results of the absolute metrics from goal G2 . . . 53
4.7 Results of the ratio metrics from goal G2 . . . 53
4.8 Results of the absolute metrics from goal G3 . . . 54

4.9 Results of the ratio metrics from goal G3 55
4.10 Results of the absolute metrics from goal G3 (cont.) 55
4.11 Results of the absolute metrics from goal G4 55
4.12 Results of the ratio metrics from goal G4 56

5.1 Coarsest grained modular features distribution by the GoF implementations 61
5.2 Number of features per design pattern and language for the coarsest feature types 61
5.3 Number of features per *FeatureType* and *Language* 62
5.4 Number of features per *Design Pattern* and *Language* 62
5.5 Number of dependencies per *DependencyType* and *Language* 63
5.6 Number of dependencies per *Design Pattern* and *Language* 64
5.7 Variables used, their scale types and description 64
5.8 Descriptive statistics on the PIMETRICS (indexes) 65
5.9 Outlier analysis for the PIMETRICS (indexes) 66
5.10 QQ Plots for SCouCovI, TCouCovI, SCouCluI and TCouCluI 67
5.11 QQ Plots for SCouComI, TCoucomI, SCohComI and TCohComI 68
5.12 Testing Normal distribution adherence with the Kolmogorov-Smirnov test for the PIMETRICS 68
5.13 Testing Normal distribution adherence with the Kolmogorov-Smirnov test for the remaining metrics 68
5.14 Testing the influence of the Java design pattern on coupling and cohesion metrics with the Kruskal-Wallis one-way analysis of variance test . 70
5.15 Testing the influence of the AspectJ design pattern on coupling and cohesion metrics with the Kruskal-Wallis one-way analysis of variance test 70
5.16 Testing the influence of the Java design pattern on modularity metrics with the Kruskal-Wallis one-way analysis of variance test 70
5.17 Testing the influence of the AspectJ design pattern on modularity metrics with the Kruskal-Wallis one-way analysis of variance test 70
5.18 PIMETRICS average per GoF implementation 72
5.19 *t-test* applied to the PIMETRICS indexes 72
5.20 Paired samples statistics for core modular features coupling 73
5.21 Paired samples correlations for core modular features coupling 74
5.22 Paired samples test for core modular features coupling 74
5.23 Paired samples statistics for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ 75
5.24 Levene's and t-test for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ 76

5.25 Paired descriptive statistics for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ 77
5.26 Paired correlations for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ 77
5.27 *t-test* for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ 77
5.28 Paired samples statistics for *PCI* and *MMF* for the GoF patterns in Java and AspectJ 78
5.29 Levene's and t-test for *PCI* and *MMF* for the GoF patterns in Java and AspectJ 78

[This page was intentionally left blank]

List of Tables

1.1 System architecture features 5

4.1 Goals summary 37
4.2 GQM model for coupling evaluation for *features* 38
4.3 GQM model for coupling evaluation for *modular features* 39
4.4 GQM model for cohesion evaluation for *modular features* 40
4.5 GQM model for system modularity evaluation 42
4.6 Metrics formalization for coupling evaluation for features 43
4.7 Metrics formalization for coupling evaluation for features (cont.) 44
4.8 Metrics formalization for coupling evaluation for features (cont.) 45
4.9 Metrics formalization for coupling evaluation for modular features . . . 46
4.10 Metrics formalization for cohesion evaluation for modular features . . . 47
4.11 Metrics formalization for cohesion evaluation for modular features (cont.) 48
4.12 Metrics formalization for *Modularity* evaluation for systems 49

A.1 Java Feature Types 94
A.2 Java Dependency Types 95

B.1 AspectJ Feature Types 98
B.2 AspectJ Features (cont.) 99
B.3 AspectJ Dependency Types 100
B.4 AspectJ Dependency Types (cont.) 101

C.1 Feature operations 104
C.2 Feature operations (cont.) 105
C.3 Feature operations (cont.) 106
C.4 Feature operations (cont.) 107
C.5 Feature operations (cont.) 108
C.6 ModularFeature operations (cont.) 109
C.7 ModularFeature operations (cont.) 110
C.8 ModularFeature operations 111

[This page was intentionally left blank]

Chapter 1

Introduction

Contents

1.1 Introduction 2
1.2 Modularity 4
1.3 Coupling 5
1.4 Cohesion 5
1.5 Aspect-Oriented Programming 6
1.6 Meta-Model Driven Measurement 7
1.7 Contributions of this Thesis 8
1.8 Outline of the Dissertation 9

This chapter introduces the main concepts that are present throughout this dissertation and the motivation to the quantitative modularity assessment of systems developed with aspect-oriented programming. It also enumerates the main contributions of this dissertation and presents its outline, with a brief summary of each of the remaining chapters.

1.1 Introduction

New software development paradigms usually emerge and grow on a bottom-up fashion, following a sequence of overlapping concern phases. This evolution path can be observed through surveys on researchers' and leading practitioners' work evidences (conference communications, journal papers, books, etc).

First, computer scientists come up with new programming metaphors usually bundled within programming languages; then commercial software-houses and open-source communities conceive development environments supporting those new languages; software engineering researchers then focus their attention on analysis and design concerns; methodologists start to be concerned on supporting new kinds of development and assessment processes; information systems people develop new arguments for business value of technology migration; and finally everybody gets worried on evolution and legacy systems.

This was the case for the procedural and object-oriented paradigms and the whole story will most probably repeat for the more recent one - the aspect-oriented paradigm.

First of all notice that each paradigm *wave* is usually long. They often span over more than a decade. This long period of time corresponds to the *ad nausea* exhaustion of each of the above mentioned research concerns in the context of the paradigm in hand.

If we take a specific concern from the above list, we can observe, throughout time, a strong evidence of periodic research output bursts, each one corresponding to the emergence of a new paradigm. For instance, regarding the concern of the quantitative assessment of software products we had a wave of interest centered on the procedural paradigm until the mid-eighties, then on the object-oriented paradigm until the late nineties and currently a wave is rising around the aspect-oriented paradigm. While the creation of new research niches may be interesting for PhD students searching for a potential new research topic, there are several perverse side-effects associated with this paradigm-shift phenomenon both on the research and practitioner's sides.

Let us look at the research side first. When performing a state of the art survey for finding who is doing what and where your bright new ideas can be published, you will often disregard or even avoid the exhausted old gold lode. In other words, either you ignore the work developed in the same topic, but in the context of a previous paradigm, more than a decade old as we saw, or you look at it with haughtiness to quickly conclude that the whole work performed in the past simply does not apply for the new paradigm. This is a very convenient conclusion for new researchers eager for publishing and creating their own space.

Besides, comparing things across the paradigm border is a hard nut to crack, since new paradigm heralds tend to stress innovative paradigm characteristics and not the commonalities with the *old* paradigm. You have to be an expert on both (old and new)

paradigms to understand the whole picture. The perverse effects of just looking at the new paradigm are that good reusable ideas of the past are forgotten and new researchers often reinvent the wheel. Also reprehensible is to adapt good old ideas, presenting them as innovative (i.e. not crediting their origin).

Now let us look to the world of business. In smaller businesses where you can afford the risk of being adventurous, you are usually in the hands of hype marketers when it comes to paradigm shift. You are simply left in the dark because the quantitative studies bringing evidence on the technological and consequent managerial benefits of migration are very scarce [GVR02] [RGV04].

This unfortunate tradition for a lack of insufficient validation of claims has been reported for more than a decade [TLPH95] [ZW97]. It has been pointed out that one of the strongest reasons for this scarcity is the lack of cooperation among academic researchers and practitioners facing the migration dilemma. As far as we know very few joint works of this sort were published [LTBM99].

On the other hand, although researchers gather to discuss migration concerns, their focus, although technical, is qualitative in nature (e.g. [MRC06] reports on aspect oriented software development and software engineering researchers with a focus on reverse engineering, program comprehension, software evolution and software maintenance confronting their thoughts).

However, for software managers, concerned with training, development and maintenance costs, Return On Investment (ROI) and other economic figures, a pure technical and qualitative discussion is fruitless. A responsible Chief Information Officer (CIO) or project manager (PM) will not take unplanned risks. He will let others prove that the technology supporting the new paradigm is mature enough before streamlining it in his development shop. In the end, strong players, such as big software vendors with their huge marketing sales force, will be the ones to impose the evolution track, after very careful technology maturation. By that time the careful CIO may have lost a competitive advantage.

Summing up, this status quo of long transition periods for paradigm shift (widespread adoption) will not change unless researchers do their share of the work: provide quantitative evidence that migration is cost-effective. This proof can be performed partly if researchers show, beyond doubt, that a system developed with a new paradigm is easier to understand and maintain than a functionally equivalent one, built with the previous paradigm (see for instance [DMM05]).

To be adequate and fair, this kind of quantitative comparisons require that you do not compare apples with oranges. If, for instance, you are interested in comparing the object-oriented programming (OOP) and the aspect-oriented programming (AOP) paradigms, then you should use the same ruler for measuring (quantifying) the characteristics of functionally-equivalent OOP and AOP artifacts. However, for reasons presented above, academic researchers are too much often tempted to do the opposite,

that is, suggesting new rulers [GBB06] [ZH03] [Zha04] [ZX04] [CT04]. The justification for proposing specific metrics is that each paradigm is supported in formalisms that offer different constructs, introducing new sorts of containment and dependency relations.

To compare apples with oranges, we must be able to abstract from their differences and concentrate on their commonalities. After all, apples and oranges have previously been shown to be remarkably similar [Bar00]. In our case this means that we should concentrate on the empiric concepts that explain the software properties under measurement (see the representational theory of measurement in [FP97]). For instance, if we are interested in evaluating modularity, then we should consider modules, items belonging to them and their internal and external dependencies. Each paradigm (and in fact each reification language), will have its own instantiation of those concepts.

This dissertation proposes and demonstrates a sound alternative for cross-paradigm modularity quantitative evaluation, based on the Meta-Model Driven Measurement (M2DM) [Abr01], which encompasses the usage of metrics defined upon meta-models, and presents the modularity improvements obtained with aspect-oriented programming in a functional equivalent implementation.

Throughout the remainder sections of this chapter, the main concepts encompassed in this dissertation will be summarily introduced, followed by the presentations of the main contributes and of the dissertation outline.

1.2 Modularity

One of the main reasons why systems are modularized is because one of the most fundamental principles of solving large and complex problems is that *breaking up the problem into smaller parts enhances understandability and tractability* [Pol57].

According to Myers, *modularity* [1] *is the single attribute of software that allows a program to be intellectually manageable* [Mye78]. With this and *reusability*[2] in mind, software is divided into modules, *a set of one or more continuous program statements having a name by which other parts of the system can invoke it, and preferably having its own distinct set of variable names that interact to satisfy problem requirements* [SMC74].

The criteria followed to accomplish this division is the key issue to achieve *good* modularity [3], which is claimed to provide lower development effort and costs [Pre00].

With such consequences, it is not surprising that modularity has been addressed by several programming paradigms, within the last 40 years, from which Aspect-Oriented Programming is one of the newest.

Notice that this definition of a module, such an important concept for software

[1] The division of software into modules

[2] The capability provided by a module to be used outside the scope to which it was developed to reduce the development effort

[3] *Undermodularity* or *overmodularity* should be avoided.

development, was given 40 years ago but is still up-to-date, is quite broad, and not bound to any programming paradigm or language in particular.

According to Pressman, modules should be developed in order to be *functionally independent*. They should address a specific set of requirements and, this way, their interfaces will be easier to understand and maintain and reusability is possible. This independence is usually assessed using two criteria: coupling and cohesion [Pre00].

Many taxonomies could be used to describe the architecture of a software system, like the one from Meyer's [Mey97], for instance, but since that one of the purposes of this research is to achieve its goal without being bound to any particular programming paradigm or language, it was chosen to describe such an architecture in neutral terms, more abstract, to avoid semantic bias, that will be used throughout this dissertation.

A software system architecture is then composed by architectural *features* which implement the requirements. These *features* can be *atomic features* or *modular features* regarding their capability of containing other *features* . If a *feature* is able to contain other *features* , then it is a *modular feature*. If the *feature* does not have this capability, then it is an *atomic feature*. Table 1.1 shows concrete examples for these concepts in the context of UML, Java and AspectJ.

	Modular Feature	Atomic Feature
UML	Component, Class, Interface	Method, Property
Java	Package, Class, Interface	Function, Variable
AspectJ	Package, Aspect	Advice, Pointcut

Table 1.1: Examples from different languages for the system architecture features

1.3 Coupling

Coupling has been defined as the degree of interaction within two modules [Mye78] or, more recently, as a measure of the relative interdependence among modules [Pre00].

Either way, modules, by definition, must interact with each other to satisfy requirements, but should do it as little as possible. In other words, they should not depend too much on the remaining modules.

Low coupling has been pointed out as a benefic factor on several aspects such as understandability, reusability and reduction on the impact of modifications and therefore error propagation [Pre00].

1.4 Cohesion

Cohesion has been defined as the degree of interaction within a module [Mye78] or, more recently, as a measure of the relative functional strength of a module [Pre00].

To summarize, a module contains code that interacts to achieve a common goal. The higher this interaction, the higher the cohesion, as long as the functional objective to achieve remains the same.

When the code from a module is combined with a loose functional relation, the module is more difficult to maintain and reuse.

1.5 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP), proposed by Kiczales et al. [KLM+97], is a programming paradigm of growing interest, given its claimed modularity benefits both by researchers and practitioners as in [HK02], [SGK+04] or [KSG+06].

AOP tries to eliminate two important causes for the loss of modularity from Object-Oriented Programming. These are *code tangling* and *code scattering*, which are originated by the implementation of the *crosscutting concerns*.

Code tangling is the name given to the introduction of code into a module that does not implement the same requirement for which the module was designed, in other words, the module is implementing a *crosscutting concern*. This action potentially decreases cohesion.

Code scattering is the name given to the introduction of code that implements one requirement into several modules simultaneously. This action potentially increases coupling.

These concepts are depicted in figure 1.1 [PHP08], where the *database access* and the *logging* concerns originate code scattering and code tangling when implemented within the core system modules, *Bookshell*, *Book* and *Modules*.

The way AOP tries to solve these problems is by encapsulating the *crosscutting concerns* into *aspects*, removing them from the modules they are *deforming* and, with specific constructs, applying the code again to the places where it should be executed, in compile or run-time. From a design point-of-view, the software allegedly gains modularity due to the decrease of coupling and the increase of cohesion. However, this analysis is merely qualitative, restricted to the core classes, and does not mean that AOP does not encompass other dependencies that can harm modularity.

By quantitatively evaluating the modularity obtained with aspect-oriented programming, it will be possible to clearly understand the impact of this new paradigm, from which many qualitative appreciations have been made so far, exemplified by those in [KLM+97], in [HK02] or in [Lad03]. It will also be possible to objectively decide if, when, and how, it should be applied, as well as to identify where it can be improved.

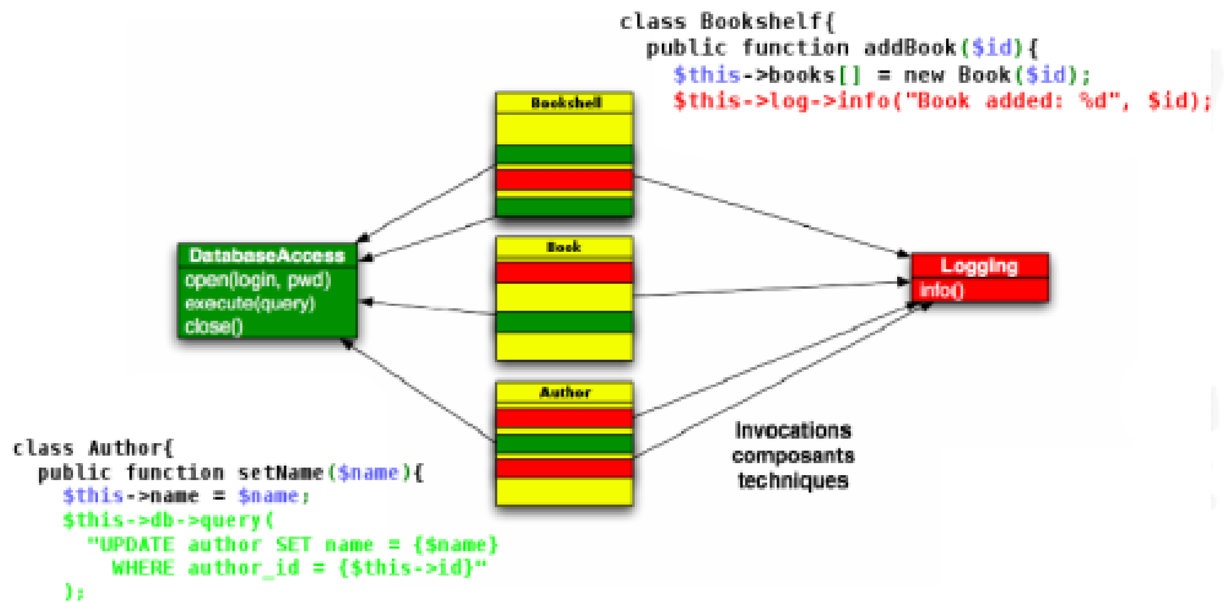

Figure 1.1: Code scattering and tangling

1.6 Meta-Model Driven Measurement

The Meta-Model Driven Measurement (M2DM) approach is a technique introduced in [Abr01], where the concepts of the domain to measure and their relations are first specified in a Unified Modeling Language (UML) [OMG03c] [OMG03a] meta-model. Having this meta-model as a ground, a set of metrics is developed and formalized in the Object Constraint Language (OCL) [OMG03b]. Then, this meta-model is instantiated with the concrete objects to measure, and finally the measurement is done with a UML tool like the USE tool [GBR07]. This technique is depicted at figure 1.2.

The most relevant benefits of this methodology are:

i) the formalization of metrics by using a clarified conceptual content provided by the meta model;

ii) the expressiveness and adequacy of OCL for this purpose;

iii) the usage of UML Class and Object diagrams to represent both the domain and the system to measure as well as the OCL metrics, considering that UML is probably one of the most used languages for the analysis and design of software systems;

iv) the OCL expressions are executable thus eliminating the need for further metrics collection instruments.

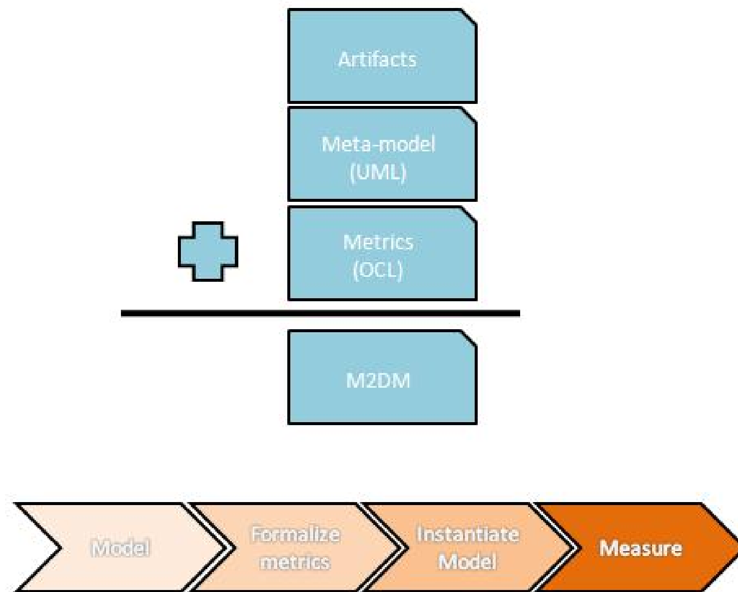

Figure 1.2: Meta-model Driven Measurement Technique

1.7 Contributions of this Thesis

We may summarize the research work discussed in this dissertation as follows:

- We have developed PIMETA, a Paradigm Independent Meta-model for modularity;
- We have researched the modularity concepts and its manifestations to understand how it can be described in terms of the dependencies among software modules and developed PIFACTORS, the Paradigm Independent Modularity Factors;
- We have defined, validated and formalized PIMETRICS, a set of Paradigm Independent Metrics for modularity assessment;
- We have evaluated modularity for the Gang-of-Four Design Patterns functionally equivalent implementations in Java and AspectJ and discussed the results obtained with Aspect-Oriented Programming.

The above contributions can be further detailed as follows:

a) *PIMETA.* This is a meta-model independent from any particular programming paradigm or language which describes the relations between the concepts subjacent to modularity in a system. This meta-model is instantiated, within this research, with the already mentioned design patterns implementation presented in [HK02]

b) *PIFACTORS.* The modularity of a software module can be described in terms of its coupling and cohesion, that is, in terms of the dependencies that exist among the

module and the remainder modules of the system (coupling), and in terms of the dependencies that exist among the inner parts of a module (cohesion). The PIFACTORS are a set of dependency-based properties, created to distinguish the modules from a system based on their modularity. The PIFACTORS are independent of any particular programming paradigm or language.

c) *PIMETRICS.* This set of modularity metrics was created to measure the PIFACTORS and is formalized in OCL upon the PIMETA;

d) *Design Patterns modularity assessment and improvements with AOP.* We have evaluated the modularity at both design pattern implementations using the M2DM methodology with the PIMETA and the PIMETRICS. The functionally equivalent implementations of the Gang-of-Four Design Patterns, implemented in Java and AspectJ by Hannemann and Kiczales, encompass 186 classes, 58 interfaces and 40 aspects. Even though not being real systems, the overall size and diversity of the features implemented and the fact of having been developed by experts is a good guarantee regarding the validity and inference capability of the comparative study between the two programming languages. Also, these implementations have already been used for similar studies, thus increasing the comparability of our results.

1.8 Outline of the Dissertation

This dissertation is organized in a set of chapters which are briefly summarized as follows:

Chapter 2. This chapter presents the PIMETA and its instantiation with OOP and AOP. It also presents its instantiation with the functional equivalent implementations of the Observer Design Pattern in Java and AspectJ, to show how PIMETA can be instantiated with any of these languages and paradigms, and how effectively their modularity differences are evidenced.

Chapter 3. This chapter introduces the PIFACTORS. Here, the different ways features can be distinguished based on the types of dependencies in which they are involved, are thoroughly discussed.

Chapter 4. In this chapter, the PIMETRICS are defined according to the Goal Question Metric methodology, based on the PIMETA and the PIFACTORS, formalized with OCL according to the M2DM methodology, exemplified with the Screen class from the Observer design pattern functional equivalent implementations and finally validated.

Chapter 5. In this chapter the results from the measurement of the GoF design patterns in Java and AspectJ with the PIMETRICS will be presented and discussed.

Chapter 6. In this chapter an overview of the related work is presented and discussed.

Chapter 7. This chapter summarizes the achievements of the research work described in this thesis and lists some open issues, which should and hopefully will ground our future research work.

Chapter 2

PIMETA: Paradigm Independent Meta-model

Contents

2.1 Introduction 12

2.2 Paradigm Independent Meta-model 12

2.3 PIMETA instantiation with OOP and AOP 16

2.4 The Observer Design Pattern in Java and AspectJ 17

2.5 PIMETA instantiation with the Observer versions 18

This chapter presents the PIMETA and its instantiation with OOP and AOP. It also presents its instantiation with the functional equivalent implementations of the Observer Design Pattern in Java and AspectJ, to show how PIMETA can be instantiated with any of these languages and paradigms, and how effectively their modularity differences are evidenced.

2.1 Introduction

In this chapter, a meta-model for describing the structural relationships of software systems, required to perform modularity assessments, across paradigm borders, will be proposed. The constructs of a given development language, either textual or graphical, implementing a paradigm, are mapped to elementary paradigm-independent concepts. Those concepts (e.g. features and dependencies) are represented within a single layer of abstraction, instead of being scattered throughout several layers, like for instance in a system modeled with several UML diagrams.

The meta-model is instantiated either from source code or from design models using appropriate loaders (transformers). The resulting model can be assessed on a paradigm independent fashion, since the meta-model only includes generic concepts, which are omnipresent in all paradigms. Since the meta-model is expressed as an UML meta-class diagram, modularity metrics can then be defined over the meta-model using a formal language, the Object Constraint Language (OCL) [OMG03b], like proposed in [Abr01]. OCL is also used for expressing well-formedness rules for the meta-model itself. To add support for a new language, or paradigm, a mapping of its specific constructs to the core concepts represented in the meta-model can be proposed. This possibility makes the meta-model based approach to metrics definition flexible.

In the next section the paradigm-independent meta-model(PIMETA) [BA07] is presented.

In section 2.3 the PIMETA will be instantiated with Java and AspectJ, to demonstrate how the specific constructs of different languages and paradigms can be mapped to its elementary modularity concepts.

In section 2.4 the two functional equivalent implementations, in Java and AspectJ, of the Observer Design pattern, are presented. These two languages are good representatives of the object-oriented and of the aspect-oriented paradigms, respectively.

These implementations will be the ground for section 2.5, where PIMETA is instantiated with both versions, to demonstrate how it can be instantiated by any system implemented with either Java or AspectJ, and its effectiveness in evidencing the modularity differences between these languages and the paradigms they represent.

2.2 Paradigm Independent Meta-model

2.2.1 PIMETA description

Modularity is an architectural property with impact on software maintenance and reusability [Pre00]. From a modularity perspective, a system is composed of features, which interact among each other, originating dependencies. Features are modular, and consequently called *modules*, when they have the capability to aggregate other features

to logically organize them, like packages, classes or operations. Features are atomic when they do not allow the aggregation of other features (e.g. a parameter or a class attribute).

The meta-model depicted in figure 2.1 can be logically divided into two different parts. This division is represented by the double line and is inspired by the *powertypes* concept [Hal04].

The upper part, composed by the *Paradigm*, *Language*, *FeatureType*, *ModularFeatureType*, *AtomicFeatureType* and *DependencyType* meta-classes, allows representing paradigms and languages, the kind of features and dependencies they offer, and how they can be organized. The *Paradigm* meta-class is meant to represent paradigms like OOP or AOP. The *Language* meta-class represents languages like Java or AspectJ. Each language implements a specific paradigm, and one language can extend another (e.g. AspectJ extends Java), thus enabling the reuse of the base language definitions. The abstract meta-class *FeatureType* represents the types of features that languages offer. *ModularFeatureType* represents the different modules offered by languages (e.g. the class and package modules are offered by Java). A *ModularFeatureType* may contain any other *FeatureType*(e.g. a class may contain an attribute or another class in Java). *AtomicFeatureType* represents all non-modular features offered by languages (e.g. interface declaration or inheritance implementation are offered by Java). Finally, the *DependencyType* meta-class represents the different dependency types that can exist among features, provided by languages (e.g. an operation calls another operation in Java).

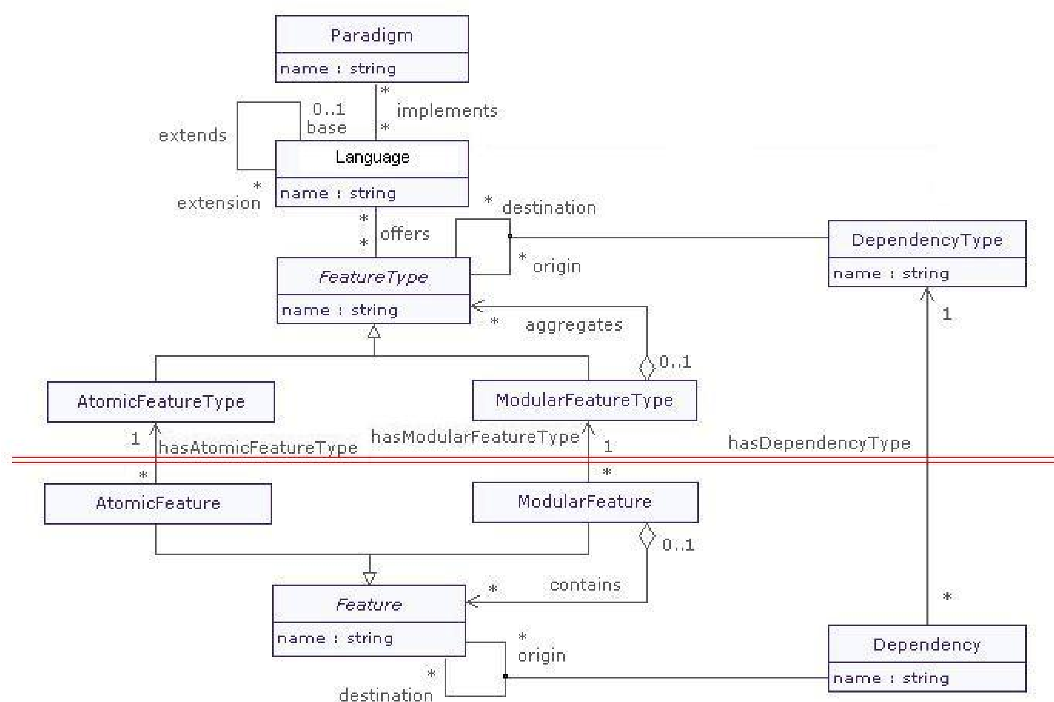

Figure 2.1: PIMETA Meta-class diagram

The lower part, composed by the *Feature*, *ModularFeature*, *AtomicFeature* and *Dependency* meta-classes, allows the representation of the system to be analyzed, its concrete features, dependencies and organization. The abstract meta-class *Feature* represents the features from the system under analysis. The *ModularFeature* meta-class is meant to be instantiated with the modules from the system under analysis (e.g. system classes or packages if Java is the language). A *ModularFeature* may contain any *Feature*, like a class may contain an attribute or another class in Java. The *AtomicFeature* meta-class should be instantiated by the features which are not modular, from the system under analysis, like an instance variable if Java is, again, the language used. The *Dependency* meta-class should be instantiated with the dependencies that exist in the system under analysis, like an operation calling another operation in a Java system. *ModularFeature*, *AtomicFeature* and *Dependency* instances always have a type from the corresponding meta-classes of the upper part of the meta-model.

2.2.2 PIMETA well-formedness rules

To ensure the meta-model's consistency, several well-formedness rules need to be expressed. These were formalized as OCL invariants and can be found at figure2.2. This is the same technique as that used in the UML series of standards for defining the corresponding meta-model [OMG03c] [OMG03a].

```
1   Language
2   // a language cannot extend itself either directly or indirectly
3   inv: not self.circularBaseLanguages()
4
5   FeatureType
6   // there are no circular aggregations (either direct or indirect)
7   inv: not self.circularModularFeatureTypes()
8
9   DependencyType
10  // a language must provide the feature types involved in a given dependency
11  inv: language.featureType->includesAll(Bag{self.origin, self.destination})
```

Figure 2.2: PIMETA well-formedness rules in OCL

The OCL invariants make use of several auxiliary operations, also defined in OCL, which can be found at figure 2.3.

```
13
14
15  // auxiliary operation to support implementing circularBaseLanguages()
16  Language::circularBaseLanguagesAux(setF:Set(Language)):
17      Bag(Language) =
18  if (setF->includes(self.base)) then
19      setF->asBag()->including(self.base)
20  else
21      self.base.circularBaseLanguagesAux(setF->including(self.base))
22  endif
23
24  // predicate that detects if a circular base reference exists
25  Language::circularBaseLanguages(): Boolean =
26      circularBaseLanguagesAux(oclEmpty(Set(Language)))->size() <>
27      circularBaseLanguagesAux(oclEmpty(Set(Language)))->asSet()->size()
28
29  // aux. operation to support implementing circularModularFeatureTypes()
30  FeatureType::circularModularFeatureTypesAux
31  (setF: Set(ModularFeatureType)) : Bag(ModularFeatureType) =
32  if (setF->includes(self.modularFeatureType)) then
33      setF->asBag()->including(self.modularFeatureType)
34  else
35      self.modularFeatureType.circularModularFeatureTypesAux
36  (setF->including(self.modularFeatureType))
37  endif
38
39  // predicate that detects if a circular aggregation exists
40  FeatureType::circularModularFeatureTypes(): Boolean =
41      circularModularFeatureTypesAux
42  (oclEmpty(Set(ModularFeatureType)))->size() <>
```

Figure 2.3: PIMETA well-formedness rules auxiliary operations in OCL

2.3 PIMETA instantiation with OOP and AOP

To demonstrate how PIMETA can be used for representing software systems designed (input is a model) or written (input is source code) in different languages and paradigms, a full instantiation has been performed, for the Java and AspectJ languages.

In the following subsections, fragments of that instantiation are presented, represented as UML meta-object diagrams. The complete instantiation cannot be shown due to space limitation, since the corresponding diagrams are very large. Nevertheless, it is believed that the represented extracts are sufficient for a good understanding of the instantiation process.

2.3.1 PIMETA instantiation with OOP/Java

Figure 2.4 exhibits a partial representation of the PIMETA instantiation with the OOP paradigm, the Java language which implements it, as well as some of its features, namely the *Package* which aggregates other packages, and the *Class* which, in turn, aggregates other classes and variables, and the type of a variable, represented by the variable type dependency type binding the variable and the class which provides its type.

Further details on how and which *Features* and *Dependencies*, from the Java language, have been instantiated at PIMETA, can be found at appendix A.

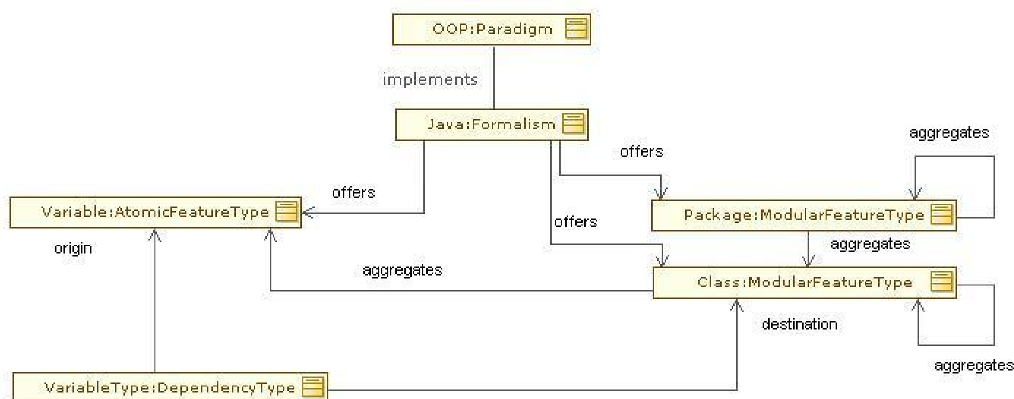

Figure 2.4: PIMETA instantiation extract for Java

2.3.2 PIMETA instantiation with AOP/AspectJ

Figure 2.5 shows a fragment of the resulting meta-object diagram from the PIMETA instantiation with AOP and AspectJ. In the history of programming languages and also of modeling languages, many proposals were made to extend existing languages while guaranteeing backward compatibility. Well-known cases are C++ (compatible with C) and AspectJ (compatible with Java). PIMETA provides a language extension mechanism (reflexive meta-dependency *extends* in figure 2.1). This mechanism allows reusing the meta-objects created for Java in the context of AspectJ. The specific features of AspectJ are the only ones that must then be added, such as the fact that a *package* may contain *aspects* and that an *aspect* may contain *variables*. The dependency corresponding to an *aspect* inheriting from a *class* is also represented, through the *AspectInheritsFromClass* meta-object.

Further details on how and which *Features* and *Dependencies*, from the AspectJ language, have been instantiated at PIMETA, can be found at appendix B.

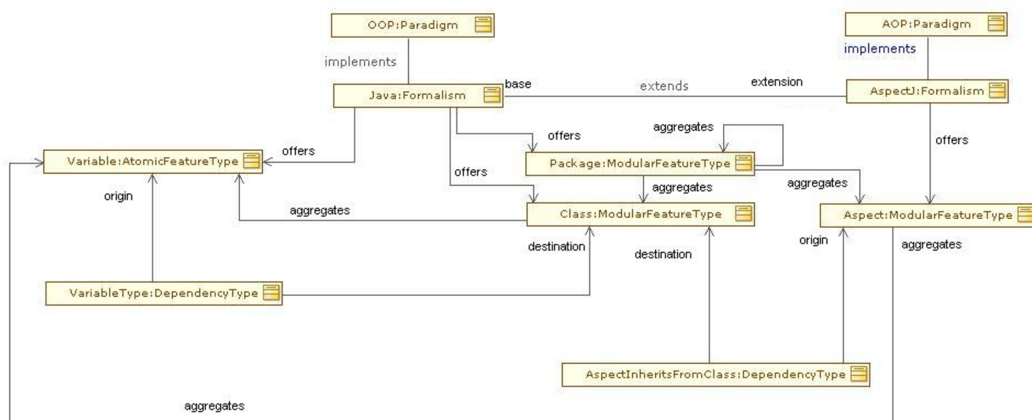

Figure 2.5: PIMETA instantiation extract for AspectJ

2.4 The Observer Design Pattern in Java and AspectJ

The Observer Design Pattern [GHJV95], implemented both with Java and AspectJ by Hannemann and Kiczales [HK02], was chosen to show how PIMETA can be instantiated with systems from any of these two different languages and paradigms, and how their modularity differences can effectively be put in evidence by PIMETA. This choice has the following advantages:

i) its small size facilitates understanding;

ii) it complements the PIMETA instantiation examples started in previous subsections;

iii) the instantiated systems are functionally equivalent;

iv) the Observer pattern is a paradigmatic example of a way to avoid high coupling;

v) it was developed by experts with the purpose of putting in evidence the differences between both paradigms (OOP and AOP).

However, this example only encompasses a small subset of the features offered by both languages. Another disadvantage is the fact of not being a real world example.

The UML class diagram of the Observer Design Pattern implementation in Java is presented in figure 2.6. The UML class diagram of the Observer Design Pattern implementation in AspectJ is presented in figure 2.7.

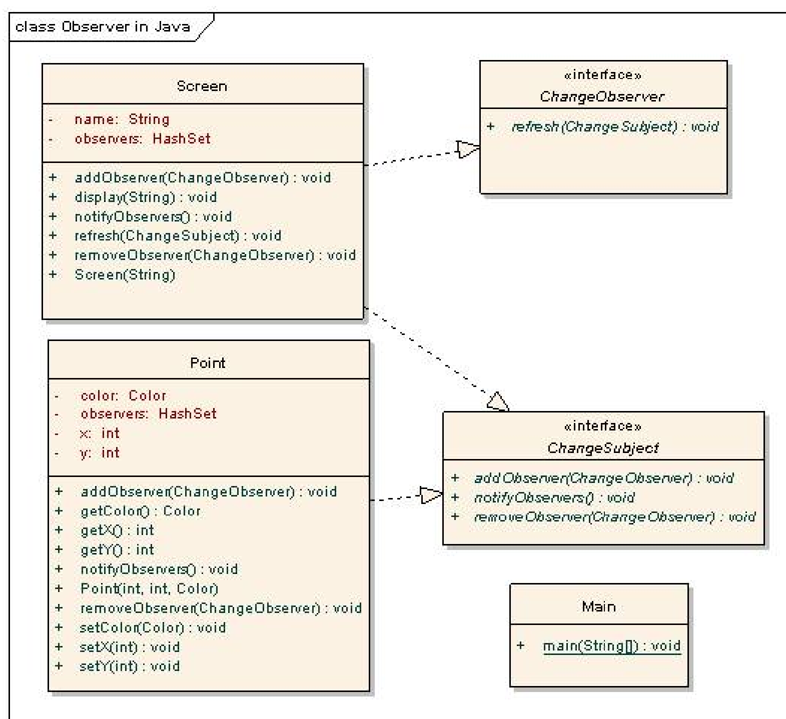

Figure 2.6: Observer - Java version class diagram

2.5 PIMETA instantiation with the Observer versions

2.5.1 PIMETA instantiation with the Java version of the Observer

To demonstrate how PIMETA can be instantiated by a Java system, PIMETA was instantiated with the Java version of the Observer design pattern. An extract of that instantiation is shown at figure 2.9 where only the *Screen* class is represented, also due to space constraints. For simplicity and understandability reasons, we chose not to represent this instantiation as an object diagram but, instead, to represent it as a graph, based on the PIMETA meta-classes. For the same reasons, several objects are

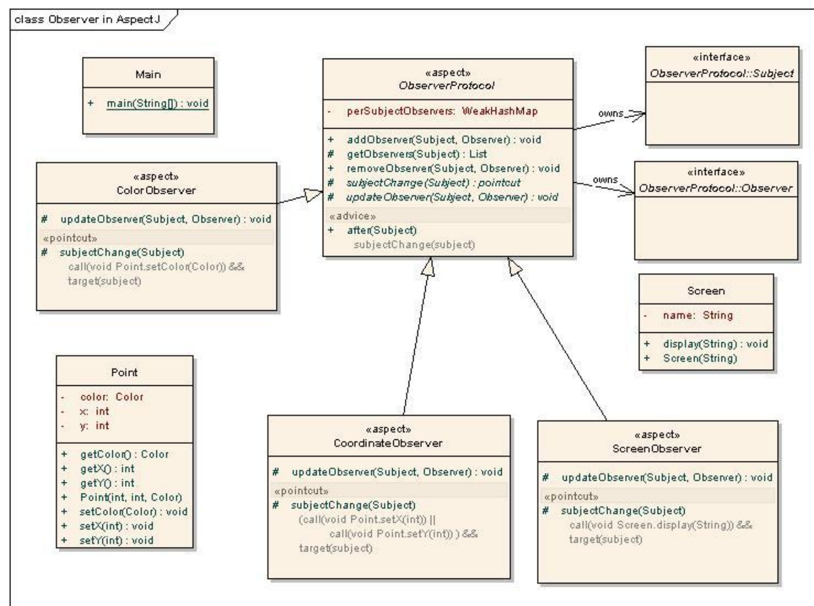

Figure 2.7: Observer - AspectJ version class diagram

not shown, namely the Java objects needed by this instantiation, the profuse *DependencyType* meta-objects corresponding to the *Dependency* meta-objects represented, and the dependencies' names.

The code corresponding to the *Screen* class from the Java implementation of the Observer design pattern is fully presented at figure 2.8. At this example, depicted in figure 2.9 there are atomic features, represented by a circle (i.e. *Screen.name*), modular features represented by a square (i.e. *Screen.display*) and features with an undetermined[1] type are represented by a polygon (i.e. *Iterator*).

Each solid black line represents a containment relation, that is, each feature that contains another feature has a solid black line with an arrow pointing towards it (i.e. *Screen* contains *Screen.display*). Each dashed line represents a dependency, that is, when one feature dependends on another, a dashed line is drawn from the dependent towards the depended with an arrow pointing towards it (i.e. Screen.display calls System.out.println).

The big grey box represents the border of the *Screen* class. All the features outside this box do not belong to the *Screen* class, while the opposite is also true. In this example, all the dependencies that leave the box are responsible for the coupling of the *Screen* class, while those that remain inside the box are responsible for its cohesion.

For clarification, all features have their names fully qualified according to their container (i.e. *Screen.Screen* is the constructor of the *Screen* class). Also should be noticed that the *Screen* class is represented inside its own border for two reasons. One is for coherence on the graphical representation of the containment relations, and the other is because the class itself can be part of dependencies that may contribute either for coupling or cohesion, and having the class inside its own border makes it easy to see

[1] Undetermined due to limitations of the automatic process which generated this graph

those that are directed towards the outside (coupling) and those that remain in the inside (cohesion).

This notation for the representation of features and dependencies from a system as a graph will be used at similar examples throughout this dissertation.

Each solid black line represents a containment relation, that is, each feature that contains another feature has a solid black line with an arrow pointing towards it. Each dashed line represents a dependency from the dependent towards the depended.

The big gray box represents the border of the *Screen* class. None of the features outside this box belong to the *Screen* class, while all the ones inside it do. In this example, all the dependencies that leave the box are responsible for the coupling of the *Screen* class, while those that remain inside the box are responsible for its cohesion.

```
1  import java.util.HashSet;
2  import java.util.Iterator;
3  public class Screen implements ChangeSubject, ChangeObserver {
4
5      private HashSet observers;
6      private String name;
7
8      public Screen(String s) {
9          this.name = s;
10         observers = new HashSet();
11     }
12
13     public void display (String s) {
14         System.out.println(name + ": " + s);
15         notifyObservers();
16     }
17
18     public void addObserver(ChangeObserver o) {
19         this.observers.add(o);
20     }
21
22     public void removeObserver(ChangeObserver o) {
23         this.observers.remove(o);
24     }
25
26     public void notifyObservers() {
27         for (Iterator e = observers.iterator() ; e.hasNext() ;) {
28             ((ChangeObserver)e.next()).refresh(this);
29         }
30     }
31
32     public void refresh(ChangeSubject s) {
33         String subjectTypeName = s.getClass().getName();
34         subjectTypeName = subjectTypeName.substring(
35             subjectTypeName.lastIndexOf(".")+1, subjectTypeName.length());
36         display("update received from a "+subjectTypeName+" object");
37     }
38 }
```

Figure 2.8: Screen class source code from the Observer pattern in Java

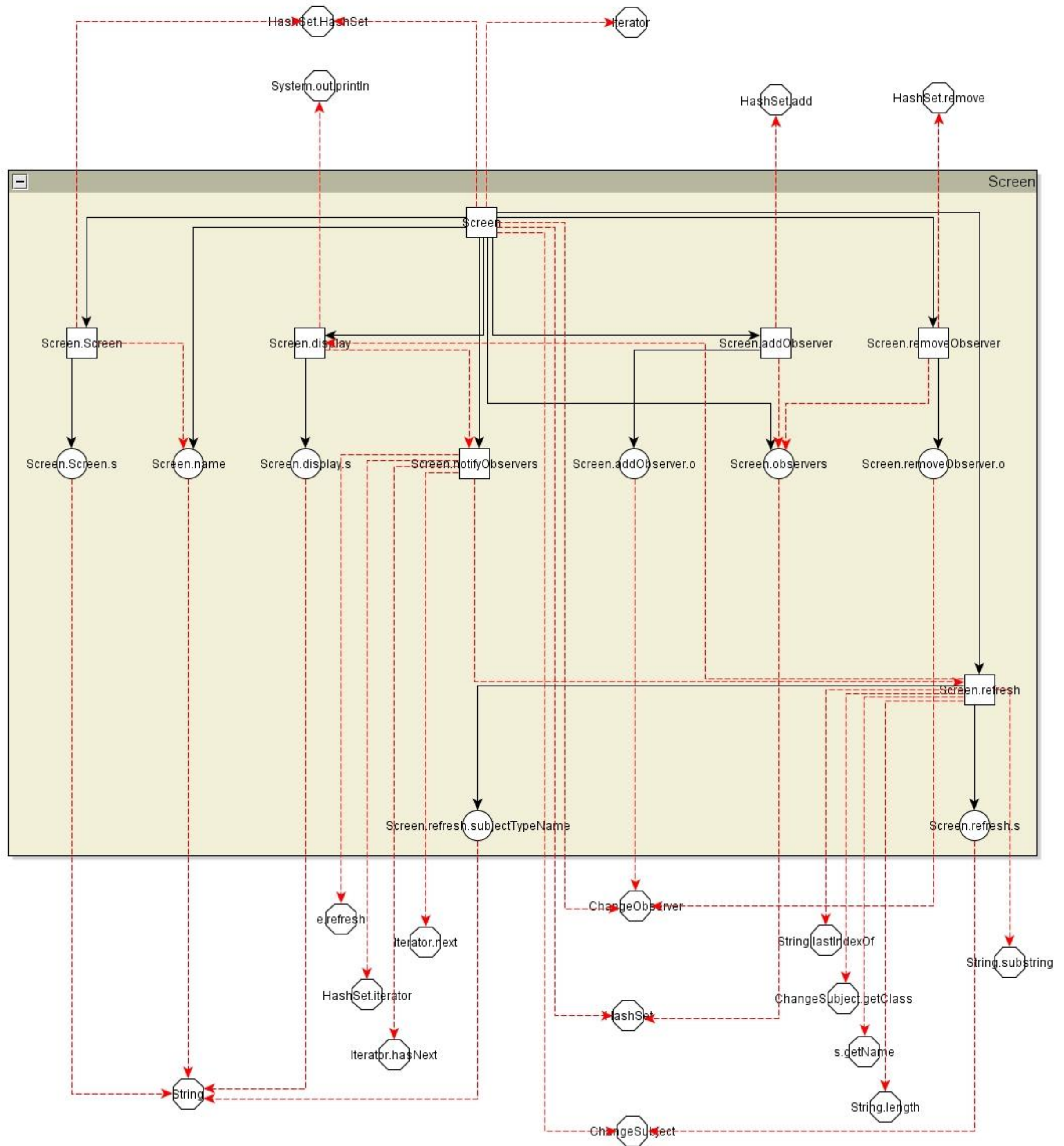

Figure 2.9: PIMETA instantiation with the *Screen* class of the Observer in Java

2.5.2 PIMETA instantiation with the AspectJ version of the Observer

To demonstrate how PIMETA can be used for representing an AspectJ system, PIMETA was instantiated with the AspectJ version of the Observer design pattern. An extract of that instantiation is shown at figure 2.11 where only the *Screen* class is represented[2].

Several objects are also not shown, namely, the AspectJ objects needed by this instantiation, the profuse *DependencyType* meta-objects corresponding to the *Dependency* meta-objects represented, and the dependencies' names.

The code corresponding to the *Screen* class from the AspectJ implementation of the Observer design pattern is fully depicted at figure 2.10.

By looking at figures 2.9 and 2.11, the modularity differences introduced by AOP at this concrete pattern implementation are perceivable, being this capability one of the benefits of PIMETA.

```
1
2  public class Screen {
3
4      private String name;
5
6      public Screen(String s) {
7          this.name = s;
8      }
9
10     public void display (String s) {
11         System.out.println(name + ": " + s);
12     }
13 }
14
```

Figure 2.10: Screen class source code from the Observer pattern in AspectJ

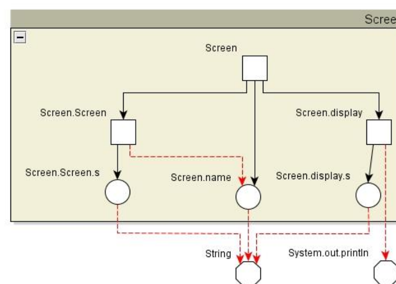

Figure 2.11: PIMETA instantiation with the Screen class of the Observer in AspectJ

[2]The notation used is the same described at section 2.5.1

Chapter 3

PIFACTORS: Paradigm Independent Modularity Factors

Contents

3.1 Introduction 24
3.2 Taxonomy for dependency topologies 25

This chapter introduces the PIFACTORS. Here, the different ways features can be distinguished based on the types of dependencies in which they are involved, are thoroughly discussed.

3.1 Introduction

As mentioned in chapter 1, modularity can be assessed using the concepts of coupling and cohesion which, in turn, are the result of the dependencies that exist among the features from a software system. So, to evaluate modularity, these dependencies must be evaluated.

If, when evaluating the modularity of a feature, the dependencies under consideration are those towards other features rather than the one being considered and those it may contain, then we are talking about coupling. If, on the other hand, the dependencies under consideration are only those between the features that are contained within the feature being considered, then we are talking about cohesion.

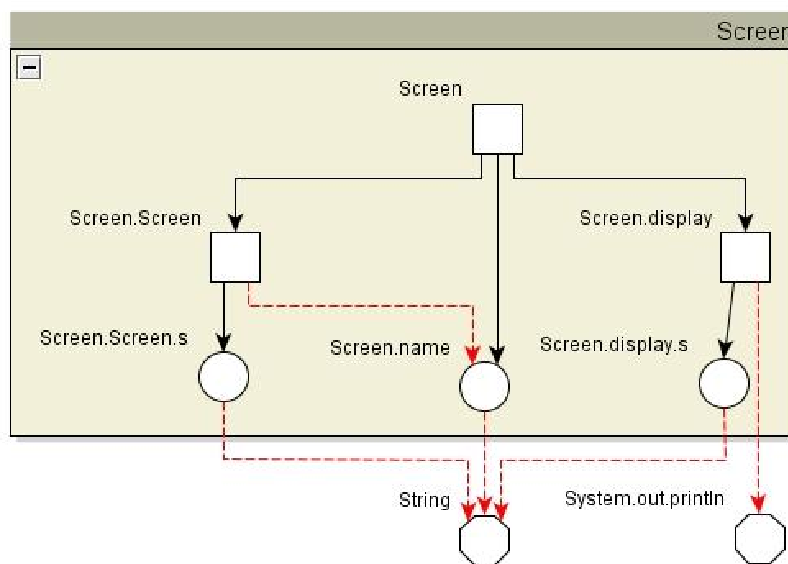

Figure 3.1: Aspectized Screen class dependencies graph

Take for instance the example at figure 3.1[1]. In it, we can observe a graph with the dependencies of the *Screen* class from the AspectJ implementation of the Observer Design Pattern.

If we are measuring the modularity of a Java class, then its coupling comes from the dependencies between it and the remainder features of the system, not contained by it (i.e. other classes), and cohesion comes from the dependencies among the features it contains (i.e. methods).

However, if we raise the level of abstraction and try to measure the modularity of a Java package, then its coupling comes from the dependencies between it and the remainder features of the system not contained by it (i.e. other packages), while its cohesion comes from the dependencies among the features it contains (i.e. Internal Java classes coupling). When we change the level of abstraction, we notice that the border

[1]Uses the same notations as those described in section 2.5.1

between coupling and cohesion is the same as the border of the feature under consideration. Also, if we consider, for any feature, that it only has dependencies among the features it contains and dependencies towards external features, then we can say that coupling and cohesion are complementary.

Several ways have been proposed for improving cohesion(increasing it) and coupling(reducing it) for the object-oriented paradigm. The application of design patterns [GHJV95] is one of them. More recently, object-oriented programming was *refactored* to accomodate a new modularization paradigm as described in section 1.5. AOP, by encapsulating the crosscutting concerns in a special kind of modules called aspects, claims to improve modularity by improving coupling and cohesion. Since coupling and cohesion result from dependencies among the features of a software system, we should study the different manifestations of those dependencies in terms of how the corresponding features participate in them.

3.2 Taxonomy for dependency topologies

Dependencies can be analyzed in many ways. The following PIFACTORS are ways to analyze these dependencies with the purpose to differentiate the features from a software system.

3.2.1 Role

A dependency always encompasses a pair of features, even though sometimes these may be the same. The dependent feature is called *source*, and the depended is called *target*.

The role factor regards to the part a feature plays at each dependency in which it is involved. A feature acting as a source must know about the features on which it depends, because those may affect its purpose. On the other hand, a feature acting as a target should be aware of its probable impact at the features which rely on it.

Features contribute to the overall modularity when they act as sources or targets, thus it is important to understand how a feature contributes to modularity from both roles. Besides, a feature might also exclusively act as a source or as a target. So, if one of the roles is disregarded, some features may not be considered when evaluating modularity, in spite of playing a part in it.

At the following factors descriptions, each factor will be explained considering both perspectives (source and target) a feature can have from a dependency. For a better understanding of how each factor can distinguish features by itself, the previously explained factors will always be made equal at those features. For instance, when looking at commitment, coverage is made equal at the examples, as not to interfere with the modularity evaluation among the features, which will only be done with the commitment factor, in this case.

3.2.2 Coverage

The coverage factor distinguishes features based on how many other different features are involved within its dependencies.

If coverage is being considered for coupling evaluation then, from the source feature perspective, the more different features a feature uses, the more that feature is coupled.

On the other hand, if coverage is being considered for cohesion evaluation, the higher the coverage of the feature which is being analyzed, the higher is its contribute to the cohesion of the modular feature to which it belongs, as long as it shares its functional concern with the targets.

At Figure 3.2 there are three modular features represented by the gray squares (A, B and C), each of them containing two features represented by the polygons(A1, A2, B1, B2, C1 and C2). There are also four dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on C1, A2 depends on C2, B1 and B2 depend on C1. At this example, modular feature A has more coverage (2), as a source, than modular feature B (1).

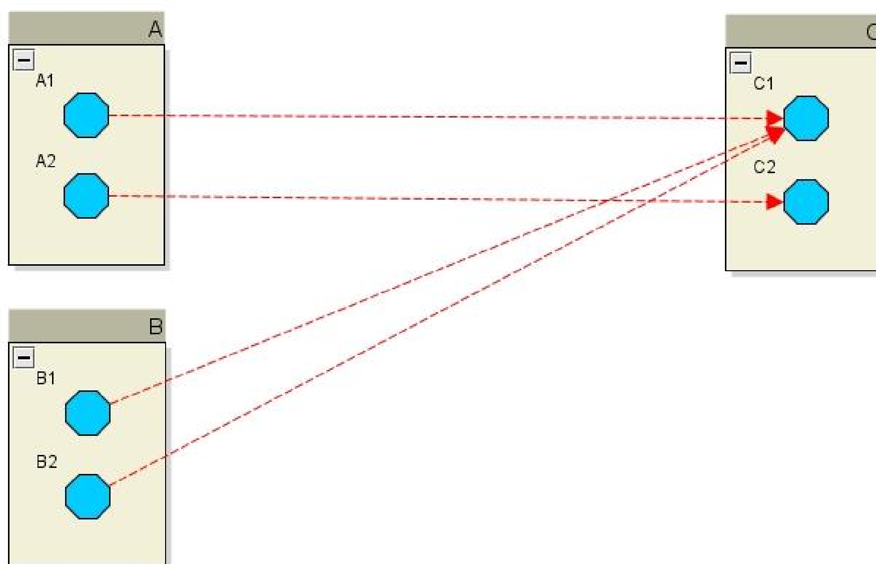

Figure 3.2: Source Coverage example

From the target feature perspective, if coverage is being considered for coupling evaluation then, the more different features use it, the higher is its importance to the overall coupling.

On the other hand, if coverage is being considered for cohesion evaluation, then the higher the coverage of the feature which is being analyzed, the higher is its contribution to the cohesion of the modular feature to which it belongs, as long as it shares its functional concern with the sources.

In Figure 3.3 there are three modular features represented by the squares(A, B and C), each of them containing two features represented by the polygons(A1, A2, B1, B2,

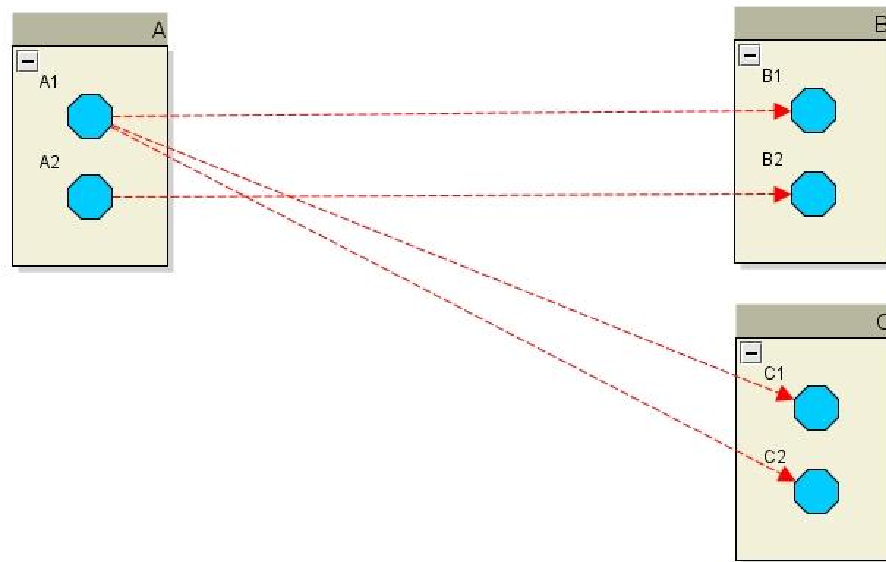

Figure 3.3: Target Coverage example

C1 and C2). There are also four dependencies represented by the dashed lines. A1 depends on B1, C1 and C2, and A2 depends on B2.

At this example, modular feature B has more coverage (2), as a target, than modular feature C (1).

3.2.3 Commitment

The commitment factor expresses how much of a module is committed to the dependencies in which it is involved.

If commitment is being considered for coupling evaluation then, the more different features, within this modular feature are involved as sources in dependencies towards external features, the more this modular feature is coupled.

On the other hand, if commitment is being considered for cohesion evaluation then, the more different features, within this modular feature, are involved as sources in dependencies towards internal features, the more this modular feature is cohesive, as long as they share their functional concern with their targets.

At Figure 3.4 there are three modular features represented by the squares (A, B and C), each of them containing two features represented by the polygons (A1, A2, B1, B2, C1 and C2).

There are also four dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on C1, A2 depends on C2, B1 depends on C1 and C2. In this example, modular feature A has more commitment (2), as a source, than modular feature B (1). Notice that coverage, from the source perspective, has been made equal for modular features B and A.

If commitment is being considered for coupling evaluation then, the more different features, within this modular feature, are involved as sources in dependencies towards

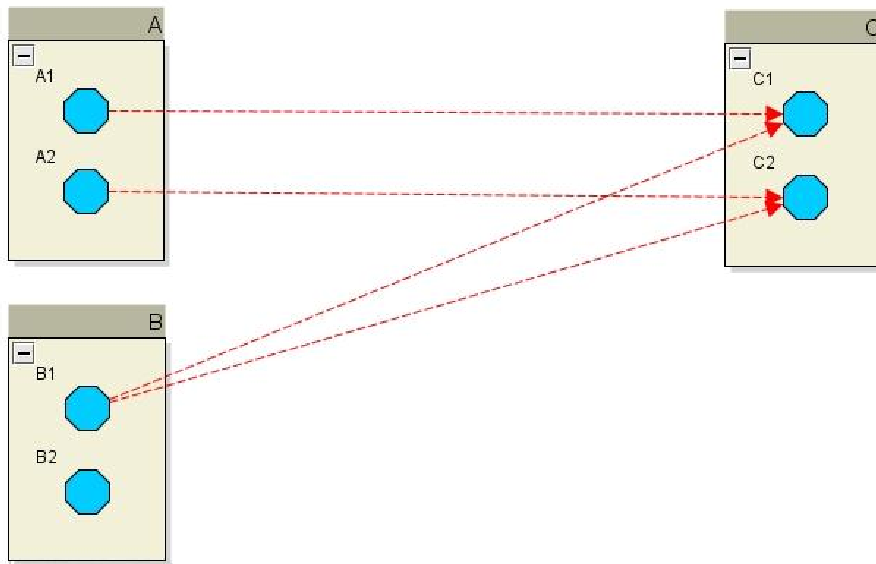

Figure 3.4: Source Commitment example

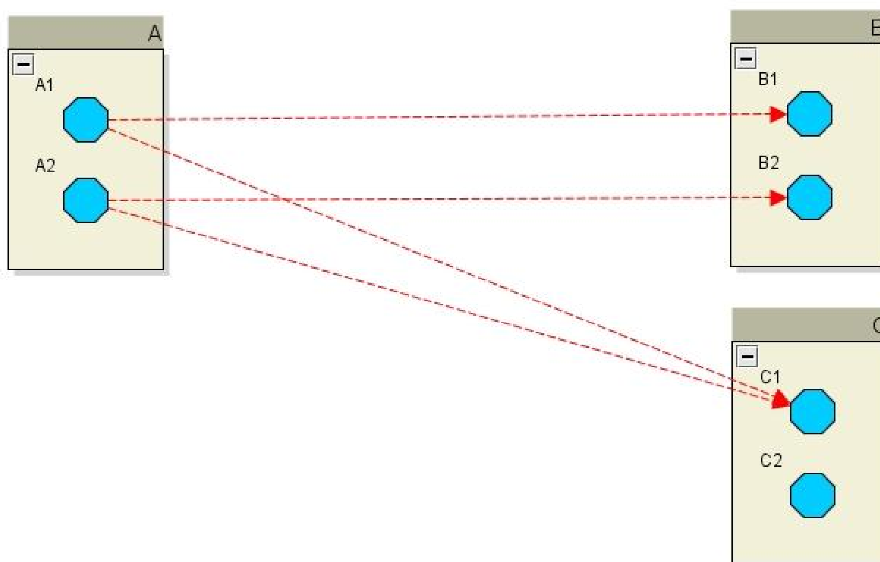

Figure 3.5: Target Commitment example

external features, the more this modular feature is coupled.

On the other hand, if commitment is being considered for cohesion evaluation then, the more different features, within this modular feature, are involved as sources in dependencies towards internal features, the more this modular feature is cohesive, as long as they share their functional concern with their targets.

In figure 3.5 there are three modular features represented by the squares(A, B and C), each of them containing two features represented by the polygons (A1, A2, B1, B2, C1 and C2). There are also four dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on B1 and C1, A2 depends on B2 and C1. In this example, modular feature B has more commitment (2), as a target, than modular feature C (1). Notice that coverage, from the target perspective, has been made equal for modular features B and C.

3.2.4 Clustering

The clustering factor distinguishes features based on how many disjoint clusters are involved within the dependencies at which they are involved. Disjoint clusters are modular features that do not share any of the modular features that contains them (i.e. a system composed by two packages. Each package is a cluster disjoint from the other).

Clustering only makes sense when evaluating coupling, once that for cohesion, only the dependencies from (and to) the features within the same modular feature are considered.

The more different disjoint clusters contain targets of dependencies, the more the feature under evaluation is coupled.

At figure 3.6 there are two disjoint clusters represented by the big dotted rectangles (X and Y). At cluster X, there are three modular features represented by the gray squares(A, B and D). Modular Features A and B contain three features (A1, A2 and A3) and (B1, B2 and B3) respectively, while modular feature D contains only two (D1 and D2). At cluster Y, there is only one modular feature (C) containing also only one feature (C1). Each of the previously mentioned features are represented by the polygons. There are also four dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on C1, A2 depends on D1, B1 depends on A3 and B2 depends on D2. In this example, modular feature A has more clustering (1), as a source, than modular feature B (0). Notice that coverage and commitment, from the source perspective, have been made equal for modular features B and A.

The more different disjoint clusters contain features that use the feature under evaluation, the higher is its importance to the overall coupling.

At Figure 3.7 there are two disjoint clusters represented by the big dotted rectangles (X and Y). At cluster X, there are two modular features represented by the squares(A, and B). At cluster Y, there is only one modular feature (C). Modular Features C and B contain only one feature (C1 and B1) respectively, while modular feature A contains

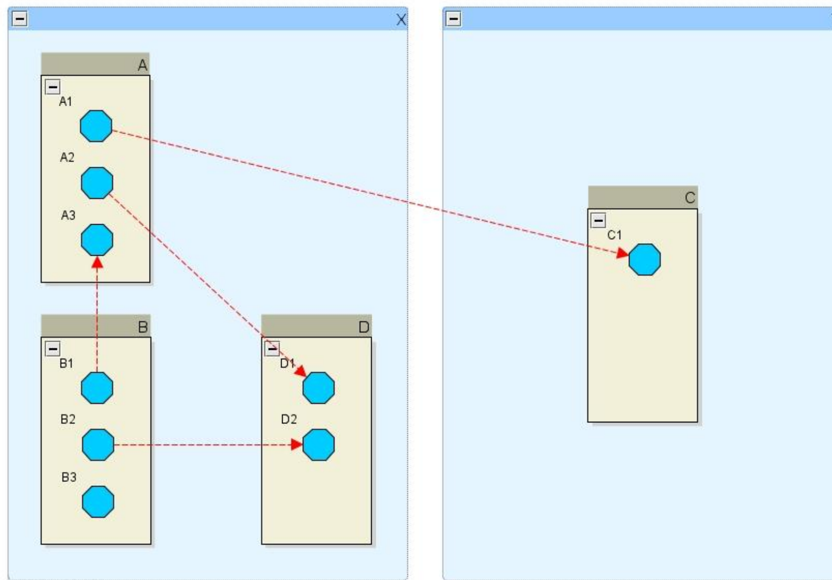

Figure 3.6: Source Clustering example

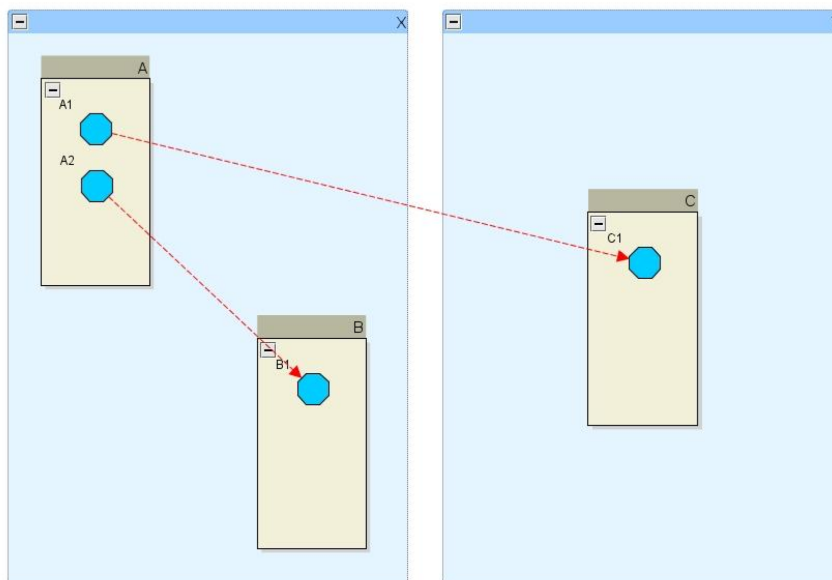

Figure 3.7: Target Clustering example

two (A1 and A2). Each of the previously mentioned features are represented by the polygons. There are also two dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on C1, A2 depends on B1. In this example, modular feature C has more clustering (1), as a source, than modular feature B (0). Notice that coverage and commitment, from the target perspective, have been made equal for modular features B and C.

3.2.5 Density

The density factor distinguishes features based on how many instances a dependency has. Only design-time instances are being considered (i.e. a method that within its code calls another method several times).

If density is being considered for coupling evaluation then, from the source feature perspective, the more instances a dependency has, the more that feature is coupled.

On the other hand, if density is being considered for cohesion evaluation, the higher the density of the dependencies from the feature which is being analyzed, the higher is its contribute to the cohesion of the modular feature to which it belongs, as long as it shares its functional concern with the targets.

At Figure 3.8 there are three modular features represented by the squares(A, B and C), each of them containing two features represented by the polygons (A1, A2, B1, B2, C1 and C2). There are also five dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on C1 (twice), A2 depends on C2, B1 and B2 depend on C1 and C2 respectively. In this example, modular feature A has more density (3), as a source, than modular feature B (2). Notice that coverage, commitment and clustering, from the source perspective, have been made equal for modular features A and B.

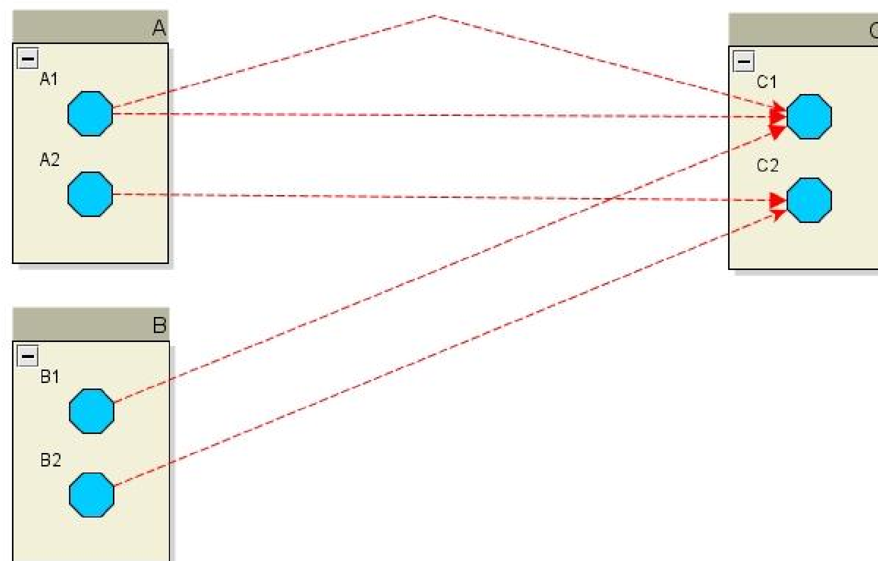

Figure 3.8: Source Density example

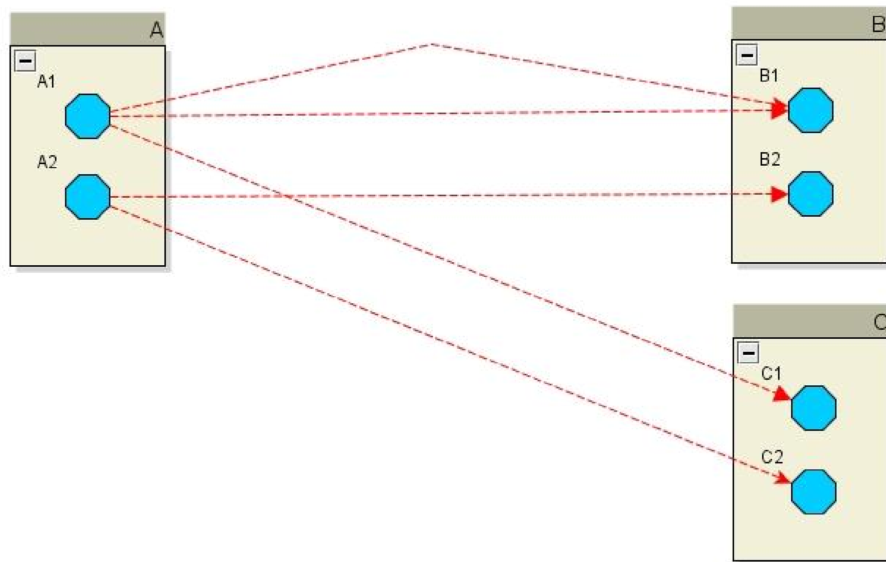

Figure 3.9: Target Density example

From the target feature perspective, if density is being considered for coupling evaluation then, the more different instances from a dependency use it, the higher is its importance to the overall coupling.

On the other hand, if density is being considered for cohesion evaluation, then the higher the density of the feature which is being analyzed, the higher is its contribute to the cohesion of the modular feature to which it belongs, as long as it shares its functional concern with the sources.

At Figure 3.9 there are three modular features represented by the squares(A, B and C), each of them containing two features represented by the polygons (A1, A2, B1, B2, C1 and C2). There are also five dependencies represented by the dashed lines, where the target is indicated by an arrow. A1 depends on B1(twice) and C1, and A2 depends on B2 and C2. In this example, modular feature B has more density (3), as a target, than modular feature C (2). Notice that coverage, commitment and clustering, from the target perspective, have been made equal for modular features B and C.

3.2.6 Type

Each paradigm and programming language has its own set of features, each one implementing dependencies towards other features in many different ways.

We believe that each way these dependencies are implemented does not necessarily produce the same impact at modularity, and that this impact can be estimated based on a statistical analysis on maintenance effort. Part of this idea is based at the traditional coupling and cohesion types [Pre00].

However, these traditional modularity types are represented in an ordinal scale, while it is believed that the different dependencies types for any programming language can be represented in a continuous scale. This representation is out of the scope

of this dissertation, however a *weight* attribute for dependencies has already been reserved at the PIMETA, which will most certainly be used at future work. For the moment, this weight will be set to 1 for any dependency, as a default value, assuming that their influence at modularity is equal.

[This page was intentionally left blank]

Chapter 4

PIMETRICS: Paradigm Independent Metrics

Contents

4.1 Introduction 36

4.2 Definitions 37

4.3 Formalizations 43

4.4 Metrics collection 50

4.5 Exemplification 51

4.6 Validation 56

In this chapter, the PIMETRICS are defined according to the Goal Question Metric methodology, based on the PIMETA and the PIFACTORS, formalized with OCL according to the M2DM methodology, exemplified with the Screen class from the Observer design pattern functional equivalent implementations and finally validated.

4.1 Introduction

To enable meta-model driven measurement (M2DM) [Abr01], two core artifacts are required, besides the system to measure:

i) a meta-model;

ii) a set of metrics expressed as a set of OCL operations upon the preceding meta-model.

While in the previous chapters, the meta-model (PIMETA) and the concepts which embody it were presented, in this chapter, a set of metrics will be defined, formalized, exemplified and validated, with the purpose of evaluating modularity independently of paradigm using the M2DM.

Along this chapter, Abreu's criteria for the development of software metrics [AC94] are ever present concerns:

i) metrics determination should be formally defined;

ii) non-size metrics should be system size independent;

iii) metrics should be dimensionless or expressed in some consistent unit system;

iv) metrics should be obtainable early in the life cycle;

v) metrics should be down scalable;

vi) metrics should be easily computable;

vii) metrics should be language independent;

The Goal-Question-Metric (GQM) paradigm [BCR94] has proven to be an effective approach to selecting and implementing metrics, is widely adopted for this purpose and, as such, will be used to define the PIMETRICS. Succintly, this process consists in defining metrics in a top-down approach by first declaring the goals to be met, then posing the questions which answers will allow to understand if the goals are being met and, last but not least, identifying the metrics required to answer the questions. While ensuring that the identified metrics are relevant to understand if the goals are being met, the GQM also provides traceability between them and the metrics.

According to the *Representational Theory of Measurement* [FP97], the direct measurement of an attribute must be preceeded by its intuitive understanding. Since that the PIFACTORS have been developed to meet this, the metrics definition wil be driven by those modularity factors.

Any metric should be formalized, for the simple reason that when this does not occur, a door is open to different interpretations which, among other things, will almost

surely lead to different results for the same subject. Obviously this is not acceptable but, unfortunately, metrics proposal without an adequate formalization is still a common practice, in spite of adequate alternatives have already been proposed, like using the OCL [OMG03b] language to formalize metrics upon UML models [Abr01]. This approach will be used in section 4.3, where the PIMETRICS will be formalized in OCL upon the PIMETA meta-model.

In section 4.5, an example of the PIMETRICS will be presented, which will be used to quantify the modularity of the *Screen* class of the Observer design pattern implementations in Java and AspectJ. The metrics collection was made as described in section 4.4.

Last but not least, a fundamental step in metrics development is making sure that metrics actually assess the right attribute, because making decisions with wrong information often leads to wrong decisions and this will most surely happen if we measure the wrong attributes. The process by which this is ensured is called *validation*, and involves this demonstration in the sense of the measurement theory which, in this case, means that the PIMETRICS must preserve the intuitive notions about modularity, as described by the PIFACTORS. This process will be addressed at section 4.6.

4.2 Definitions

The PIMETRICS will be defined in this section, as stated before, by following the GQM paradigm.

Since that the main purpose of this dissertation is to evaluate modularity, the GQM goals are defined in terms of modularity and its main drivers, coupling and cohesion. The questions are elaborated in terms of the PIFACTORS, because these are what it is believed to affect coupling and cohesion. Finally, the metrics are defined with the PIMETA concepts.

Since that in PIMETA there are two kinds of features, those that are modular, and those that are not, the atomic features, and that the PIFACTORS only apply partially to the latter, namely those for cohesion, the goals have been divided into those applicable only to modular features and those applicable to any feature or the whole system in general. These goals are summarized in table4.1 and described from table 4.2 to table 4.5.

Goals
G1 - Decrease features coupling
G2 - Decrease modular features coupling
G3 - Increase modular features cohesion
G4 - Increase systems modularity

Table 4.1: Goals summary

G1 - Goal: Decrease features coupling		
Purpose	Decrease	
Issue	Coupling	
Object	Features	
Viewpoint	Software Engineer	
Questions and Metrics		
Q1.1 - What is the coupling coverage of a feature as source?		
M1.1.1 - SCouCovS	Source Coupling Coverage Size	Number of different external features on which a feature depends
M1.1.2 - SCouCovI	Source Coupling Coverage Index	Percentage of different external features on which a feature depends
Q1.2 - What is the coupling coverage of a feature as target?		
M1.2.1 - TCouCovS	Target Coupling Coverage Size	Number of different external features depending on a feature
M1.2.2 - TCouCovI	Target Coupling Coverage Index	Percentage of different external features depending on a feature
Q1.3 - What is the coupling clustering of a feature as source?		
M1.3.1 - SCouCluS	Source Coupling Clustering Size	Number of different disjoint clusters containing features on which a feature depends
M1.3.2 - SCouCluI	Source Coupling Clustering Index	Percentage of different disjoint clusters containing features on which a feature depends
Q1.4 - What is the coupling clustering of a feature as target?		
M1.4.1 - TCouCluS	Target Coupling Clustering Size	Number of different disjoint clusters with inner features depending on a feature
M1.4.2 - TCouCluI	Target Coupling Clustering Index	Percentage of different disjoint clusters with inner features depending on a feature
Q1.5 - What is the coupling density of a feature as source?		
M1.5.1 - SCouDenS	Source Coupling Density Size	Number of different dependencies from a feature and its internals towards its external features
M1.5.2 - SCouWDenS	Source Coupling Weigted Density Size	Weighed number of different dependencies from a feature and its internals towards its external features
Q1.6 - What is the coupling density of a feature as target?		
M1.6.1 - TCouDenS	Target Coupling Density Size	Number of different dependencies from the external features towards a feature and its internal features
M1.6.2 - TCouWDenS	Target Coupling Weighed Density Size	Weighed number of different dependencies from the external features towards a feature and its internal features

Table 4.2: GQM model for coupling evaluation for *features*

G2 - Goal: Decrease modular features' coupling		
Purpose	Decrease	
Issue	Coupling	
Object	Modular Features	
Viewpoint	Software Engineer	
Questions and Metrics		
Q2.1 - What is the coupling commitment of a modular feature as source?		
M2.1.1 - SCouComS	Source Coupling Commitment Size	Number of different internal features depending on external features
M2.1.2 - SCouComI	Source Coupling Commitment Index	Percentage of different internal features depending on external features
Q2.2 - What is the coupling commitment of a modular feature as target?		
M2.2.1 - TCouComS	Target Coupling Commitment Size	Number of different internal features on which external features depend
M2.2.2 - TCouComI	Target Coupling Commitment Index	Percentage of different internal features on which external features depend

Table 4.3: GQM model for coupling evaluation for *modular features*

G3 - Goal: Increase modular features' cohesion		
Purpose	Increase	
Issue	Cohesion	
Object	Modular Features	
Viewpoint	Software Engineer	
Questions and Metrics		
Q3.1 - What is the cohesion commitment of a modular feature as source?		
M3.1.1 - SCohComS	Source Cohesion Commitment Size	Number of different internal features depending on any internal feature
M3.1.2 - SCohComI	Source Cohesion Commitment Index	Percentage of different internal features depending on any internal feature
Q3.2 - What is the cohesion commitment of a modular feature as target?		
M3.2.1 - TCohComS	Target Cohesion Commitment Size	Number of different internal features on which any internal feature depends
M3.2.2 - TCohComI	Target Cohesion Commitment Index	Percentage of different internal features on which any internal feature depends
Q3.3 - What is the cohesion density of a modular feature as source?		
M3.3.1 - SCohDenS	Source Cohesion Density Size	Number of different dependencies from a feature and its internals towards its internal features
M3.3.2 - SCohWDenS	Source Cohesion Weiged Density Size	Weighed number of different dependencies from a feature and its internals towards its internal features
Q3.4 - What is the cohesion density of a modular feature as target?		
M3.4.1 - TCohDenS	Target Cohesion Density Size	Number of different dependencies from the external features towards a feature and its internal features
M3.4.2 - TCohWDenS	Target Cohesion Weighed Density Size	Weighed number of different dependencies from the external features towards a feature and its internal features

Table 4.4: GQM model for cohesion evaluation for *modular features*

To summarize the, so far, defined metrics, figure 4.1 presents them in a different perspective, according to the modularity factor they measure, as well as the feature types and roles to which they apply.

Figure 4.1 evidences that other metrics could have been defined, namely weighed size and weighed index metrics. This is in fact true however, since that the dependency weights for the dependencies have not yet been defined, they would be fruitless and, as such, their study, definition, formalization and validation have been postponed for future work.

		Coupling				Cohesion	
		Modular or Atomic Features			Modular Features only	Modular Features only	
		Coverage	Clustering	Density	Commitment	Density	Commitment
Source	Size	SCouCovS	SCouCluS	SCouDenS	SCouComS	SCohDenS	SCohComS
	Weighed size	-	-	SCouWDenS	-	SCohWDenS	
	Index	SCouCovI	SCouCluI	-	SCouComI	-	SCohComI
Target	Size	TCouCovS	TCouCluS	TCouDenS	TCouComS	TCohDenS	TCohComS
	Weighed size	-	-	TCouWDenS	-	TCohWDenS	
	Index	TCouCovI	TCouCluI	-	TCouComI	-	TCohComI

Figure 4.1: Summary of the metrics for features

The PIMETRICS defined so far provide us an evaluation of coupling and cohesion from the feature perspective, while the fourth goal implies an evaluation of modularity from the system perspective. To achieve this purpose we will use three modularity metrics proposed by Abreu in [AG01], the Average Module Membership (AMM), the Intra-Modular Coupling Density (ICD) and the Modularity Merit Factor (MMF), and define two new metrics for crosscutting, the Potential Crosscutting Size (PCS) and the Potential Crosscutting Index (PCI), from which modularity will be inferred.

The MMF is an indicator of how balanced coupling and cohesion are at the modules of a a system. To achieve good modularity, we should strive for high cohesion and low coupling, so the better these two are balanced, the better is modularity. The higher the value of MMF the better is the modularity of a system. The AMM and ICD are primitives which contribute to the definition and formalization of the MMF.

As introduced in section 1.5, code tangling and code scattering are the facets of the concerns that crosscut a system diminishing its modularity. We can not tell yet, for sure, which of the features from a system that originate coupling belong to crosscutting concerns; however, we can tell that all of them are potential candidates. Thus, we can tell that if we implement the same system in a way to remove the code scattering

provoked by those crosscutting concerns, we will unavoidably diminish these potential candidates. In other words, for functional equivalent systems, a reduction of the potential candidates for crosscutting, means a reduction in the crosscutting itself, since at a functional equivalent system, the crosscutting concerns are the same. Since that a reduction in crosscutting means an increase in modularity, a reduction in the potential candidates for crosscutting also means an increase in modularity.

The Potential Crosscutting Size (PCS) and the Potential Crosscutting Index (PCI) will be defined and formalized together with the remaining metrics. While the first counts the potential candidates for crosscutting, the latter provides the same information in a dimensionless way, that is, contextualized within the system dimensions,as a ratio. Within two functional equivalent systems, PCs and PCI are reverse indicators of modularity. When they rise, modularity falls, and the opposite is also true.

G4 - Goal: Increase a system's modularity		
Purpose	Increase	
Issue	Modularity	
Object	System	
Viewpoint	Software Engineer	
Questions and Metrics		
Q4.1 - What is the potential crosscutting?		
M4.1.1 - PCS	Potential Crosscutting Size	Total of different dependent features in a system
M4.1.2 - PCI	Potential Crosscutting Index	Ratio between the dependent features and all the features from a system
Q4.2 - What is the modularity merit factor?		
M4.2.1 - AMM	Average Module Membership	Ratio between the total of features in a system, except the total of modular features with the required grain[1], and the total of modular features with the required grain
M4.2.2 - ICD	Intramodular Coupling Density	Ratio between the total of internal dependencies from all the modular features with the required grain, and the total dependencies in a system
M4.2.3 - MMF	Modularity Merit Factor	Ratio between ICD and AMM

Table 4.5: GQM model for system modularity evaluation

4.3 Formalizations

In this section the previously defined metrics will be formalized. As stated before, this is a very important step towards its verification, validation and eventual adoption.

The PIMETRICS will be formalized with OCL upon the PIMETA. This formalization required the development of finer grained functions in OCL which, for reasons of understandability, are defined in appendix C.

For this dissertation, the dependency weight attributes, for every weighed metric, have been set to one, as explained in section 3.2.6, so all the results between for the unweighed and its corresponding weighed metric will be the same.

The formalization that follows, is presented from table4.6 to table 4.12, according to the following criteria:

i) separate metrics applicable only to modular features from those applicable to any feature or the whole system in general;

ii) separate coupling metrics from those of cohesion and modularity.

Name	**SCouCovS - Source Coupling Coverage Size**
Informal definition	Number of different external features on which a feature depends
Formal definition	*PIMETA::Feature* `SCouCovS(): Integer = externalDependeeFeatures()->size()`
Name	**SCouCovI - Source Coupling Coverage Index**
Informal definition	Percentage of different external features on which a feature depends
Formal definition	*PIMETA::Feature* `SCouCovI(): Real = if self.externalFeatures()->isEmpty() then 0.0 else SCouCovS() / self.externalFeatures()->size() endif`
Name	**TCouCovS - Target Coupling Coverage Size**
Informal definition	Number of different external features depending on a feature
Formal definition	*PIMETA::Feature* `TCouCovS(): Integer = externalDependentFeatures()->size()`
Name	**TCouCovI - Target Coupling Coverage Index**
Informal definition	Percentage of different external features depending on a feature
Formal definition	*PIMETA::Feature* `TCouCovI(): Real = if self.externalFeatures()->isEmpty() then 0.0 else TCouCovS() / self.externalFeatures()->size() endif`

Table 4.6: Metrics formalization for coupling evaluation for features

Name	**SCouCluS - Source Coupling Clustering Size**
Informal definition	Number of different disjoint clusters on which a feature depends
Formal definition	*PIMETA::Feature* `SCouCluS():Integer = disjointExternalDependeeFeatures(). rootFeature()->asSet()->size()`
Name	**SCouCluI - Source Coupling Clustering Index**
Informal definition	Percentage of different disjoint clusters on which a feature depends
Formal definition	*PIMETA::Feature* `SCouCluI(): Real = if self.featuresWithoutCommonRoot()-> isEmpty() then 0.0 else SCouCluS() / self.featuresWithoutCommonRoot(). rootFeature()->asSet()->size() endif`
Name	**TCouCluS - Target Coupling Clustering Size**
Informal definition	Number of different disjoint clusters depending on a feature
Formal definition	*PIMETA::Feature* `TCouCluS():Integer = disjointExternalDependentFeatures().rootFeature()-> asSet()->size()`
Name	**TCouCluI - Target Coupling Clustering Index**
Informal definition	Percentage of different disjoint clusters depending on a feature
Formal definition	*PIMETA::Feature* `TCouCluI(): Real = if self.featuresWithoutCommonRoot()->isEmpty() then 0.0 else TCouCluS() / self.featuresWithoutCommonRoot().rootFeature() ->asSet()->size() endif`

Table 4.7: Metrics formalization for coupling evaluation for features (cont.)

Name	**SCouDenS - Source Coupling Density Size**
Informal definition	Number of different dependencies from a feature and its internals towards its external features
Formal definition	*PIMETA::Feature* `SCouDenS(): Integer = outboundDependencies()->collect(counter())->sum`
Name	**SCouWDenS - Source Coupling weighed Density Size**
Informal definition	Weighed number of different dependencies from a feature and its internals towards its external features
Formal definition	*PIMETA::Feature* `SCouWDenS(): Real = outboundDependencies()->collect(weighedCounter())->sum`
Name	**TCouDenS - Target Coupling Density Size**
Informal definition	Number of different dependencies from the external features towards a feature and its internal features
Formal definition	*PIMETA::Feature* `TCouDenS(): Integer = inboundDependencies()->collect(counter())->sum`
Name	**TCouWDenS - Target Coupling weighed Density Size**
Informal definition	Weighed number of different dependencies from the external features towards a feature and its internal features
Formal definition	*PIMETA::Feature* `TCouWDenS(): Real = inboundDependencies()->collect(weighedCounter())->sum`

Table 4.8: Metrics formalization for coupling evaluation for features (cont.)

Name	**SCouComS - Source Coupling Commitment Size**
Informal definition	Number of different internal features depending on external features

Formal definition	*PIMETA::ModularFeature*
	`SCouComS(): Integer =`
	`externalDependeeFeatures().dependencyDestinations.`
	`origin->asSet->`
	`intersection(self.allEnclosedFeaturesRecursively())->`
	`size()`

Name	**SCouComI - Source Coupling Commitment Index**
Informal definition	Percentage of different internal features depending on external features

Formal definition	*PIMETA::ModularFeature*
	`SCouComI(): Real =`
	`if self.internalFeatures()->isEmpty()`
	`then 0.0 else`
	`SCouComS() / self.internalFeatures()->size()`
	`endif`

Name	**TCouComS - Target Coupling Commitment Size**
Informal definition	Number of different internal features on which external features depend

Formal definition	*PIMETA::ModularFeature*
	`TCouComS(): Integer =`
	`self.externalDependentFeatures().dependencyOrigins.`
	`destination->asSet->`
	`intersection(self.allEnclosedFeaturesRecursively())`
	`->size()`

Name	**TCouComI - Target Coupling Commitment Index**
Informal definition	Percentage of different internal features on which external features depend

Formal definition	*PIMETA::ModularFeature*
	`TCouComI(): Real =`
	`if self.internalFeatures()->isEmpty()`
	`then 0.0 else`
	`TCouComS() / self.internalFeatures()->size()`
	`endif`

Table 4.9: Metrics formalization for coupling evaluation for modular features

Name	**SCohComS - Source Cohesion Commitment Size**
Informal definition	Number of different internal features depending on any internal feature
Formal definition	*PIMETA::ModularFeature* `SCohComS(): Integer = RSCohComS(self)`
Name	**SCohComI - Source Cohesion Commitment Index**
Informal definition	Percentage of different internal features depending on any internal feature
Formal definition	*PIMETA::ModularFeature* `SCohComI(): Real = RSCohComI(self)`
Name	**TCohComS - Target Cohesion Commitment Size**
Informal definition	Number of different internal features on which any internal feature depends
Formal definition	*PIMETA::ModularFeature* `TCohComS(): Integer = RTCohComS(self)`
Name	**TCohComI - Target Cohesion Commitment Index**
Informal definition	Percentage of different internal features on which any internal feature depends
Formal definition	*PIMETA::ModularFeature* `TCohComI(): Real = RTCohComI(self)`

Table 4.10: Metrics formalization for cohesion evaluation for modular features

Name	**SCohDenS - Source Cohesion Density Size**
Informal definition	Number of different dependencies from a feature and its internals towards its internal features
Formal definition	*PIMETA::ModularFeature* `SCohDenS(): Integer = (allDependencies()-outboundDependencies())->collect(counter())->sum`
Name	**SCohWDenS - Source Cohesion Weighed Density Size**
Informal definition	Weighed number of different dependencies from a feature and its internals towards its internal features
Formal definition	*PIMETA::ModularFeature* `SCohWDenS(): Real = (allDependencies()-outboundDependencies())->collect(weighedCounter())->sum`
Name	**TCohDenS - Target Cohesion Density Size**
Informal definition	Number of different dependencies from the external features towards a feature and its internal features
Formal definition	*PIMETA::ModularFeature* `TCohDenS(): Integer = (allDependencies()-inboundDependencies()-outboundDependencies()) ->collect(counter())->sum`
Name	**TCohWDenS - Target Cohesion Weighed Density Size**
Informal definition	Weighed number of different dependencies from the external features towards a feature and its internal features
Formal definition	*PIMETA::ModularFeature* `TCohWDenS(): Real = (allDependencies()-inboundDependencies()-outboundDependencies())->collect(weighedCounter())->sum`

Table 4.11: Metrics formalization for cohesion evaluation for modular features (cont.)

Name	**PCS - Potential Crosscutting Size**
Informal definition	Total of different dependent features in a system
Formal definition	*PIMETA::ModularFeature* `PCS():Integer = AOPmodules().TCouCovS()->sum()`
Name	**PCI - Potential Crosscutting Index**
Informal definition	Ratio between the dependent features and all the features from a system
Formal definition	*PIMETA::ModularFeature* `PCI():Real = self.PCS()/Feature.allInstances()->sum()`
Name	**AMM - Average Module Membership**
Informal definition	Ratio between the total of features in a system, except the total of modular features with the required grain, and the total of modular features with the required grain[2]
Formal definition	*PIMETA::ModularFeature* `AMM(): Real = (Feature.allInstances-AOPmodules())->size() / AOPmodules()->size()`
Name	**ICD - Intramodular Coupling Density**
Informal definition	Ratio between the total of internal dependencies from all the modular features with the required grain, and the total dependencies in a system
Formal definition	*PIMETA::ModularFeature* `ICD(): Real = ModularFeature.allInstances.internalDependencies()->collect(counter())->sum / ModularFeature.allInstances.internalDependencies()->union(ModularFeature.allInstances.outboundDependencies())->collect(counter())->sum`
Name	**MMF - Modularity Merit Factor**
Informal definition	Ratio between ICD and AMM
Formal definition	*PIMETA::ModularFeature* `MMF(): Real = ICD() / AMM()`

Table 4.12: Metrics formalization for *Modularity* evaluation for systems

4.4 Metrics collection

The collection method plays an important role in the whole process of measurement. When this is done automatically, the process is less prone for errors, and even though some may occur, they are easily traceable, thus increasing the accuracy of the whole process. Another important advantage of having an automatic process for metrics collection is that once set up, can be performed quickly, which can be of great advantage when dealing with medium or large scale systems. Besides, while tuning the process, this can be quickly corrected and repeated whenever deemed necessary, without further effort.

An automatized collection process has been developed within the context of this dissertation, as depicted in figure 4.2 having the observer design pattern as an example. Between the source code and the metrics' results, five steps take place which main artifacts are sorted top down at the left side of figure 4.2. First the source code is parsed into XML(.ajfml files) by using a parser for AspectJ and Java, the AspectJ-front parser [Bra08]. The second step is transforming the output from the AspectJ-Front parser into XML files (.pimetaml files) compliant with the PIMETA XML schema, using a XSLT file(AJFML2PIMETAML). From here, another transformation occurs for generating input(.cmd files), with another XSLT file(PIMETAML2USE), for the USE tool [RG], which will be used to instantiate the PIMETA. An excerpt of one of these files is presented at the right side of figure4.2. Finally, the metrics are ran(step 4) against the previously loaded USE tool(step 5), making the results available for the analysis.

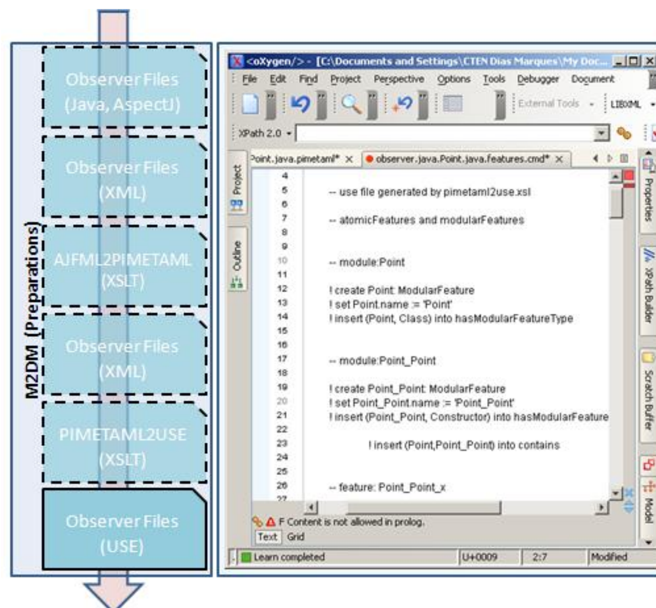

Figure 4.2: Metrics collection process

The dimension[3] of the GoF implementation in Java and AspectJ difficults a systematic manual validation of all the transformation results for each class, aspect or inter-

[3] More than 275 classes, interfaces and aspects.

face. However, along the elaboration of this dissertation randomly selected classes, interfaces and aspects were validated manually.

4.5 Exemplification

Now that the PIMETRICS have been defined and formalized, and the collection process explained, their usage will be exemplified. To achieve this purpose, the *Screen* class from the *Observer* pattern implementations in Java and AspectJ will be used. Their graphical representations and source code were already presented and can be found at figures 2.9 and 2.8 for the Java implementation, and at figures 2.11 and 2.10 for the AspectJ implementation. Besides having already been presented in a previous chapter, thus decreasing the effort to understand them and allowing a better focus at the purpose of this example, these classes are small enough to allow manual measurement and verification. All the metrics values presented here were extracted by the process described in section 4.4.

The metrics results will now be depicted and analyzed in graphs, from figure 4.3 to figure 4.12, according to the following criteria:

i) understandability;

ii) different graphs for different scales (absolute and ratio);

iii) directed mapping to the goals (defined at section 4.2).

For the first goal(G1), *to decrease feature's coupling*, the corresponding metrics' results are depicted at figures 4.3, 4.4 and 4.5.

It is expectable that with AOP, the coupling as source of dependencies for the core[4] Java classes to decrease in their aspectized versions. the clustering metrics are as expected , because there is only one cluster.

For the second goal(G2), *to decrease modular features' coupling*, the corresponding metrics' results are depicted at figures 4.6 and 4.7.

The absolute and ratio metrics are coherent with the expected results. Notice that for TCouComS and TCouComI, the result for both languages is zero. Strange would be if in AspectJ these values would have increased.

As for the third goal(G3), *to increase modular features' cohesion*, the corresponding metrics' results are depicted at figures 4.8, 4.9 and 4.10. The absolute and ratio metrics are also coherent with the expected results. Notice that TCouComS and TCouComI also have the values zero like TCouDenS and TCouWDenS and, like these, if this value would have increased with AspectJ it would not be normal and worth of a deeper investigation.

[4]Java classes that implement the core functional concerns, from which the screen class is an example

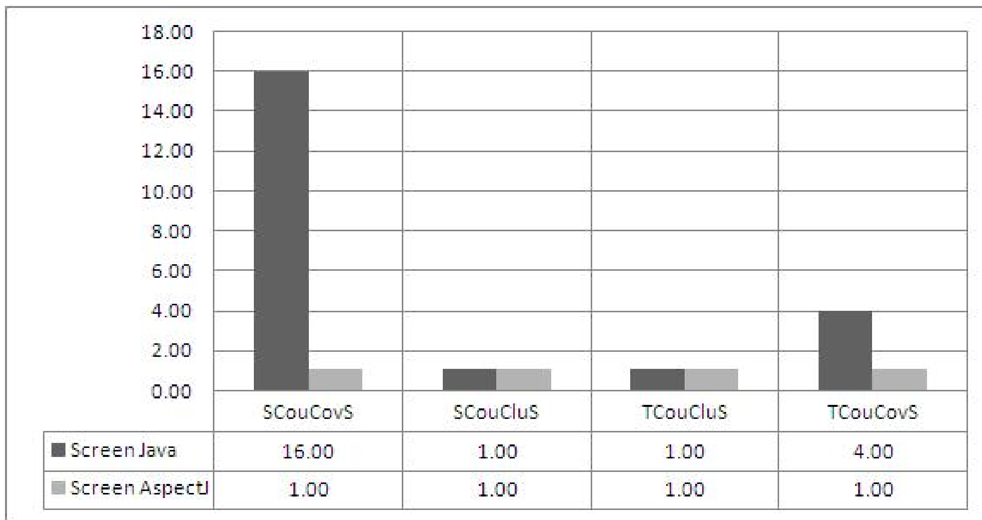

Figure 4.3: Results of the absolute metrics from goal G1

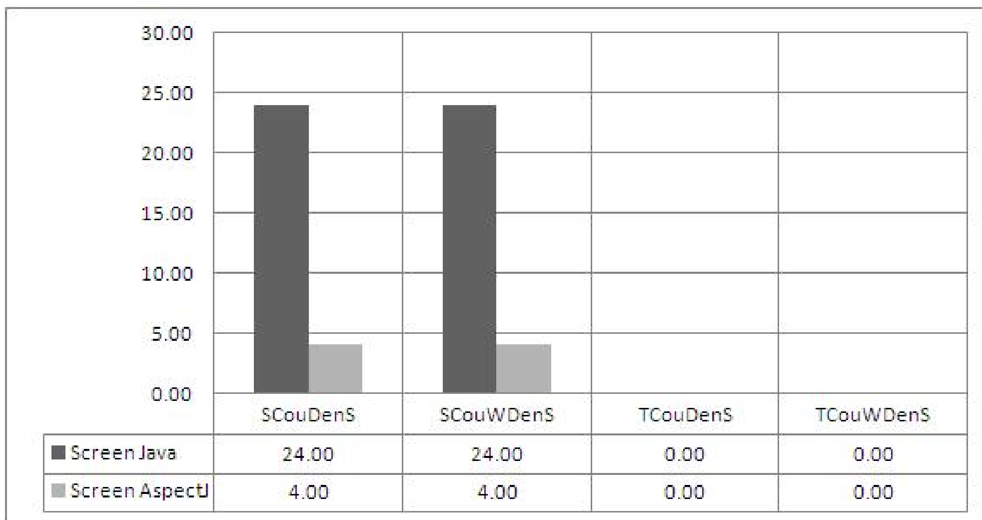

Figure 4.4: Results of the absolute metrics from goal G1 (cont.)

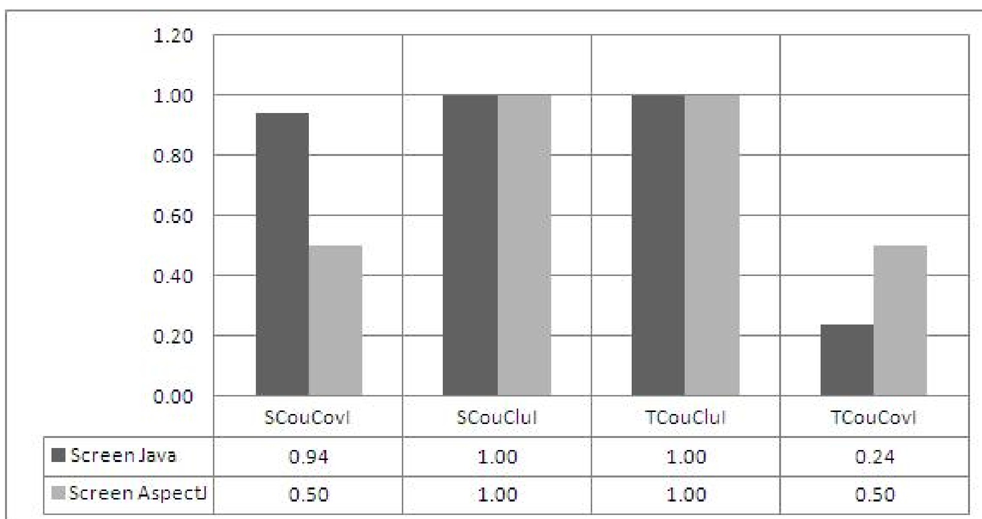

Figure 4.5: Results of the ratio metrics from goal G1

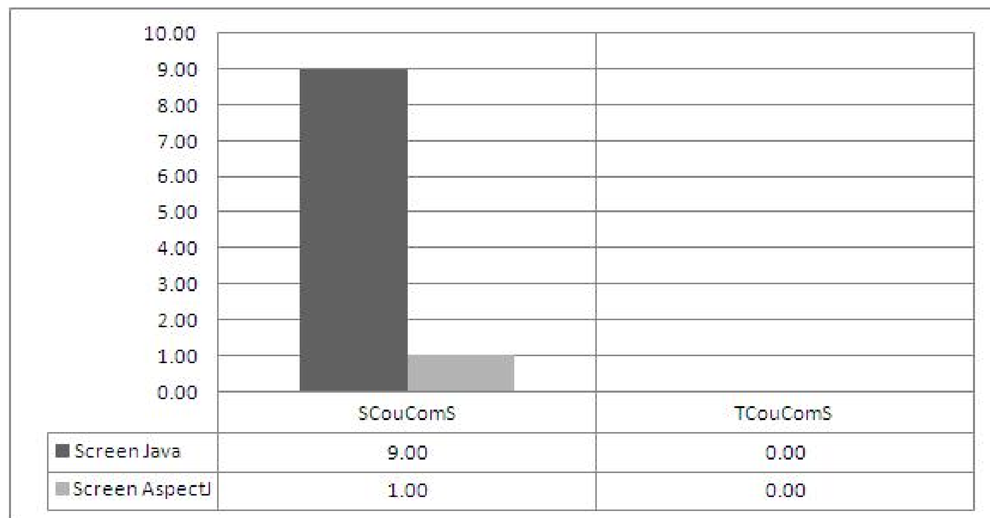

Figure 4.6: Results of the absolute metrics from goal G2

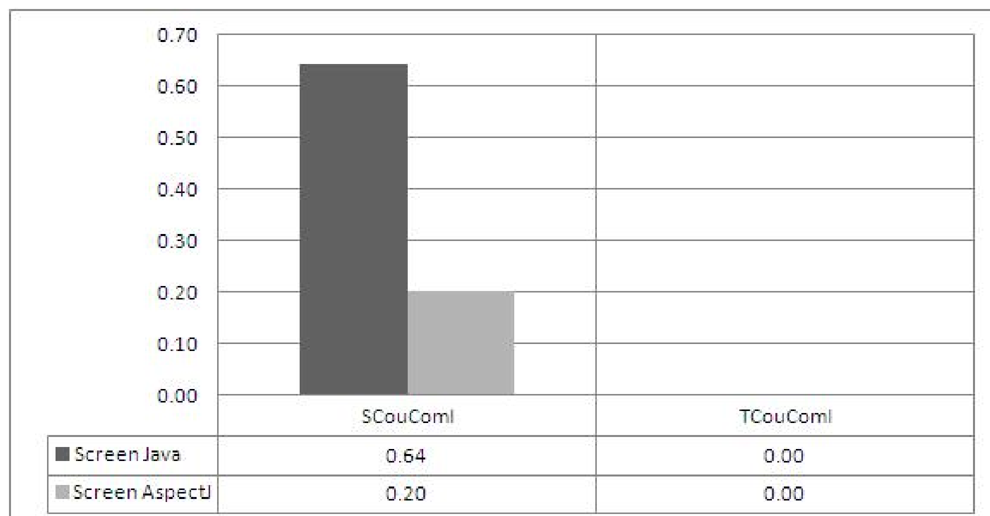

Figure 4.7: Results of the ratio metrics from goal G2

It may seem surprising that as cohesion is increasing the metrics' values are decreasing. However, the explanation for this is that with cross-cutting concerns, when code tangling occurs, the *native* code[5] highly depends on the *alien* code[6].

It is worth noticing that cohesion not only has to do with the relative interdependence among the features from a module, but this interdependence must adhere to a commom purpose or concern. Therefore, these interdependences between the core and the alien code, should not be considered cohesion. As such, when code tangling occurs, we believe to be in the presence of *delusional* cohesion, that is, the cohesion values increase, but due to an inadequate implementation of dependencies.

It is worth to mention that if these metrics are sensible to delusional cohesion, they probably may be used for identifying candidate aspects, together with the remainder metrics. Even though being out of the scope of this dissertation, this subject will most certainly be included in future work.

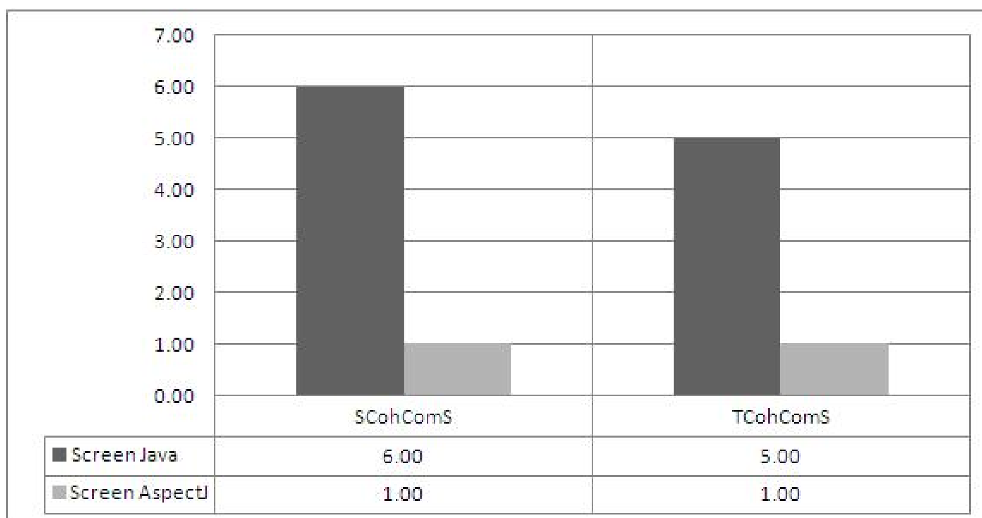

Figure 4.8: Results of the absolute metrics from goal G3

Finally, for the last goal(G4), *to increase a system's modularity*, the corresponding metrics' results are depicted at figures 4.11 and 4.12. The absolute and ratio metrics are also coherent with the expected results.

It is worth noticing that crosscutting has decreased and modularity has increased, if considered the PCI and MMF metrics, respectively.

To summarize the above described results, we can say that, for the Screen class:

i) most of the metrics for coupling have improved. Some have remained the same and only the results of TCouCovI show that AspectJ worsened the class;

ii) all of the metrics for cohesion confirm that cohesion is worse at the aspectj version of the screen class. We explain that with the *delusional cohesion* originated by the code tangling;

[5]Code that implements the core concern of the class
[6]Code that implements secondary concerns, from the class perspective

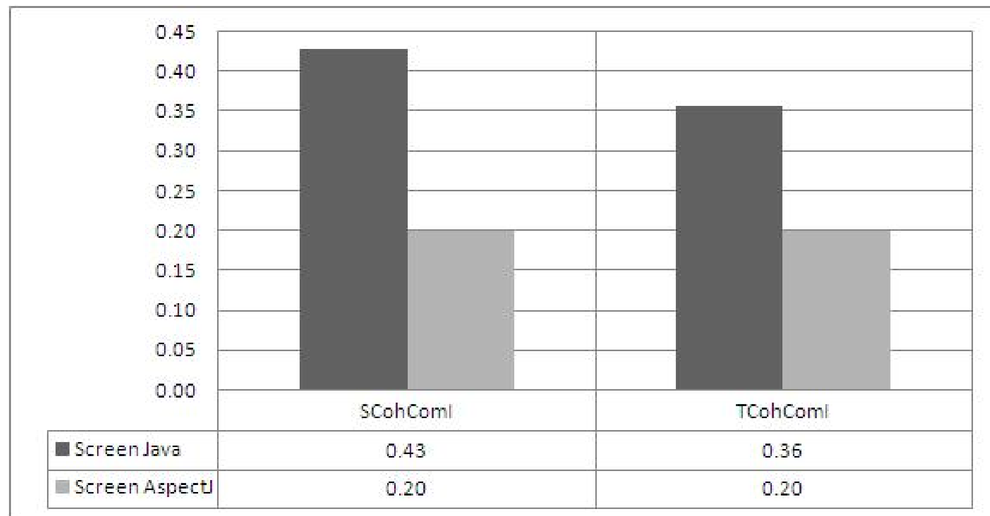

Figure 4.9: Results of the ratio metrics from goal G3

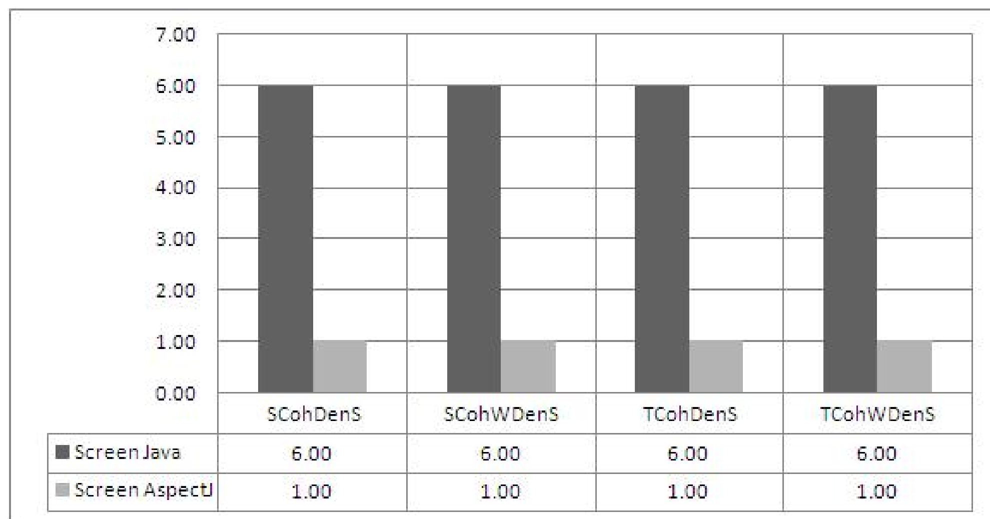

Figure 4.10: Results of the absolute metrics from goal G3 (cont.)

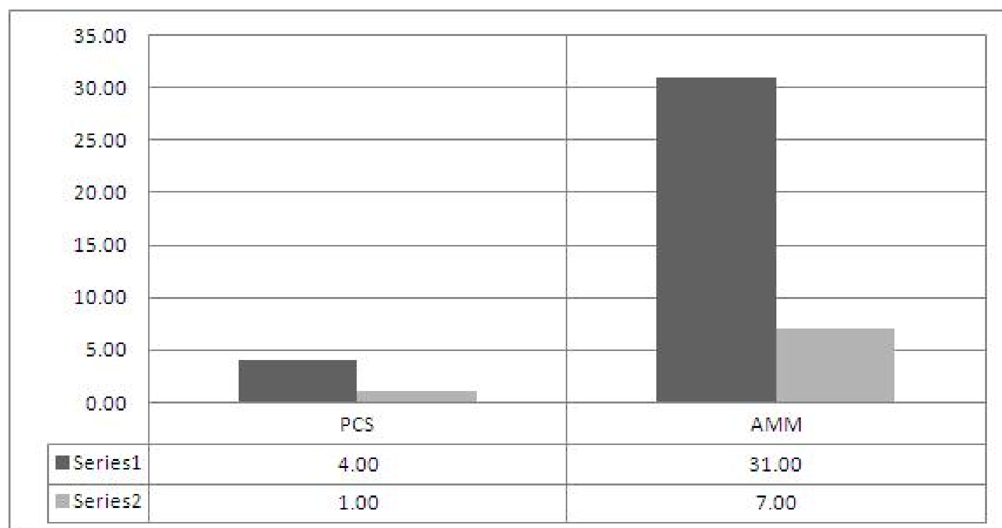

Figure 4.11: Results of the absolute metrics from goal G4

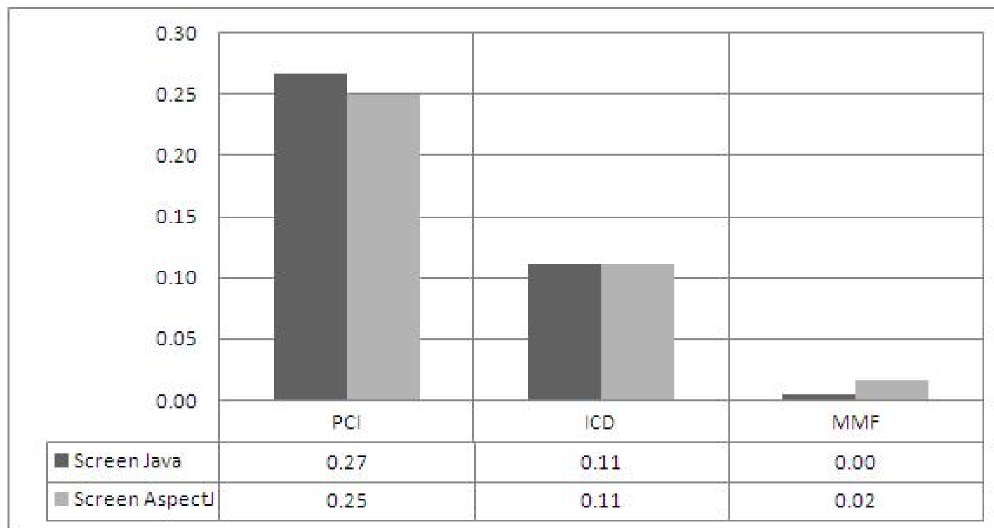

Figure 4.12: Results of the ratio metrics from goal G4

iii) the modularity of the class has slightly improved, according to the modularity metrics.

4.6 Validation

Several publications address this subject, the validation of software metrics, being those from Fenton [FP97], Henderson-Sellers [HS96] and Briand [BEEM95] unavoidable.

Even though each of them encompassing several methods to achieve this purpose, they agree on two important requirements for considering a metric valid:

i) a metric must undoubtedly measure the attribute it was defined for;

ii) to be useful, a metric should be related to external software properties;

Since that several metrics have been defined and formalized in the previous sections, it seems appropriate to, at this moment, define their purpose.

The goal of this dissertation is to evaluate modularity improvements obtained with AOP, which will be accomplished by using the PIMETRICS to compare modularity at functional equivalent systems implemented both in AOP and OOP. Therefore, to achieve this goal, only internal software properties must be assessed, which means that establishing relations between the PIMETRICS and other external software properties, like maintainability, for instance, are not required, thus being out of the scope of this dissertation.

As such, the PIMETRICS will not be *useful* as previously defined. They will only provide the means to perform the intended evaluation.

Of course, by not fulfilling the second requirement, the PIMETRICS can not aspire to be considered valid in *latu sense*, but only *internally* or *theoretically* valid.

The requirements for a theoretical validation are [BEEM95]:

i) the measure must be consistent with the intuitive understanding of the concept;

ii) a model of the intuitive understanding of the attribute under measurement must be developed;

Chapter 3 presented a model of the intuitive understanding about modularity with the PIFACTORS, while section 4.2 provides the alignment between Modularity, PIFACTORS and the PIMETRICS. Further, the example of the PIMETRICS usage, provided at section 4.5, with a real case, enforces the previously showed consistency between the measures and the intuitive understanding of modularity.

By fulfilling both the requirements for theoretical validation, the PIMETRICS can be considered *internally* or *theoretically* valid.

[This page was intentionally left blank]

Chapter 5

GoF Design Patterns Modularity

Contents

5.1 Introduction 60

5.2 The sample 60

5.3 Descriptive statistics 63

5.4 EDA: Exploratory Data Analysis 65

5.5 Hypotheses identification and testing 69

In this chapter the results from the measurement of the GoF design patterns in Java *and* AspectJ *with the PIMETRICS will be presented and discussed.*

5.1 Introduction

In this chapter the results from the measurement of the GoF design patterns in Java and AspectJ with the PIMETRICS will be discussed. These results will provide the grounds to answer the following research questions:

i) Does the refactoring of software systems produced with an OOP language to an AOP extension of the same language produce a significant change on the overall coupling?

ii) Does the refactoring of software systems produced with an OOP language to an AOP extension of the same language produce a significant change on the coupling of the core modular features [1]?

iii) Does the refactoring of software systems developed with an OOP language to an AOP extension of the same language produce a significant change on the overall cohesion?

iv) Does the refactoring of software systems developed with an OOP language to an AOP extension of the same language produce a significant change on the cohesion of the core modular features?

v) Does the refactoring of software systems developed with an OOP language to an AOP extension of the same language produce a significant change on the overall modularity?

To answer the previous questions, statistics will be used to validate the data obtained and support the conclusions. These will be explained as appropriate, along the following sections.

5.2 The sample

The subjects of our experiment are a set of 2003 features (either atomic or modular). The coarsest grained modular features (classes, interfaces and aspects) are distributed among the two design patterns implementations as depicted in figure 5.1 and figure 5.2. All of the 2003 features are distributed as depicted in figure 5.3 and figure 5.4. For each feature we have calculated the values for the PIMETRICS, using the M2DM approach, upon the PIMETA loaded with a set of 2039 dependencies among the previous features. The latter are distributed as depicted in figure 5.5 and figure 5.6.

Considering all the Java feature types described at appendix A and the data presented at figure 5.3, only *Package* and *Enumeration* were not found within the sample.

[1] Classes and interfaces that existed in the Java patterns and were kept in the AspectJ implementations

FeatureTypeName	Language		Total
	AspectJ	Java	
Aspect	40	0	40
Class	89	97	186
Interface	38	20	58
Total	167	117	284

Figure 5.1: Coarsest grained modular features distribution by the GoF implementations

	Java		AspectJ		
	class	interface	aspect	class	interface
AbstractFactory	3	1	1	2	1
Adapter	6	1	1	6	1
Bridge	6	1	1	5	2
Builder	4		1	3	1
ChainOfResponsibility	5	1	2	6	2
Command	4	1	2	5	3
Composite	3	1	2	3	5
Decorator	5	1	2	2	
Facade	5		1	5	
FactoryMethod	4		2	3	1
Flyweight	4	1	2	3	2
Interpreter	8	1	1	8	1
Iterator	3	1	1	3	1
Mediator	3	2	2	3	2
Memento	3		2	3	2
Observer	3	2	4	3	2
Prototype	3		2	3	1
Proxy	4	1	4	3	1
Singleton	3		1	3	1
State	5	2	1	5	1
Strategy	4	1	2	4	2
TemplateMethod	4		1	3	1
Visitor	5	2	2	5	5
Total	97	20	40	89	38

Figure 5.2: Number of features per design pattern and language for the coarsest feature types

FeatureType	Java	AspectJ	Total
advice after		9	**9**
advice around		5	**5**
aspect		40	**40**
class	97	89	**186**
constructor	42	33	**75**
declare parents		34	**34**
exception	2	1	**3**
field	72	70	**142**
inter-type operation		23	**23**
interface	20	38	**58**
local variable	120	134	**254**
method	255	234	**489**
parameter	243	286	**529**
pointcut definition		8	**8**
return value	72	76	**148**
Total	**923**	**1080**	**2003**

Figure 5.3: Number of features per *FeatureType* and *Language*

Design pattern	Java	AspectJ	Total
AbstractFactory	17	14	**31**
Adapter	63	62	**125**
Bridge	42	46	**88**
Builder	35	38	**73**
ChainOfResponsibility	35	60	**95**
Command	23	72	**95**
Composite	54	72	**126**
Decorator	28	13	**41**
Facade	24	25	**49**
FactoryMethod	26	30	**56**
Flyweight	33	46	**79**
Interpreter	105	80	**185**
Iterator	37	40	**77**
Mediator	27	41	**68**
Memento	20	35	**55**
Observer	57	73	**130**
Prototype	30	40	**70**
Proxy	41	57	**98**
Singleton	19	22	**41**
State	74	63	**137**
Strategy	35	49	**84**
TemplateMethod	39	39	**78**
Visitor	59	63	**122**
Total	**923**	**1080**	**2003**

Figure 5.4: Number of features per *Design Pattern* and *Language*

Considering all the AspectJ feature types described at appendix B and the data presented at figure 5.3, only the modular feature *AdviceBefore* was not found within the sample. As for the atomic features only *DeclareParents* was found within the sample.

		Language		
		AspectJ	Java	Total
		Count	Count	Count
DependencyType	AdviceAfterCallsMethod	16	0	16
	AdviceAfterUsesPointcut Definition	6	0	6
	AdviceAroundCallsMethod	8	0	8
	AdviceAroundUses PointcutDefinition	3	0	3
	AspectImportsClass	41	0	41
	ClassImportsClass	50	62	112
	ClassImportsPackage	6	6	12
	ClassInheritance	13	20	33
	ClassInterface Implementation	32	44	76
	ConstructorCalls Constructor	13	16	29
	ConstructorCallsMethod	10	8	18
	ConstructorUsesField	20	34	54
	ExceptionType	1	2	3
	FieldType	50	53	103
	InterfaceImportsClass	3	2	5
	InterfaceInheritance	4	0	4
	InterTypeOperationCalls Method	39	0	39
	InterTypeOperation DestinationClass	41	0	41
	LocalVariableType	121	111	232
	MethodCallsConstructor	5	13	18
	MethodCallsInterType Operation	3	0	3
	MethodCallsMethod	286	312	598
	MethodUsesField	5	15	20
	ParameterType	235	182	417
	ReturnValueType	76	72	148
	Total	1087	952	2039

Figure 5.5: Number of dependencies per *DependencyType* and *Language*

Considering all the Java dependency types described at appendix A and the data presented at figure 5.5, only a few were not found within the sample.

Considering all the AspectJ dependency types described at appendix B and the data presented at figure 5.5, only a few were found within the sample. It is worth mentioning that while the referred appendix was developed based at some of the latest AspectJ documentation [AT08], the sample under study was published in [HK02] six years ago, meaning that the features of the AspectJ version available at the time were not as many as today.

5.3 Descriptive statistics

The variables used in this experiment are self-described in figure 5.7.

As expected, all PIMETRICS indexes are within the expected range [0, 1], as described in figure 5.8. However, notice that while SCouCluI, TCouCluI, SCouComI and TCouComI take values along all the range, SCouCovI, TCouCovI, SCohComI and TCohComI do not behave the same. Another aspect that deserves our attention is that the standard deviation (a measure of dispersion around the mean, that is measured in the

		Language		
		AspectJ	Java	Total
		Count	Count	Count
Pattern	AbstractFactory	103	102	205
	Adapter	9	15	24
	Bridge	43	42	85
	Builder	29	26	55
	ChainOfResponsibility	78	54	132
	Command	81	34	115
	Composite	94	43	137
	Decorator	9	25	34
	Facade	28	27	55
	FactoryMethod	46	41	87
	Flyweight	35	28	63
	Interpreter	89	110	199
	Iterator	38	35	73
	Mediator	52	37	89
	Memento	27	11	38
	Observer	67	75	142
	Prototype	31	25	56
	Proxy	51	43	94
	Singleton	22	20	42
	State	47	61	108
	Strategy	28	16	44
	TemplateMethod	29	32	61
	Visitor	51	50	101
	Total	1087	952	2039

Figure 5.6: Number of dependencies per *Design Pattern* and *Language*

Variable	Scale	Description
Language	Nominal	Name of the programming language
Pattern	Nominal	Name of the design pattern
Feature Type	Nominal	Name of the feature type
Feature Name	Nominal	Name of the feature
PIMETRICS (size)	Absolute	PIMETRIC of size kind
PIMETRICS (indexes)	Ratio	PIMETRIC of index kind
PCS	Absolute	Potential Crosscutting Size
PCI	Ratio	Potential Crosscutting Index
AMM	Ratio	Average Module Membership
ICD	Ratio	Intramodular Coupling Density
MMF	Ratio	Modularization Merit Factor

Figure 5.7: Variables used, their scale types and description

same units as the variable itself) is considerably high for most indexes, specially for SCouCluI and SCouComI. The mentioned indexes are measures of coupling and cohesion relative to the context of the feature under measurement. The above facts tell us that for example, it is easier for a class to depend on or be depended by all of the remainder classes of the pattern to which it belongs (SCouCluI, TCouCluI) and to have all its features committed to this coupling (SCouComI, TCoucomI), than to have dependencies towards all the features of a system or the other way around (SCouCovI, TCouCovI), or even have all its features depending on each other(SCohComI, TCohComI).

	Mean	Std. Deviation	Minimum	Maximum
SCouCovI	.0653074	.07207051	.00000	.40000
SCouCluI	.2260046	.19041716	.00000	1.00000
TCouCluI	.3833822	.26842803	.00000	1.00000
TCouCovI	.0837142	.08924299	.00000	.76923
SCouComI	.2724524	.26467632	.00000	1.00000
TCouComI	.0968510	.18469321	.00000	1.00000
SCohComI	.0481582	.08397524	.00000	.42857
TCohComI	.0561044	.09486656	.00000	.37500

Figure 5.8: Descriptive statistics on the PIMETRICS (indexes)

5.4 EDA: Exploratory Data Analysis

Some of the main purposes for conducting an exploratory data analysis are:

i) identify data problems like outliers or non-normal distributions;

ii) make sure that the assumptions for using the intended statistics are gathered;

The following subsections encompass this analysis.

5.4.1 Outlier analysis

From the box-plots in figure 5.9 we can conclude that most PIMETRICS indexes have outliers (represented as empty circles) and half of them also have extreme values (represented as asterisks). Each outlier or extreme value should be investigated to assess if it is due to one of two alternatives:

i) a collection or transcription error - in this case we should remove the corresponding case from the sample (or correct it);

ii) a source of natural variability - here we should identify that source and keep the case in the sample.

Although the first alternative occurs more frequently, that was not the case here. We have investigated manually most outliers and extremes and found out that this phenomenon was indeed due to options made by the developers while implementing the patterns.

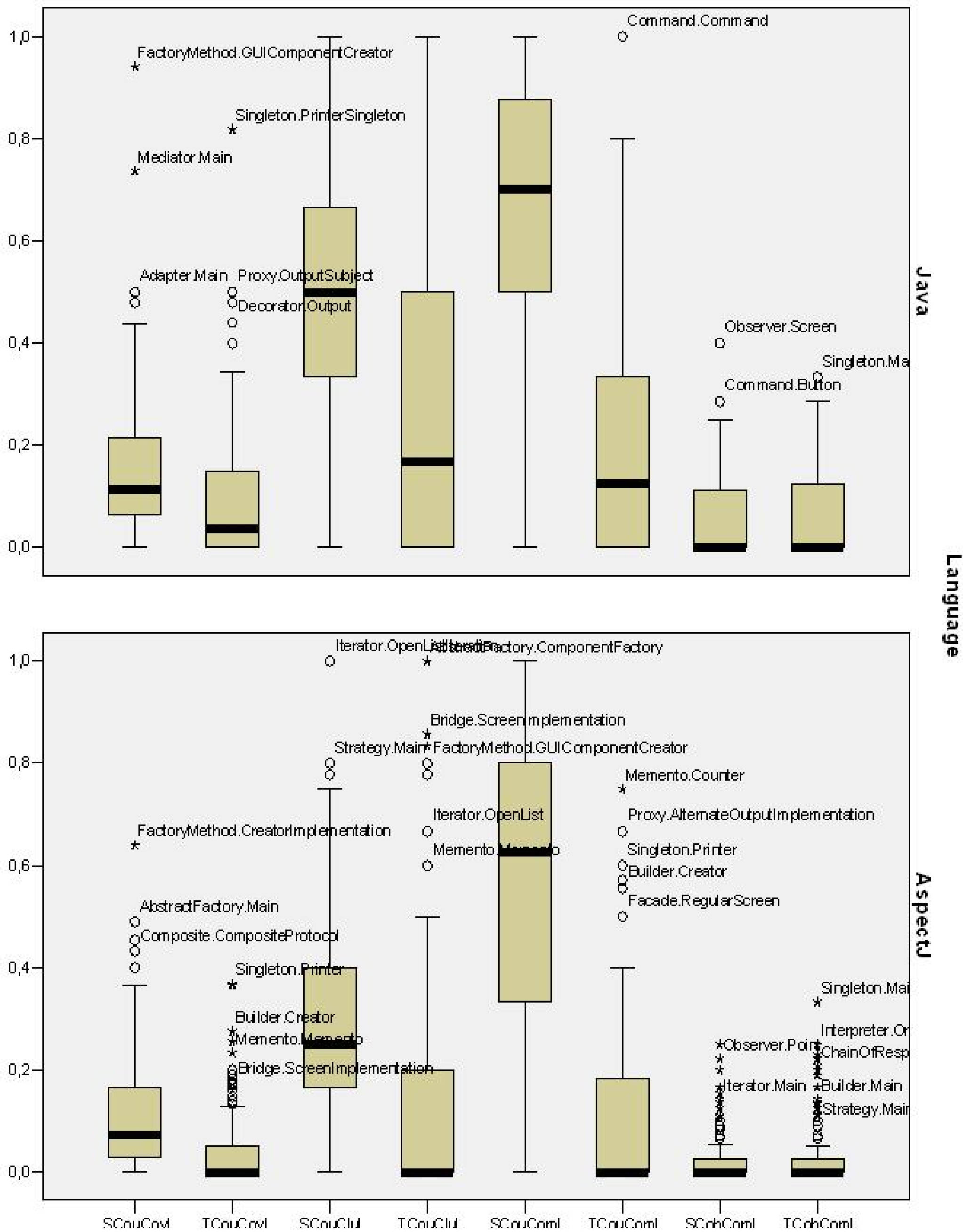

Figure 5.9: Outlier analysis for the PIMETRICS (indexes)

5.4.2 Testing distribution adherence

To assess if we can apply parametric tests in the evaluation of our hypothesis, we need to test if the PIMETRICS values in our sample match a Normal distribution. In figure 5.10 and figure 5.11 we reproduce the detrended standardized QQ plots. The latter plot the standardized deviation of the observed values from a Normal distribution. To be Normal, a given variable should have its points near the horizontal line corresponding to a null deviation. None of the metrics being analyzed exhibits such a behavior, so most probably they do not have a Normal distribution.

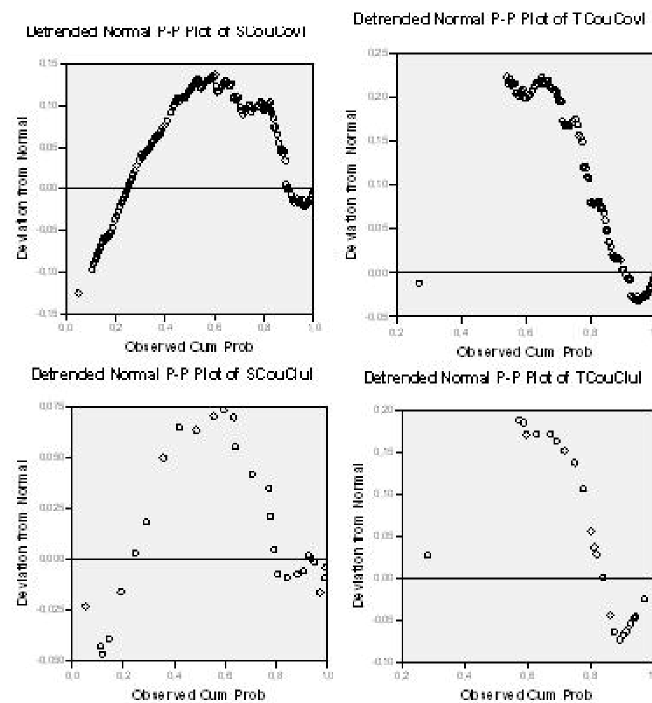

Figure 5.10: QQ Plots for SCouCovI, TCouCovI, SCouCluI and TCouCluI

To test the hypothesis of normality we have applied the Kolmogorov-Smirnov one-sample test, which is based on the maximum difference between the sample cumulative distribution and the hypothesized cumulative distribution. The underlying hypotheses for this test are the following:

$$H0 : X \sim N(\mu;\sigma); H1 : \neg X \sim N(\mu;\sigma)$$

Considering a test significance $\alpha = 0.05$ (probability of Type I error of 5 %) we can conclude, from figure 5.12 and figure 5.13, that we must reject the null hypothesis for all variables, since we get a significance $p \leq 0.05$, which means that we have significant Z statistics for all variables being analyzed. In other words, *we cannot sustain that the considered variables of our sample come from a Normal population*.

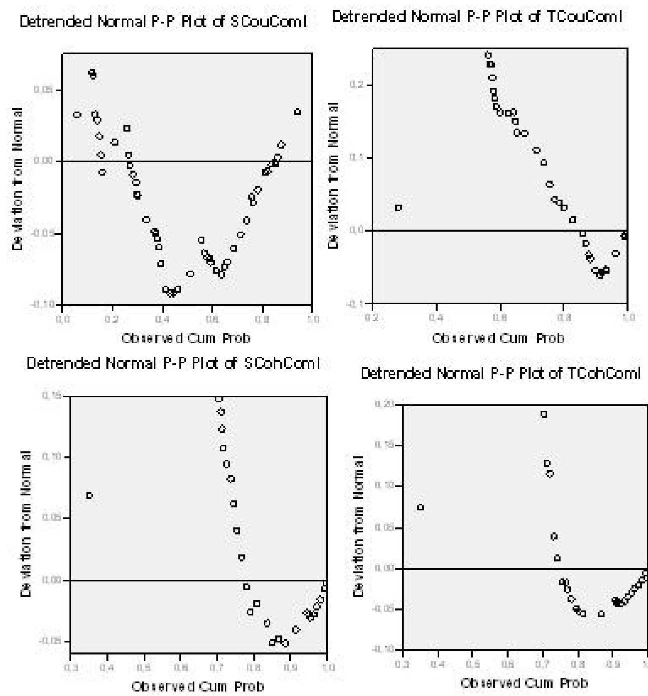

Figure 5.11: QQ Plots for SCouComI, TCoucomI, SCohComI and TCohComI

	SCou CovI	TCou CovI	SCou ComI	TCou ComI	SCou CluI	TCou CluI	SCoh ComI	TCoh ComI
Kolmogorov-Smirnov Z	2.937	4.738	2.033	5.254	2.078	5.175	7.064	7.158
Asymp. Sig. (2-tailed)	.000	.000	.001	.000	.000	.000	.000	.000

Figure 5.12: Testing Normal distribution adherence with the Kolmogorov-Smirnov test for the PIMETRICS

	PCS	PCI	AMM	ICD	MMF
Kolmogorov-Smirnov Z	3.643	2.437	1.520	1.655	1.927
Asymp. Sig. (2-tailed)	.000	.000	.020	.008	.001

Figure 5.13: Testing Normal distribution adherence with the Kolmogorov-Smirnov test for the remaining metrics

5.5 Hypotheses identification and testing

In this section we identify and justify the research questions statistical hypotheses that must be tested in order to answer the previously stated research questions. We then apply the adequate statistical tests and interpret their results.

From now on, for the sake of comparability, we will only consider the coarsest grained features in both the Java and AspectJ implementations, which are the following modular features: classes, interfaces and aspects, which are described in figure 5.1.

5.5.1 Metrics discriminative power

Research question: Do the proposed metrics for coupling, cohesion and modularity vary significantly from pattern to pattern within the same implementation language?

Justification: While it can easily be observed that the 23 GoF pattern implementations, either on Java or AspectJ, differ significantly in size (see figure 5.4), the same is not expected for size-independent metrics like the PIMETRICS ones. However, according to what was said in chapter 4, we expect that those metrics have some discriminative power. In other words, they are expected to show some kind of variance, depending on the considered design pattern.

Statistical hypothesis: We want to know if each of the coupling, cohesion and modularity metrics, taken from the independent samples corresponding to features of each of the design patterns may be considered as drawn from the same population. In other words, we want to see if the values of those metrics vary significantly from design pattern to design pattern. We will perform two separate tests, one only for the Java patterns and another for the AspectJ patterns. Due to the fact that the considered metrics are not normally distributed, as we have seen on a previous section, we can only perform a nonparametric analysis of variance. We will use the Kruskal-Wallis one-way analysis of variance, an extension of the Mann-Whitney U test, which is the nonparametric analog of the one-way ANOVA test. The Kruskal-Wallis H test allows assessing whether several independent samples are from the same population (i.e. if they have similar statistical distributions). In our case those independent samples correspond to the 23 implementations of the GoF patterns for a given language. Let M be a coupling, cohesion or modularity metric and i and j two different design patterns. Then, the underlying hypotheses for this test are the following:

$$H0 : \forall i, j : Mi \sim Mj; H1 : \forall i, j : Mi \sim Mj$$

Results interpretation: The *Kruskal-Wallis H* test statistic is distributed approximately as *chi-square*. Consulting a chi-square table with $df = 22$ (degrees of freedom) and for a significance of $\alpha = 0.05$ (probability of Type I error of 1 %) we obtain a critical value of chi-square of 12.3. Since this value is less than the computed H values (for each of the

	SCouCovI	TCouCovI	SCouCluI	TCouCluI	SCouComI	TCouComI	SCohComI	TCohComI
Chi-Square	28.465	12.186	14.638	12.474	31.521	19.732	18.553	17.629
df	22	22	22	22	22	22	22	22
Asymp. Sig.	.161	.953	.877	.947	.086	.600	.673	.728

Figure 5.14: Testing the influence of the Java design pattern on coupling and cohesion metrics with the Kruskal-Wallis one-way analysis of variance test

	SCouCovI	TCouCovI	SCouCluI	TCouCluI	SCouComI	TCouComI	SCohComI	TCohComI
Chi-Square	34.949	16.459	41.877	15.776	38.769	19.541	25.190	23.445
df	22	22	22	22	22	22	22	22
Asymp. Sig.	.039	.792	.006	.827	.015	.612	.288	.377

Figure 5.15: Testing the influence of the AspectJ design pattern on coupling and cohesion metrics with the Kruskal-Wallis one-way analysis of variance test

	PCS	PCI	AMM	ICD	MMF
Chi-Square	116.000	115.985	116.000	116.000	116.000
df	22	22	22	22	22
Asymp. Sig.	.000	.000	.000	.000	.000

Figure 5.16: Testing the influence of the Java design pattern on modularity metrics with the Kruskal-Wallis one-way analysis of variance test

	PCS	PCI	AMM	ICD	MMF
Chi-Square	166.000	165.714	166.000	166.000	166.000
df	22	22	22	22	22
Asymp. Sig.	.000	.000	.000	.000	.000

Figure 5.17: Testing the influence of the AspectJ design pattern on modularity metrics with the Kruskal-Wallis one-way analysis of variance test

metrics in figure 5.14 to figure 5.17), we reject the null hypothesis that the samples do not differ on the criterion variable (the Design Pattern). In other words, given any of the proposed metrics, we cannot sustain that the statistical distributions of the groups of modular features corresponding to each of the design patterns are the same. This means that we accept the alternative hypothesis that the design pattern has influence on the values of the proposed metrics. In other words, *the metrics have discriminative power*, as we have planned.

5.5.2 Overall coupling changes

Research question: Does the refactoring of software systems produced with an OOP language to an AOP extension of the same language produces a significant change on the overall coupling?

Justification: Regarding coupling, we expect that the AOP pattern versions are statistically different and, as claimed, exhibiting lower values for coupling from the OOP ones.

Statistical hypothesis: If we consider that we have two independent groups of cases, one for the Java language and another for the AspectJ language, we can compare the averages for each of the testing variables in the two groups. In other words we want to test if there is a significant difference between the averages of the variables describing cohesion, for the two languages.

$$H0 : \mu\ SCouCovI\ \text{Java} = \mu\ SCouCovI\ \text{AspectJ}$$
$$H1 : \mu\ SCouCovI\ \text{Java} \neq \mu\ SCouCovI\ \text{AspectJ}$$

$$H0 : \mu\ TCouCovI\ \text{Java} = \mu\ TCouCovI\ \text{AspectJ}$$
$$H1 : \mu\ TCouCovI\ \text{Java} \neq \mu\ TCouCovI\ \text{AspectJ}$$

$$H0 : \mu\ SCouCluI\ \text{Java} = \mu\ SCouCluI\ \text{AspectJ}$$
$$H1 : \mu\ SCouCluI\ \text{Java} \neq \mu\ SCouCluI\ \text{AspectJ}$$

$$H0 : \mu\ TCouCluI\ \text{Java} = \mu\ TCouCluI\ \text{AspectJ}$$
$$H1 : \mu\ TCouCluI\ \text{Java} \neq \mu\ TCouCluI\ \text{AspectJ}$$

$$H0 : \mu\ SCouComI\ \text{Java} = \mu\ SCouComI\ \text{AspectJ}$$
$$H1 : \mu\ SCouComI\ \text{Java} \neq \mu\ SCouComI\ \text{AspectJ}$$

$$H0 : \mu\ TCouComI\ \text{Java} = \mu\ TCouComI\text{AspectJ}$$
$$H1 : \mu\ TCouComI\ \text{Java} \neq \mu\ TCouComI\ \text{AspectJ}$$

The corresponding test is called the *Independent Samples T-Test*. If both groups have

sizes above 30 cases, then the distribution t with v degrees of freedom $t(v)$, approaches the *Normal* distribution. For samples below 30 cases, this test is only applicable if the testing variables are Normal, which is not the case as we have previously determined. From figure 5.1 we can then see that both groups have much more than 30 cases, so this test is applicable.

	Language	Mean	Std. Deviation	Std. Error Mean
SCouCovI	Java	.1737486	.17471922	.01615280
	AspectJ	.1086638	.11191045	.00865989
SCouCluI	Java	.5033476	.27691786	.02560107
	AspectJ	.3030415	.22679354	.01754981
TCouCluI	Java	.2868946	.36143127	.03341433
	AspectJ	.1411748	.23578489	.01824558
TCouCovI	Java	.0915600	.13574756	.01254987
	AspectJ	.0393134	.06914411	.00535053
SCouComI	Java	.6734601	.26181452	.02420476
	AspectJ	.5445686	.32528246	.02517111
TCouComI	Java	.1840133	.23350916	.02158793
	AspectJ	.1163521	.19121343	.01479654

Figure 5.18: PIMETRICS average per GoF implementation

		Levene's Test for Equality of Variances		t-test for Equality of Means						
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
									Lower	Upper
SCouCovI	Equal variances assumed	9.152	.003	3.824	282	.000	.06508479	.01701982	.03158278	.09858680
	Equal variances not assumed			3.551	181.773	.000	.06508479	.01832775	.02892229	.10124729
SCouCluI	Equal variances assumed	1.838	.176	6.682	282	.000	.20030604	.02997618	.14130057	.25931152
	Equal variances not assumed			6.453	217.132	.000	.20030604	.03103885	.13913003	.26148206
TCouCluI	Equal variances assumed	40.093	.000	4.111	282	.000	.14571979	.03545020	.07593920	.21550039
	Equal variances not assumed			3.828	184.052	.000	.14571979	.03807123	.07060765	.22083194
TCouCovI	Equal variances assumed	38.801	.000	4.251	282	.000	.05224660	.01229156	.02805176	.07644144
	Equal variances not assumed			3.830	158.347	.000	.05224660	.01364285	.02530117	.07919203
SCouComI	Equal variances assumed	8.954	.003	3.554	282	.000	.12889159	.03626494	.05750725	.20027593
	Equal variances not assumed			3.691	276.548	.000	.12889159	.03492070	.06014742	.19763575
TCouComI	Equal variances assumed	8.790	.003	2.677	282	.008	.06766113	.02527536	.01790881	.11741345
	Equal variances not assumed			2.585	217.108	.010	.06766113	.02617205	.01607730	.11924495

Figure 5.19: *t-test* applied to the PIMETRICS indexes

Results interpretation: Based upon the result of the *Levene's* test and an $\alpha = 0.05$, we can say that we have to reject the hypothesis of the equality of variances between the two groups, for all testing variables. Therefore we have to choose the rows signalized with *Equal variances not assumed*. The positive value for the t statistic shows us that the values of the coupling metrics are, on the average, superior for Java than for AspectJ. Considering a test significance $\alpha = 0.05$ we can conclude from figure 5.19, that we

must reject the null hypotheses that the averages of all coupling variables are the same between the two groups. In other words, we can say that there is statistical evidence that *the coupling for the OOP versions is, on average, higher than the coupling for AOP module versions*.

5.5.3 Core modular features coupling changes

Research question: Does the refactoring of software systems produced with an OOP language to a AOP extension of the same language produces a significant change on the coupling of the core modular features [2]?

With a rotation operation we were able to produce a paired list version of our data, so that for equivalent modular features, we get two variables within the same case, representing the values for the same variable in Java and in AspectJ, like for instance: *TCouComI-Java* and *TCouComI-AJ*. Notice that we get fewer cases than for independent tests, because only classes and interfaces are *pairable*. The hypotheses are the same as above. The corresponding test is called the "Paired Samples T-Test".

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	SCouCovI_JAVA	.1864945	101	.18174212	.01808402
	SCouCovI_AJ	.1207381	101	.10496375	.01044428
Pair 2	TCouCovI_JAVA	.0707024	101	.10481631	.01042961
	TCouCovI_AJ	.0548336	101	.07268496	.00723242
Pair 3	SCouCluI_JAVA	.5314356	101	.28067102	.02792781
	SCouCluI_AJ	.3499136	101	.20161386	.02006133
Pair 4	TCouCluI_JAVA	.2341584	101	.33202766	.03303799
	TCouCluI_AJ	.2073550	101	.26787232	.02665429
Pair 5	SCouComI_JAVA	.7015251	101	.25655363	.02552804
	SCouComI_AJ	.6256588	101	.27145847	.02701113
Pair 6	TCouComI_JAVA	.1545595	101	.21916487	.02180772
	TCouComI_AJ	.1603489	101	.20305263	.02020449

Figure 5.20: Paired samples statistics for core modular features coupling

The positive value for the t statistic shows us that the values of the coupling metrics are on the average superior for Java than for AspectJ. Considering a test significance $\alpha = 0.10$ we can conclude from figure 5.22, that for some paired variables we can reject the null hypotheses that the averages of both coupling variables are the same between the two groups, and for others we cannot. Specifically speaking we can say that:

i) for coupling coverage (SCouCovI, TCouCovI), for coupling clustering on the source perspective (SCouCluI) and for coupling commitment also on the source perspective (SCouComI) *there is statistical evidence that the coupling for the OOP modules version is, on average, higher than the coupling of the corresponding AOP modules version*;

[2] Classes and interfaces that existed in the Java patterns and were kept in the AspectJ implementations

		N	Correlation	Sig.
Pair 1	SCouCovI_JAVA & SCouCovI_AJ	101	.553	.000
Pair 2	TCouCovI_JAVA & TCouCovI_AJ	101	.811	.000
Pair 3	SCouCluI_JAVA & SCouCluI_AJ	101	.606	.000
Pair 4	TCouCluI_JAVA & TCouCluI_AJ	101	.807	.000
Pair 5	SCouComI_JAVA & SCouComI_AJ	101	.716	.000
Pair 6	TCouComI_JAVA & TCouComI_AJ	101	.759	.000

Figure 5.21: Paired samples correlations for core modular features coupling

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	SCouCovI_JAVA - SCouCovI_AJ	.06575641	.15145390	.01507023	.03585751	.09565531	4.363	100	.000
Pair 2	TCouCovI_JAVA - TCouCovI_AJ	.01586883	.06252685	.00622165	.00352525	.02821242	2.551	100	.012
Pair 3	SCouCluI_JAVA - SCouCluI_AJ	.18152208	.22535913	.02242407	.13703336	.22601080	8.095	100	.000
Pair 4	TCouCluI_JAVA - TCouCluI_AJ	.02680339	.19622804	.01952542	-.01193448	.06554127	1.373	100	.173
Pair 5	SCouComI_JAVA - SCouComI_AJ	.07586631	.19945750	.01984676	.03649090	.11524172	3.823	100	.000
Pair 6	TCouComI_JAVA - TCouComI_AJ	-.00578943	.14731079	.01465797	-.03487043	.02329156	-.395	100	.694

Figure 5.22: Paired samples test for core modular features coupling

ii) coupling clustering on the target perspective (TCouCluI) and for coupling commitment also on the target perspective (TCouComI), and with a degree of confidence as low as 90%, emphthere is no statistical evidence that the coupling for the OOP modules version is, on average, different from the coupling of the corresponding AOP modules version;

5.5.4 Overall cohesion changes

Research question: Does the refactoring of software systems developed with an OOP language to an AOP extension of the same language produces a significant change on the overall cohesion?

Justification: Regarding cohesion, we expect that the AOP pattern versions are statistically different and exhibiting, as claimed, higher values for cohesion from the OOP ones.

Statistical hypothesis: If we consider that we have two independent groups of cases, one for the Java language and another for the AspectJ language, we can compare the averages for each of the testing variables in the two groups. In other words, we want to test if there is a significant difference between the averages of the variables describing cohesion, for the two languages.

$$H0 : \mu\ SCohComI\ Java = \mu\ SCohComI\ AspectJ$$
$$H1 : \mu\ SCohComI\ Java \neq \mu\ SCohComI\ AspectJ$$

$$H0 : \mu\ TCohComI\ Java = \mu\ TCohComI\ AspectJ$$
$$H1 : \mu\ TCohComI\ Java \neq \mu\ TCohComI\ AspectJ$$

Here we will use again the *Independent Samples T-Test*.

	Language	N	Mean	Std. Deviation	Std. Error Mean
SCohComI	Java	117	.0548843	.08442152	.00780477
	AspectJ	167	.0334034	.06270103	.00485195
TCohComI	Java	117	.0616623	.09064244	.00837990
	AspectJ	167	.0408484	.07643820	.00591497

Figure 5.23: Paired samples statistics for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ

Results interpretation: Based upon the result of the Levene test and an $\alpha = 0.05$ we can say that we have to reject the hypothesis of the equality of variances between the two groups, for both testing variables. Therefore we have to choose the rows signalized with *Equal variances not assumed*. The positive value for the t statistic shows us that the values of the cohesion metrics are on the average superior for Java than for AspectJ. Considering a test significance $\alpha = 0.05$, we can conclude from figure 5.24 that we must

Testing Variable		Levene's Test for Equality of Variances		t-test for Equality of Means						
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
									Lower	Upper
SCohComI	Equal variances assumed	14.777	.000	2.460	282	.014	.02148087	.00873210	.00429251	.03866923
	Equal variances not assumed			2.337	201.912	.020	.02148087	.00918999	.00336021	.03960153
TCohComI	Equal variances assumed	9.608	.002	2.091	282	.037	.02081384	.00995564	.00121703	.04041065
	Equal variances not assumed			2.029	221.893	.044	.02081384	.01025717	.00059991	.04102777

Figure 5.24: Levene's and t-test for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ

reject the null hypotheses that the averages of both cohesion variables are the same between the two groups. In other words, we can say that there is statistical evidence that *the cohesion for the OOP versions is, on average, higher than the cohesion for AOP module versions*.

These results, in spite of not corroborating the most common claims regarding cohesion of AOP systems, had already been announced in section 4.5 when analyzing cohesion for the *Screen* class from the *Observer*, and has now been evidenced by the remainder design patterns. As explained earlier, we believe these results do not evidence the real cohesion differences but, instead, the differences between the *delusional* cohesion at the OOP implementations, due to the code tangling and the cohesion of their AOP equivalent versions.

5.5.5 Core modular features cohesion changes

Research question: Does the refactoring of software systems developed with an OOP language to an AOP extension of the same language produces a significant change on the cohesion of the core modular features?

With a rotation operation we were able to produce a paired list version of our data, so that for equivalent modular features, we get two variables within the same case, representing the values for the same variable in Java and in AspectJ, like for instance: TCohComI-Java and TCohComI-AJ. Notice that we get fewer cases than for independent tests, because only classes and interfaces are *pairable*. The hypotheses are the same as above. The corresponding test is called the *Paired Samples T-Test*.

Results interpretation: The positive value for the *t* statistic shows us that the values of the cohesion metrics are on the average superior for Java than for AspectJ. Considering a test significance $\alpha = 0.10$ we can conclude from figure 5.27, that we must reject the null hypotheses that the averages of both cohesion variables are the same between the two groups. Again we confirm that there is statistical evidence that *the cohesion for the OOP modules version is, on average, higher than the cohesion of the corresponding AOP modules version*.

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	SCohComl_JAVA	.0566006	101	.08628742	.00858592
	SCohComl_AJ	.0466738	101	.07270910	.00723483
Pair 2	TCohComl_JAVA	.0644524	101	.09312942	.00926672
	TCohComl_AJ	.0544761	101	.08586724	.00854411

Figure 5.25: Paired descriptive statistics for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ

		N	Correlation	Sig.
Pair 1	SCohComl_JAVA & SCohComl_AJ	101	.816	.000
Pair 2	TCohComl_JAVA & TCohComl_AJ	101	.777	.000

Figure 5.26: Paired correlations for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
				Lower	Upper			
SCohComl_JAVA - SCohComl_AJ	.00992681	.04990227	.00496546	.00007547	.01977814	1.999	100	.048
TCohComl_JAVA - TCohComl_AJ	.00997624	.06015491	.00598564	-.00189909	.02185157	1.667	100	.099

Figure 5.27: *t-test* for *SCohComI* and *TCohComI* for the Gof in Java and AspectJ

5.5.6 Overall modularity changes

Research question: Does the refactoring of software systems developed with an OOP language to an AOP extension of the same language produce a significant change on the overall modularity?

Justification: We expect that the AOP pattern versions are statistically different and exhibiting, as claimed, higher values for modularity from the OOP ones.

Statistical hypothesis: If we consider that we have two independent groups of cases, one for the Java language and another for the AspectJ language, we can compare the averages for each of the testing variables in the two groups. In other words, we want to test if there is a significant difference between the averages of the variables describing modularity, for the two languages.

$$H0 : \mu\, PCI\ Java = \mu\, PCI\ AspectJ$$
$$H1 : \mu\, PCI\ Java \neq \mu\, PCI\ AspectJ$$

$$H0 : \mu\, MMF\ Java = \mu\, MMF\ AspectJ$$
$$H1 : \mu\, MMF\ Java \neq \mu\, MMF\ AspectJ$$

Here we will use once again the *Independent Samples T-Test*.

	Language_	N	Mean	Std. Deviation	Std. Error Mean
PCI	Java	117	.4604379	.17936594	.01658239
	AspectJ	167	.2941760	.15235661	.01178971
MMF	Java	117	.0275730	.01458509	.00134839
	AspectJ	167	.0431848	.03348947	.00259149

Figure 5.28: Paired samples statistics for *PCI* and *MMF* for the GoF patterns in Java and AspectJ

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
PCI	Equal variances assumed	9.466	.002	8.409	282	.000	.16626183	.01977281	.12734080	.20518286
	Equal variances not assumed			8.172	223.081	.000	.16626183	.02034632	.12616625	.20635741
MMF	Equal variances assumed	64.138	.000	-4.736	282	.000	-.01561179	.00329665	-.022101	-.009123
	Equal variances not assumed			-5.344	242.603	.000	-.01561179	.00292130	-.021366	-.009857

Figure 5.29: Levene's and t-test for *PCI* and *MMF* for the GoF patterns in Java and AspectJ

Results interpretation: Based upon the result of the Levene's test and an $\alpha = 0.05$ we can say that we have to reject the hypothesis of the equality of variances between the

two groups, for both testing variables. Therefore we have to choose the rows signalized with *Equal variances not assumed*.

The positive value for the t statistic shows us that the values of the potential crosscutting metric (PCI) is on the average superior for Java than for AspectJ. Considering a test significance $\alpha = 0.05$, we can conclude from figure 5.29 that we must reject the null hypotheses that the averages of the potential crosscutting is the same between the two groups. In other words, we can say that there is statistical evidence that *the potential crosscutting for the OOP versions is, on average, higher than the potential crosscutting for the AOP versions*, which, as explained earlier, between functional equivalent systems also means that modularity is higher on the AOP versions of those systems.

The negative value for the t statistic shows us that the values of the modularity merit factor (MMF) is on the average inferior for Java than for AspectJ. Considering a test significance $\alpha = 0.05$, we can conclude from figure 5.29 that we must reject the null hypotheses that the averages of the modularity merit factor is the same between the two groups. In other words, we can say that there is statistical evidence that *the modularity merit factor for the OOP versions is, on average, lower than the modularity merit factor for the AOP versions*, meaning that modularity is higher on the AOP versions of those systems.

[This page was intentionally left blank]

Chapter 6

Related Work

Contents

6.1 Introduction **82**

6.2 PIFACTORS **82**

6.3 PIMETRICS **83**

6.4 AOP Modularity Improvements Evaluation **84**

In this chapter an overview of the related work is presented and discussed.

6.1 Introduction

In this section, the work related to this dissertation is summarized and discussed. For this purpose it was grouped according to its relation with the PIFACTORS, the PIMETRICS and the AOP Modularity improvements evaluation.

6.2 PIFACTORS

Eder et al. coupling dimensions, interaction, component and inheritance coupling [EKS92] are heavily focused at object-oriented coupling mechanisms. This is a good example of how metrics for paradigm independent software properties like modularity have been defined so far, highly dependent on paradigms.

Hitz and Montazeri coupling levels, class and object level coupling [HM95] address static and dynamic coupling. The PIFACTORS address the static coupling and cohesion only. Anyway, dynamic coupling is believed to interfere with performance and testing issues and not so much with understandibility and reuse, therefore being out of the scope of a research on modularity like this.

Hitz and Montazeri and Briand et al. corroborate the importance of the *type* modularity factor. The former redefined coupling types for OOP [HM95], while the latter states that the strength of coupling between two classes is determined by the type of connections between the classes [BBDW99].

Briand corroborates the importance of the *density* modularity factor when mentioning that coupling between two classes is determined by two things, frequency and types of connections among modules [BBDW99].

Briand work is presented in [BBDW99] is focused essentially on coupling relations available during the high level design phase, more concretely coupling caused by interactions that occur between classes. His framework heavily focuses on object-oriented coupling mechanisms. However, by introducing the *locus of impact* kind of coupling interaction, he corroborates the relevance of the *role* modularity factor.

Ceccatto and Tonella seem to corroborate the relevance of the *coverage* factor, considering their CMC[1] and CFA[2] metrics [CT04]. Briand et al. also seem to agree on the relevance of this factor, at least from the *source* point of view, when considering that a class coupling depends on the set of invoked methods like at their SIM[3] and PIM[4] metrics [BBDW99].

Ceccato and Tonella enforce the idea of the *target role* importance on the AOP context by considering aspects new coupling mechanisms, where classes ignore their existence [CT04].

[1]Coupling on method call
[2]Coupling on field access
[3]Statically invoked metrics
[4]Polimorphically invoked metrics

Neto et al. [NRD+07] proposed the adoption of the concept *semantic coupling*, as an opposition to *syntactic coupling*, being the latter the most common one at OOP, where it is generated by the syntactic constructs of the adopted language, while the former emerges from new paradigms and languages, like AspectJ, where the dependencies may not be so explicit. He also refers that *semantic coupling* does not provoke compilation errors, however some problems may occur. This sort of *side effects*, specific to a programming language, should not be called coupling.

To summarize, the PIFACTORS capture, in a paradigm independent way, most of the concerns, related with modularity, evidenced by many paradigm specific metrics defined so far. The PIFACTORS also capture the clustering and commitment factors which are not usually addressed by traditional metrics.

6.3 PIMETRICS

Several coupling measurement frameworks exist like those by Eder et al. [EKS92], Hitz and Montazeri [HM95] and Briand et al. [HM95], but none of those has an approach like the PIMETRICS, independent of paradigm or programming languages.

We disagree when Briand concludes, after analyzing the empirical differences between two classes using MPC[5] and DAC[6], that there is not *one right way* to count frequencies of connections. There must be only one way to count them, otherwise, we will never be able to measure the same thing. Our *density* modularity factor addresses this subject.

Sant'Anna et al. proposed a framework [San03] for assessing Aspect-Oriented software. This framework extends some OOP traditional metrics into AOP, to enable cross-paradigm comparisons, and proposes a set of metrics to evaluate the separation of concerns, also at both paradigms. The metrics from this framework, apparently were not validated or formalized and we are not aware of any study about their orthogonality. For comparability and experiment replication sake, we must guarantee precision on metrics definition, which can only be achieved through formality. Even if this had been accomplished, it is our belief that, after analyzing the scope of each of the measurements defined it is very narrow, when compared to the PIFACTORS.

Some authors like Ceccato and Tonella [CT04], Zhao [Zha04] and other, also defend the idea that OOP metrics must be adapted or extended to cover AOP new coupling mechanisms. These are good examples of how researchers on metrics are dealing with paradigm shifts. They often adapt *old* metrics to cope with the innovative features proposed in formalisms supporting the new paradigm. However, the basic architectural concepts such as coupling, cohesion and modularity have not changed over the last decades. If we base our metrics on those basic concepts, instead of proposing paradigm

[5] Message passing coupling
[6] Data abstraction coupling

tailored-based metrics, we can compare systems built in different paradigms.

We disagree with Ceccato and Tonella [CT04] when, grounding their CAE[7] metric, they state that if the behavior of an operation can be altered by an advice, due to a pointcut intercepting it, there is an (implicit) dependency of the operation from the advice. The method should not depend at all from the advice. If it does, then the design is probably not right. Each class should implement its own concern alone regardless of the paradigm being used. When the method behaviour depends on the advice, either the method is implementing more than one concern, or the concern the method is implementing is scattered between the method and the advice, which should not occur also.

Too summarize, we are not aware of any other paradigm independent metrics for modularity. All the metrics discussed have paradigm dependent definitions and to provide cross-paradigm comparisons, some of these metrics have evolved to support the features of the new paradigms. Unlike the PIMETRICS, most of the metrics being used to quantitatively compare the modularity of OOP and AOP systems are not validated or formalized.

6.4 AOP Modularity Improvements Evaluation

There seems to exist two streams, when it comes to the evaluation of AOP modularity improvements. One makes use of the *Design Structure Matrixes*(DSM) and other related developments, like the one in [CCH07], while the other makes use of metrics, as in [BBDW99].

Some approaches based on DSM claim [LB06] that metrics are not thought of for *assessing design options at certain decision points*, but can be useful for *a posteriori* analysis. As a matter of fact, there are today many software metrics for assessing the software design like those proposed in [BBA02], and its usage is increasing to a point where commercial tools are being developed to support this practice, like the ones from [SDM06] or [Bor08]. The PIMETRICS can also be used for design evaluations, as long as the PIMETA is instantiated with the features from the design language being used. Metrics in general can be used for estimation models or even for identifying poor design decisions and proposing alternatives, like suggested in [BA08].

Hannemann and Kiczales [HK02] presented the, to the best of our knowledge, first functional equivalent implementation of the gang-of-four design patterns in Java and AspectJ. Along with this implementation's presentation came a comparative study, where the authors claimed modularity improvements in 17 of the 23 design patterns. Although being a qualitative study, it should be taken into consideration the empirical understanding of the paradigm effects on modularity, given the authors expertise on the subject.

[7]Coupling on advice execution

In [GSF+05], a quantitative study on the modularization of the design patterns with aspects is presented. The conclusions seem appealing, however the fact of being grounded on the framework proposed in [San03], which is not validated or formalized, poses a serious threat to the validity of those conclusions. Still, there is an apparent contradiction in [San03] that caught our attention. Within the results presented, the *prototype*, *command*, *iterator*, *chain of responsability*, *strategy* and *memento* design patterns in AspectJ are ranked among those that present the highest separation of concerns. However, they are simultaneously among those that presented the best coupling and cohesion in their OOP versions, suggesting that to a decrease in modularity corresponds an increase in the separation of concerns.

Kulesza presents another study [KSG+06] on the effects of aspect-oriented programming on maintainability. Again, the conclusions clearly favour aspect-oriented programming. However, this work is once again grounded on the same framework discussed above [San03] and therefore its conclusions are not well grounded.

Lopes and Bajra [LB06] proposed the assessment of aspect modularizations by using the *design structure matrix*(DSM) and the net option value(NOV), and present a case study where several object-oriented and aspect-oriented design variants for a software application are compared and analyzed. They concluded that aspects can be beneficial as well as detrimental. This study was grounded on a small, *however relatively complex example*, which inhibited more ambitious conclusions beyond the scope of the system itself.

Cai and Huynh [CCH07] proposed an evolution model for software modularity assessment using a *decision tree* based assessment framework grounded on *augmented constraint networks*. The authors claimed its independency of particular languages and paradigms and that the proposed model quantitatively corroborates the qualitative claims of Hannemann and Kiczales [HK02] about the modularity improvements on the Observer design pattern. However, the fact of only encompassing one simple example reduces the external validity of the conclusions.

The above DSM-based approaches do not seem to provide a scale against which the results can be compared, to find out whether a design is good or bad. Their theoretical validity was not discussed and they reach slightly different conclusions. While in [CCH07] the qualitative claims in [HK02] are corroborated, in [LB06] aspects are considered to be detrimental as well.

From all the above, we can conclude that most OOP versus AOP modularity evaluations made so far are grounded on poor metrics or processes meaning that the validity of their conclusions is threatened. We believe to have contributed with a more solid ground on which these kind of evaluations can be performed and achieve stronger results.

[This page was intentionally left blank]

Chapter 7

Conclusions and Future work

Contents

7.1 Conclusions 88
7.2 Threats to validity 89
7.3 Future work 90

This chapter summarizes the achievements of the research work described in this thesis and lists some open issues, which should and hopefully will ground our future research work.

7.1 Conclusions

We have defined a meta-model (PIMETA) and a set of metrics (PIMETRICS) based on our intuitive notions about the modularity within a software system described by the PIFACTORS. The PIMETRICS were also internally validated and formalized in OCL upon the PIMETA.

We applied the M2DM technique, with the PIMETA and the PIMETRICS, at the GoF Design Patterns functionally equivalent implementations, developed in Java and AspectJ by Hannemann and Kiczales, and measured the modularity from both implementations.

We used statistic tests and techniques to consolidate the PIMETRICS adequacy, to validate the results obtained and to support our conclusions.

Our conclusions about the modularity improvements with aspect-oriented programming are as follows:

i) The coupling of the coarser grained modules (classes and interfaces) which belong to the OOP version of the GoF design patterns in Java is, on average, higher than the coupling of the coarser grained modules (aspects, classes and interfaces) which belong to the AOP version of the same patterns in AspectJ. This difference is around 11% on average on all indexes, where the lowest difference is 7% and the highest is 20%;

ii) The cohesion of the coarser grained modules which belong to the OOP version of the GoF design patterns in Java is, on average, higher than the cohesion of the coarser grained modules which belong to the AOP version of the same patterns in AspectJ. This difference is around 20% on average on both indexes. This decrease in the AspectJ version of the patterns is, in our opinion, due to the fact of being compared with the cohesion in the modules of the Java version influenced by the non-functional cohesion inherent to code tangling which we named *delusional cohesion*;

iii) The potential crosscutting in the Java version of the patterns is, on average, higher than the potential crosscutting in the AspectJ version of the same patterns. This difference is around 16% and means also modularity improvements for functional equivalent systems, as explained earlier;

iv) The modularity merit factor in the Java version of the patterns is, on average, lower than the modularity merit factor in the AspectJ version of the same patterns. This difference is around 16%;

v) Modularity does improve when OOP systems are refactored to AOP, but these improvements, according to our results, are far more moderate than those claimed so far in equivalent studies like [SGK+04] or [KSG+06].

We believe to have provided a sound contribute towards the AOP modularity improvements evaluation, as well as towards the paradigm independent evaluation of software.

7.2 Threats to validity

Some of the PIFACTORS evidenced some unexpected values of high correlation. Eventhough they could be explained by the sample characteristics, future work should be conducted to clarify this point.

The PIMETA has only been instantiated with Java and AspectJ. To truly prove its adequacy for the representation of the features and dependencies from different paradigm and languages, it should be instantiated with some more distinct examples.

The PIMETRICS do not measure indirect dependencies [HYYT05], that is, ambiguous dependencies that are the consequence of explicit or direct dependencies, from which properties like indirect coupling can derive. Therefore, in spite of not being clear the relevance of these dependencies, they were not considered within the presented modularity evaluation.

The PIMETRICS do not distinguish when a dependency is towards a feature implemented natively in a feature different from its target(e.g. A method calling an inherited method from the child class. Should this be a double dependency, a direct dependency towards the parent, or as now is being considered, a direct dependency towards the child?). Even though this case has not been addressed, it is recognized as relevant enough to be followed.

The dependency type weight has been set to 1, for not being known, up to date, an appropriate scale which can be used for the dependencies of Java or AspectJ. This implies the consent that all dependencies have the same effect on modularity, in spite of not being at all our conviction. It is believed that these weights must be defined for each dependency from each language willing to be compared, and should come from a thorough statistical analysis based on maintenance effort.

The design patterns implementation in Java and AspectJ is a very interesting case study, yet small. If the systems under study are big and complex, the credibility of the results achieved is higher, granted that the experiments are properly conducted and validated.

The PIMETRICS must be used with larger systems. Even if only at systems programmed with the same language, the sample size is very important to externally validate the metrics with, for instance, maintenance effort. If used in systems implemented with both paradigms (OOP and AOP), it may help to consolidate the importance of PIMETRICS for non-paradigm modularity evaluation as well as it may consolidate the evaluation of the modularity benefits of AOP.

To improve the reliability of the automated data collection and measurement pro-

cess, more validation and enhancement is required. The AspectJ-Front parser evidenced the lack of some features that would be of value like, for instance, an *xml schema* for the generated xml files.

7.3 Future work

In this section, some potential research threads are presented and grouped by the following areas.

7.3.1 Quantitative Modularity Evaluation

i) Our dimensionless PIMETRIC indexes are supposed to be size independent and to represent different modularity factors. This suggests the following research questions:

   (a) Are PIMETRIC indexes size-independent?

   (b) Do PIMETRICS represent non-overlapping modularity factors?

   (c) Has the role factor an effect on the modularity?

   (d) Has the coverage factor an effect on modularity?

   (e) Has the commitment factor an effect on the modularity?

   (f) Has the clustering factor an effect on the modularity?

   (g) Has the density factor an effect on the modularity?

   (h) Is the distribution of modularity improvement the same across design patterns?

ii) Clarify the high correlation presented between some of the PIFACTORS, namely by applying the PIMETRICS to larger and real world examples.

iii) Define ordinal scales and weights for the dependencies of Java and AspectJ, considering their impact on modularity.

iv) Study indirect dependencies impact on modularity and consider evolving the PIMETRICS accordingly.

v) Study delusional cohesion.

vi) Validate the PIMETRICS externally using maintenance effort as the dependent variable.

7.3.2 Refactoring

i) Implementation of a process supported by a tool to suggest and implement cross-paradigm refactorings, from OOP to AOP, based on the PIMETRICS capability to identify code smells, as described in [BA08].

ii) Evolve the current scripts used in the measurement process into a meta-model based measurement and refactoring prototype.

[This page was intentionally left blank]

Appendix A

Java Feature and Dependency Types

This appendix presents the Java Feature Types, their mutual aggregation possibilities and their DependencyTypes with which PIMETA is instantiated.

ModularFeatureTypes
`Package`
`Class`
`Interface`
`Method`
`Constructor`
AtomicFeatureTypes
`Field`
`LocalVariable`
`Enumeration`
`Parameter`
`ReturnValue`
`Exception`
Aggregations
`(Package, Package)`
`(Package, Class)`
`(Package, Interface)`
`(Class, Class)`
`(Class, Interface)`
`(Class, Method)`
`(Class, Constructor)`
`(Class, Field)`
`(Class, Enumeration)`
`(Method, LocalVariable)`
`(Method, Class)`
`(Method, Parameter)`
`(Method, ReturnValue)`
`(Method, Enumeration)`
`(Constructor, LocalVariable)`
`(Constructor, Class)`
`(Constructor, Parameter)`
`(Constructor, Enumeration)`
`(Interface, Method)`
`(Interface, Constructor)`
`(Interface, Enumeration)`
`(Interface, Field)`
`(Interface, Class)`
`(Interface, Interface)`

Table A.1: Java Feature Types

DependencyTypes
`ClassInheritance : (Class, Class)`
`ClassInterfaceImplementation : (Class, Interface)`
`ClassImportsPackage : (Class, Package)`
`ClassImportsClass : (Class, Class)`
`ClassCallsConstructor : (Class, Constructor)`
`InterfaceInheritance : (Interface, Interface)`
`InterfaceImportsPackage : (Interface, Package)`
`InterfaceImportsClass : (Interface, Class)`
`MethodUsesField : (Method, Field)`
`MethodUsesEnumeration : (Method, Enumeration)`
`MethodCallsMethod : (Method, Method)`
`MethodCallsConstructor : (Method, Constructor)`
`MethodRaisesException : (Method, Exception)`
`ConstructorUsesField : (Constructor, Field)`
`ConstructorUsesEnumeration : (Constructor, Enumeration)`
`ConstructorCallsMethod : (Constructor, Method)`
`ConstructorCallsConstructor : (Constructor, Constructor)`
`ConstructorRaisesException : (Constructor, Exception)`
`ReturnValueType : (ReturnValue, Class)`
`ParameterType : (Parameter, Class)`
`ExceptionType : (Exception, Class)`
`FieldType : (Field, Class)`
`LocalVariableType : (LocalVariable, Class)`

Table A.2: Java Dependency Types

[This page was intentionally left blank]

Appendix B

AspectJ Feature and Dependency Types

This appendix presents the AspectJ Feature Types, their mutual aggregation possibilities and their DependencyTypes with which PIMETA is instantiated.

ModularFeatureTypes
`Aspect`
`AdviceBefore`
`AdviceAround`
`AdviceAfter`
`InterTypeOperation`
`PointcutDefinition`
AtomicFeatureTypes
`InterTypeField`
`DeclareParents`
`DeclareWarning`
`DeclareError`
`DeclareSoft`
`DeclarePrecedence`
`DeclareAtType`
`DeclareAtMethod`
`DeclareAtConstructor`
`DeclareAtField`
`Pointcut`

Table B.1: AspectJ Feature Types

Aggregations
(Package, Aspect)
(Class, Aspect)
(Aspect, Aspect)
(Aspect, Class)
(Aspect, Interface)
(Aspect, Method)
(Aspect, Field)
(Aspect, Enumeration)
(Aspect, AdviceBefore)
(Aspect, AdviceAround)
(Aspect, AdviceAfter)
(Aspect, InterTypeOperation)
(Aspect, PointcutDefinition)
(Aspect, InterTypeField)
(Aspect, DeclareParents)
(Aspect, DeclareWarning)
(Aspect, DeclareError)
(Aspect, DeclareSoft)
(Aspect, DeclarePrecedence)
(Aspect, DeclareAtType)
(Aspect, DeclareAtMethod)
(Aspect, DeclareAtConstructor)
(Aspect, DeclareAtField)
(PointcutDefinition, Pointcut)
(PointcutDefinition, Parameter)
(AdviceBefore, Pointcut)
(AdviceBefore, LocalVariable)
(AdviceBefore, Class)
(AdviceBefore, Parameter)
(AdviceBefore, ReturnValue)
(AdviceAround, Pointcut)
(AdviceAround, LocalVariable)
(AdviceAround, Parameter)
(AdviceAround, ReturnValue)
(AdviceAfter, Pointcut)
(AdviceAfter, LocalVariable)
(AdviceAfter, Class)
(AdviceAfter, Parameter)
(AdviceAfter, ReturnValue)
(InterTypeOperation, Pointcut)
(InterTypeOperation, LocalVariable)
(InterTypeOperation, Class)
(InterTypeOperation, Parameter)
(InterTypeOperation, ReturnValue)

Table B.2: AspectJ Features (cont.)

DependencyTypes

```
AspectInheritsFromClass : (Aspect, Class)
AspectInheritsFromAspect : (Aspect, Aspect)
AspectImplementsInterface : (Aspect, Interface)
AspectImportsPackage : (Aspect, Package)
AspectImportsClass : (Aspect, Class)
AspectCallsConstructor : (Aspect, Constructor)
MethodUsesInterTypeField : (Method, InterTypeField)
MethodCallsInterTypeOperation : (Method, InterTypeOperation)
ConstructorUsesInterTypeField : (Constructor, InterTypeField)
ConstructorCallsInterTypeOperation : (Constructor, InterTypeOperation)
AdviceBeforeUsesField : (AdviceBefore, Field)
AdviceBeforeUsesEnumeration : (AdviceBefore, Enumeration)
AdviceBeforeCallsMethod : (AdviceBefore, Method)
AdviceBeforeCallsConstructor : (AdviceBefore, Constructor)
AdviceBeforeRaisesException : (AdviceBefore, Class)
AdviceBeforeUsesPointcutDefinition : (AdviceBefore, PointcutDefinition)
AdviceBeforeUsesInterTypeField : (AdviceBefore, InterTypeField)
AdviceBeforeCallsInterTypeOperation : (AdviceBefore, InterTypeOperation)
AdviceAroundUsesField : (AdviceAround, Field)
AdviceAroundUsesEnumeration : (AdviceAround, Enumeration)
AdviceAroundCallsMethod : (AdviceAround, Method)
AdviceAroundCallsConstructor : (AdviceAround, Constructor)
AdviceAroundRaisesException : (AdviceAround, Class)
AdviceAroundUsesPointcutDefinition : (AdviceAround, PointcutDefinition)
AdviceAroundUsesInterTypeField : (AdviceAround, InterTypeField)
AdviceAroundCallsInterTypeOperation : (AdviceAround, InterTypeOperation)
AdviceAfterUsesField : (AdviceAfter, Field)
AdviceAfterUsesEnumeration : (AdviceAfter, Enumeration)
AdviceAfterCallsMethod : (AdviceAfter, Method)
AdviceAfterCallsConstructor : (AdviceAfter, Constructor)
AdviceAfterRaisesException : (AdviceAfter, Class)
AdviceAfterUsesPointcutDefinition : (AdviceAfter, PointcutDefinition)
AdviceAfterUsesInterTypeField : (AdviceAfter, InterTypeField)
AdviceAfterCallsInterTypeOperation : (AdviceAfter, InterTypeOperation)
InterTypeOperationUsesField : (InterTypeOperation, Field)
InterTypeOperationUsesEnumeration : (InterTypeOperation, Enumeration)
InterTypeOperationCallsMethod : (InterTypeOperation, Method)
InterTypeOperationCallsConstructor : (InterTypeOperation, Constructor)
InterTypeOperationRaisesException : (InterTypeOperation, Class)
InterTypeOperationDestinationClass : (InterTypeOperation, Class)
InterTypeOperationDestinationAspect : (InterTypeOperation, Aspect)
InterTypeOperationUsesInterTypeField : (InterTypeOperation, InterTypeField)
InterTypeOperationCallsInterTypeOperation : (InterTypeOperation, InterTypeOperation)
InterTypeFieldType : (InterTypeField, Class)
InterTypeFieldDestinationClass : (InterTypeField, Class)
InterTypeFieldDestinationAspect : (InterTypeField, Aspect)
```

Table B.3: AspectJ Dependency Types

DependencyTypes
`DeclareParentsParentClass : (DeclareParents, Class)`
`DeclareParentsParentInterface : (DeclareParents, Interface)`
`DeclareParentsParentAspect : (DeclareParents, Aspect)`
`DeclareParentsChildClass : (DeclareParents, Class)`
`DeclareParentsChildInterface : (DeclareParents, Interface)`
`DeclareParentsChildAspect : (DeclareParents, Aspect)`
`DeclareParentsImplementedInterface : (DeclareParents, Interface)`
`DeclareParentsClassImplementingInterface : (DeclareParents, Class)`
`DeclareParentsAspectImplementingInterface : (DeclareParents, Aspect)`
`DeclareWarningUsesPointcutDefinition : (DeclareWarning, PointcutDefinition)`
`DeclareErrorUsesPointcutDefinition : (DeclareError, PointcutDefinition)`
`DeclareSoftUsesPointcutDefinition : (DeclareSoft, PointcutDefinition)`
`DeclarePrecedenceUsesAspect : (DeclarePrecedence, Aspect)`
`DeclareAtTypeUsesClass : (DeclareAtType, Class)`
`DeclareAtTypeUsesAspect : (DeclareAtType, Aspect)`
`DeclareAtMethodUsesPointcutDefinition : (DeclareAtMethod, PointcutDefinition)`
`DeclareAtConstructorUsesClass : (DeclareAtConstructor, Class)`
`DeclareAtFieldUsesClass : (DeclareAtField, Class)`
`PointcutUsesPointcutDefinition : (Pointcut, PointcutDefinition)`
`PointcutUsesPackage : (Pointcut, Package)`
`PointcutUsesClass : (Pointcut, Class)`
`PointcutUsesInterface : (Pointcut, Interface)`
`PointcutUsesAspect : (Pointcut, Aspect)`
`PointcutUsesMethod : (Pointcut, Methhod)`
`PointcutUsesConstructor : (Pointcut, Constructor)`
`PointcutUsesField : (Pointcut, Field)`

Table B.4: AspectJ Dependency Types (cont.)

[This page was intentionally left blank]

Appendix C

PIMETRICS primitives

This appendix presents the primitive functions in OCL upon which the PIMETRICS are formalized.

Name	**enclosingModularFeature()**
Informal definition	Modular feature which encloses this feature

Formal definition	*PIMETA::Feature*
	`enclosingModularFeature() : ModularFeature =`
	`self.modularFeature`

Name	**rootFeature()**
Informal definition	Modular feature which encloses all the modular features which enclose this modular feature

Formal definition	*PIMETA::Feature*
	`rootFeature(): ModularFeature =`
	`if self.modularFeature = oclUndefined(ModularFeature)`
	`then`
	`self.oclAsType(ModularFeature)`
	`else`
	`self.modularFeature.rootFeature()`
	`endif`

Name	**modularSiblings()**
Informal definition	My sibling features

Formal definition	*PIMETA::Feature*
	`modularSiblings(): Set(Feature) =`
	`if enclosingModularFeature() = oclUndefined(ModularFeature)`
	`then`
	`oclEmpty(Set(Feature))`
	`else`
	`enclosingModularFeature().allEnclosedFeatures()->`
	`excluding(self)`
	`endif`

Name	**siblingDependedFeatures()**
Informal definition	Sibling features on which this feature depends

Formal definition	*PIMETA::Feature*
	`siblingDependedFeatures(): Set(Feature) =`
	`self.modularSiblings()->`
	`intersection(self.allDependedFeatures().`
	`allEnclosingModularFeaturesRecursively())`

Name	**siblingDependentFeatures()**
Informal definition	Sibling features which depend on this feature

Formal definition	*PIMETA::Feature*
	`siblingDependentFeatures(): Set(Feature) =`
	`self.modularSiblings()->`
	`intersection(self.allDependentFeatures().`
	`allEnclosingModularFeaturesRecursively())`

Table C.1: Feature operations

Name	**allEnclosingModularFeaturesRecursively()**
Informal definition	All features which enclose this feature recursively

Formal definition	*PIMETA::Feature*
	allEnclosingModularFeaturesRecursively():
	Set(ModularFeature) =
	if self.modularFeature = oclUndefined(ModularFeature)
	then
	oclEmpty(Set(ModularFeature))
	else
	self.modularFeature.
	allEnclosingModularFeaturesRecursively()->
	including(self.modularFeature)
	endif

Name	**internalDependencies()**
Informal definition	All dependencies among the features enclosed by me
Formal definition	*PIMETA::Feature* internalDependencies(): Set(Dependency) = oclEmpty(Set(Dependency))
Name	**inboundDependencies()**
Informal definition	All dependencies which destinations are any of my enclosed features
Formal definition	*PIMETA::Feature* inboundDependencies(): Set(Dependency) = self.dependencyDestinations
Name	**outboundDependencies()**
Informal definition	All dependencies which origins are any of my enclosed features
Formal definition	*PIMETA::Feature* outboundDependencies(): Set(Dependency) = self.dependencyOrigins
Name	**externalDependencies()**
Informal definition	All dependencies which origins and destinations are not any of my enclosed features
Formal definition	*PIMETA::Feature* externalDependencies(): Set(Dependency) = Dependency.allInstances-internalDependencies()- inboundDependencies()-outboundDependencies()
Name	**internalFeatures()**
Informal definition	All features enclosed by me
Formal definition	*PIMETA::Feature* internalFeatures(): Set(Feature) = oclEmpty(Set(Feature))

Table C.2: Feature operations (cont.)

Name	**externalFeatures()**
Informal definition	All features not enclosed by me
Formal definition	*PIMETA::Feature* `externalFeatures(): Set(Feature) = (Feature.allInstances->reject(oclIsTypeOf(Feature))-self.internalFeatures())->excluding(self)`
Name	**allDependedFeatures()**
Informal definition	All features which I depend on
Formal definition	*PIMETA::Feature* `allDependedFeatures(): Set(Feature) = self.dependencyOrigins.destination->asSet`
Name	**allDependentFeatures()**
Informal definition	All features which depend on me
Formal definition	*PIMETA::Feature* `allDependentFeatures(): Set(Feature) = self.dependencyDestinations.origin->asSet`
Name	**internalDependedFeatures()**
Informal definition	All enclosed features from which other features depend on
Formal definition	*PIMETA::Feature* `internalDependedFeatures(): Set(Feature) = oclEmpty(Set(Feature))`
Name	**internalDependentFeatures()**
Informal definition	All enclosed features which are dependent on other features
Formal definition	*PIMETA::Feature* `internalDependentFeatures(): Set(Feature) = oclEmpty(Set(Feature))`
Name	**externalDependedFeatures()**
Informal definition	All external features which I depend on
Formal definition	*PIMETA::Feature* `externalDependedFeatures(): Set(Feature) = self.allDependedFeatures() - self.internalDependedFeatures()`

Table C.3: Feature operations (cont.)

Name	**externalDependentFeatures()**
Informal definition	All external features which depend on me
Formal definition	*PIMETA::Feature* externalDependentFeatures(): Set(Feature) = self.allDependentFeatures() - self.internalDependentFeatures()
Name	**allDependencies()**
Informal definition	All dependencies from which I am the origin
Formal definition	*PIMETA::Feature* allDependencies(): Set(Dependency) = Dependency.allInstances->select(origin=self)
Name	**hasCommonRoot()**
Informal definition	True if the provided feature shares its root with me
Formal definition	*PIMETA::Feature* hasCommonRoot(other: Feature): Boolean = self.rootFeature() = other.rootFeature()
Name	**featuresWithoutCommonRoot()**
Informal definition	All features which do not have the same root as I do
Formal definition	*PIMETA::Feature* featuresWithoutCommonRoot(): Set(Feature) = self.externalFeatures()-> reject(f: Feature | self.hasCommonRoot(f))
Name	**featuresWithCommonRoot()**
Informal definition	All features which share their root with me
Formal definition	*PIMETA::Feature* featuresWithCommonRoot(): Set(Feature) = Feature.allInstances->excluding(self) - featuresWithoutCommonRoot()
Name	**externalDependedFeaturesSharedWith()**
Informal definition	Common external depended features between the current feature and the "other" feature
Formal definition	*PIMETA::Feature* externalDependedFeaturesSharedWith(other: ModularFeature): Set(Feature) = self.externalDependedFeatures()-> intersection(other.externalDependedFeatures())

Table C.4: Feature operations (cont.)

Name	**externalDependentFeaturesSharedWith()**
Informal definition	Common external dependent features between the current feature and the "other" feature
Formal definition	*PIMETA::Feature* `externalDependentFeaturesSharedWith(other: ModularFeature): Set(Feature) = self.externalDependentFeatures()-> intersection(other.externalDependentFeatures())`
Name	**externalDependedFeaturesAggregatedBy()**
Informal definition	External depended features belonging to the "other" feature
Formal definition	*PIMETA::Feature* `externalDependedFeaturesAggregatedBy(other: ModularFeature): Set(Feature) = self.externalDependedFeatures()-> intersection(other.internalFeatures())`
Name	**externalDependentFeaturesAggregatedBy()**
Informal definition	External dependent features belonging to the "other" feature
Formal definition	*PIMETA::Feature* `externalDependentFeaturesAggregatedBy(other: ModularFeature): Set(Feature) = self.externalDependentFeatures()-> intersection(other.internalFeatures())`
Name	**disjointExternalDependedFeatures()**
Informal definition	Different external features on which a feature depends which belongs to disjoint clusters
Formal definition	*PIMETA::Feature* `disjointExternalDependedFeatures():Set(Feature) = self.externalDependedFeatures()-> reject(f: Feature | self.hasCommonRoot(f))`
Name	**disjointExternalDependentFeatures()**
Informal definition	Different external features depending on a feature which belongs to disjoint clusters
Formal definition	*PIMETA::Feature* `disjointExternalDependentFeatures():Set(Feature) = self.externalDependentFeatures()-> reject(f: Feature | self.hasCommonRoot(f))`

Table C.5: Feature operations (cont.)

Name	**internalDependencies()**
Informal definition	All dependencies which origins and destinations are any of my enclosed features or myself
Formal definition	*PIMETA::ModularFeature* internalDependencies(): Set(Dependency) = allEnclosedFeaturesRecursively()->including(self).dependencyDestinations->asSet->intersection(allEnclosedFeaturesRecursively()->including(self).dependencyOrigins->asSet)
Name	**inboundDependencies()**
Informal definition	All dependencies which origins are any external feature and which destinations are any internal feature or myself
Formal definition	*PIMETA::ModularFeature* inboundDependencies(): Set(Dependency) = allEnclosedFeaturesRecursively()->including(self).dependencyDestinations->asSet->intersection(externalFeatures().dependencyOrigins->asSet)
Name	**outboundDependencies()**
Informal definition	All dependencies which origins are any of my internal features or myself, and which destinations are any of my external features
Formal definition	*PIMETA::ModularFeature* outboundDependencies(): Set(Dependency) = allEnclosedFeaturesRecursively()->including(self).dependencyOrigins->asSet->intersection(externalFeatures().dependencyDestinations->asSet)
Name	**allDependencies()**
Informal definition	All dependencies among all the system features
Formal definition	*PIMETA::ModularFeature* allDependencies(): Set(Dependency) = internalDependencies()->union(inboundDependencies())->union(outboundDependencies())
Name	**internalFeatures()**
Informal definition	All features enclosed by me (recursively)
Formal definition	*PIMETA::ModularFeature* internalFeatures(): Set(Feature) = allEnclosedFeaturesRecursively()

Table C.6: ModularFeature operations (cont.)

Name	**allEnclosedFeatures()**
Informal definition	All features enclosed by me (not recursively)
Formal definition	*PIMETA::ModularFeature* `allEnclosedFeatures(): Set(Feature) = self.feature`
Name	**allEnclosedFeaturesRecursively()**
Informal definition	All features enclosed by me (recursively)

Formal definition	*PIMETA::ModularFeature*
	`allEnclosedFeaturesRecursively(): Set(Feature) =`
	`allEnclosedFeatures()->union(self.allEnclosedFeatures()->`
	`iterate(e: Feature; acc: Set(Feature) =`
	`oclEmpty(Set(Feature)) |`
	`if e.oclIsTypeOf(ModularFeature) then`
	`acc -> union(e.oclAsType(ModularFeature).`
	`allEnclosedFeaturesRecursively())`
	`else`
	`acc -> including(e)`
	`endif))`

Name	**allEnclosedModularFeatures()**
Informal definition	All modular features enclosed by me (not recursively)

Formal definition	*PIMETA::ModularFeature*
	`allEnclosedModularFeatures(): Set(ModularFeature) =`
	`self.feature -> select(oclIsTypeOf(ModularFeature))->`
	`collect(oclAsType(ModularFeature)) -> asSet()`

Name	**allEnclosedModularFeaturesRecursively()**
Informal definition	All modular features enclosed by me (recursively)

Formal definition	*PIMETA::ModularFeature*
	`allEnclosedModularFeaturesRecursively():`
	`Set(ModularFeature) =`
	`self.allEnclosedFeaturesRecursively()->`
	`select(oclIsTypeOf(ModularFeature))->`
	`collect(oclAsType(ModularFeature))->asSet()`

Name	**allDependedFeatures()**
Informal definition	All features on which I or my enclosed features depend

Formal definition	*PIMETA::ModularFeature*
	`allDependedFeatures(): Set(Feature) =`
	`self.dependencyOrigins.destination->`
	`asSet->union(`
	`self.allEnclosedFeaturesRecursively()->`
	`iterate(e: Feature; acc: Set(Feature) =`
	`oclEmpty(Set(Feature)) |`
	`acc ->union(e.allDependedFeatures())))`

Table C.7: ModularFeature operations (cont.)

Name	**allDependentFeatures()**
Informal definition	All features which depend on me or my enclosed features
Formal definition	*PIMETA::ModularFeature* `allDependentFeatures(): Set(Feature) = self.dependencyDestinations.origin-> asSet->union( self.allEnclosedFeaturesRecursively()-> iterate(e: Feature; acc: Set(Feature) = oclEmpty(Set(Feature)) | acc ->union(e.allDependentFeatures())))`
Name	**internalDependedFeatures()**
Informal definition	All my enclosed features on which external features depend
Formal definition	*PIMETA::ModularFeature* `internalDependedFeatures(): Set(Feature) = self.allEnclosedFeaturesRecursively()-> intersection(allEnclosedFeaturesRecursively()-> collect(allDependedFeatures())->asSet()->flatten())`
Name	**internalDependentFeatures()**
Informal definition	All my enclosed features which depend on external features
Formal definition	*PIMETA::ModularFeature* `internalDependentFeatures(): Set(Feature) = self.allEnclosedFeaturesRecursively()-> intersection( allEnclosedFeaturesRecursively()-> collect(allDependentFeatures())->asSet()->flatten())`
Name	**AOPmodules()**
Informal definition	All system's classes, interfaces and aspects
Formal definition	*PIMETA::ModularFeature* `AOPmodules(): Set(ModularFeature) = ModularFeature.allInstances-> select(modularFeatureType.name='Class' or modularFeatureType.name='Aspect' or modularFeatureType.name = 'Interface')`

Table C.8: ModularFeature operations

[This page was intentionally left blank]

Bibliography

[Abr01] Fernando Brito Abreu. Using ocl to formalize object oriented metrics definitions. Technical Report ES007/2001, Software Engineering Group, INESC, May 2001.

[AC94] Fernando Brito Abreu and Rogerio Carapuca. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of 4th Int. Conf. on Software Quality*, 1994.

[AG01] Fernando Brito Abreu and Miguel Goulao. A merit factor driven approach to the modularization of object-oriented systems. *LObject*, 7, nr 4, 2001.

[AT08] AspectJ5-Team. Aspectj 5, quick reference guide. Available at: http:// www.eclipse.org/ aspectj/ doc/ released/ quick5.pdf, 2008.

[BA07] Sérgio Bryton and Fernando Brito e Abreu. Towards paradigm-independent software assessment. In *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology, QUATIC2007*, pages 40–54, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[BA08] Sérgio Bryton and Fernando Brito e Abreu. Modularity-oriented refactoring. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering, CSMR'08*, 2008.

[Bar00] J. Barone. Comparing apples and oranges: a randomised prospective study. *British Medical Journal*, pages 1569–1570, 2000.

[BBA02] A. Baroni, S. Braz, and F. Abreu. Using ocl to formalize object-oriented design metrics definitions. *Lecture Notes in Computer Science: Springer-Verlag*, 2002.

[BBDW99] L.C. Briand, L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *Transactions on Software Engineering*, 25(1):91–121, 1999.

[BCR94] V. Basili, G. Caldiera, and H.D. Rombach. *Encyclopedia of Software Engineering*, chapter The Goal Question Metric Paradigm, pages 528–532. John Wiley & Sons, 1994.

[BEEM95] Lionel Briand, Khaled El-Emam, and Sandro Morasca. Theoretical and empirical validation of software product measures. Technical Report ISERN-95-03, International Software Engineering Research Network, 1995.

[Bor08] Borland. Borland together - visual modeling for software architecture design. http://www.borland.com/us/products/together/index.html, 2008.

[Bra08] Martin Bravenboer. Aspectj-front parser for aspectj. Available at: http://www.program-transformation.org/ Stratego/ AspectJFront, 2008.

[CCH07] Yuangfang Cai, Yuangfang Cai, and Sunny Huynh. An evolution model for software modularity assessment. In Sunny Huynh, editor, *Proc. Fifth International Workshop on Software Quality WoSQ'07: ICSE Workshops 2007*, pages 3–3, 2007.

[CT04] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. First Workshop on Aspect Reverse Engineering (WARE 2004) in conjuction with WCRE 2004, Delft, The Netherlands, 2004.

[DMM05] A. v. Deursen, M. Marin, and L. Moonen. Ajhotdraw: A showcase for refactoring to aspects. Linking Aspect Technology and Evolution Workshop (LATE) at AOSD'2005, Chicago, USA, 2005.

[EKS92] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems, 1992.

[FP97] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. UK: International Thomson Computer Press, London, second edition, 1997.

[GBB06] J.-F. Gelinas, M. Badri, and L. Badri. A cohesion measure for aspects. *Journal of Object Technology*, 5:97–114, 2006.

[GBR07] Martin Gogolla, Fabian Buttner, and Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69:27–34, 2007.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GSF+05] Alessandro Garcia, Claudio Sant'Anna, Eduardo Figueiredo, Uira Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM.

[GVR02] R. L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44:491–506, 2002.

[Hal04] T. A. Halpin. Information modeling and higher-order types. EMMSAD'04 Workshop at CAiSE'04, Riga, Latvia, 2004.

[HK02] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In ACM Press, editor, *Proceedings of OOPSLA'02*, pages 161–173, 2002.

[HM95] Martin Hitz and Behzad Montazeri. Measuring product attributes of object-oriented systems. *Lecture Notes in Computer Science: Springer*, 989:124–136, 1995.

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

[HYYT05] R. Hong Yul Yang; Tempero, E.; Berrigan. Detecting indirect coupling. In *Software Engineering Conference, 2005. Proceedings. 2005 Australian Volume*, pages 212 – 221, March 2005.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[KSG+06] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *Proc. 22nd IEEE International Conference on Software Maintenance ICSM '06*, pages 223–233, 2006.

[Lad03] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[LB06] Cristina Videira Lopes and Sushil Krishna Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. *T. Aspect-Oriented Software Development*, I:1–35, 2006.

[LTBM99] D. J. Leigh, C. J. Theaker, N. Blackwood, and R. Mason. Measuring the impact of migration to an object oriented paradigm. Workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE'1999), Lisbon, Portugal, 1999.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[MRC06] L. Moonen, G. Ramalingam, and S. Clarke. Aspects for legacy applications. Dagstuhl Seminar 06302, International Conference and Research Center (IBFI), Schloss Dagstuhl, Germany, July 2006.

[Mye78] G. Myers. *Composite Structured Design*. Van Nostrand, 1978.

[NRD+07] Alberto Neto, Marcio de Medeiros Ribeiro, Marcos Dosea, Rodrigo Bonifacio, and Paulo Borba. Semantic dependencies and modularity of aspect-oriented software. In *Proceedings of the 1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM.07)*, 2007.

[OMG03a] OMG. Uml 2.0 infrastructure. Object Management Group, Inc., Final Adopted Specification ptc/03-09-15, September 2003.

[OMG03b] OMG. Uml 2.0 ocl. Object Management Group Inc., Final adopted specification ptc/03-10-14, October 2003.

[OMG03c] OMG. Uml 2.0 superstructure. Object Management Group Inc., Final Adopted Specification ptc/03-08-02, August 2003.

[PHP08] PHPA. phpaspect. http://phpaspect.org/documentation/aop.html, 2008.

[Pol57] G. Polya. *How to solve it*. Princeton University Press, 1957.

[Pre00] Pressman. *Software Engineering*. Mcgraw-Hill, 2000.

[RG] Mark Richters and Martin Gogolla. Use, a uml-based specification environment. Available at: http:// www.db.informatik.uni-bremen.de/ projects/ USE/.

[RGV04] V. Ramesh, R. L. Glass, and I. Vessey. Research in computer science: an empirical study. *Information and Software Technology*, 70:165–176, 2004.

[San03] Claudio SantAnna. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proc. Brazilian Symp. on Software Engineering*, pages 19–34, 2003.

[SDM06] SDMetrics. Sdmetrics. the software design metrics tool for the uml. http://www.sdmetrics.com/index.html, 2006.

[SGK+04] C. SantAnna, A. Garcia, U. Kulesza, C. Lucena, and A. Staa. Design patterns as aspects: A quantitative assessment. In *Proceedings of the XVIII Simp. Bras. de Eng. Software, Brasilia, Brasil*, Outubro 2004.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal 13 nr. 2*, pages 115–39, 1974.

[TLPH95] W. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal for Systems and Software*, 28:9–18, 1995.

[ZH03] A. A. Zakaria and H. Hosny. Metrics for aspect-oriented software design. Workshop on Aspect-Oriented Modeling with UML at AOSD'2003, Boston, USA, 2003.

[Zha04] J. Zhao. Measuring coupling in aspect-oriented systems. 10th International Software Metrics Symposium, METRICS'04, 2004.

[ZW97] M. V. Zelkowitz and D. Wallace. Experimental validation in software engineering. *Journal of Information and Software Technology*, 39:735–743, 1997.

[ZX04] J. Zhao and B. Xu. Measuring aspect cohesion. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'2004)*, 2004.

[This page was intentionally left blank]