Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

CONTRACT MODULARITY IN DESIGN BY CONTRACT LANGUAGES

Henrique Emanuel Mostaert Rebêlo

TESE DE DOUTORADO

Recife
Março de 2014

Universidade Federal de Pernambuco
Centro de Informática

Henrique Emanuel Mostaert Rebêlo

CONTRACT MODULARITY IN DESIGN BY CONTRACT LANGUAGES

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: *Ricardo Massa Ferreira Lima*
Co-orientador: *Gary T. Leavens*

Recife
Março de 2014

Catalogação na fonte
Bibliotecária Joana D’Arc Leão Salvador, CRB 4-572

Rebêlo, Henrique Emanuel Mostaert.
Contract modularity in design by contract languages../ Henrique Emanuel Mostaert Rebêlo. – Recife: O Autor, 2014.
135 f.: fig., tab.

Orientador: Ricardo Massa Ferreira Lima.
Tese (Doutorado) - Universidade Federal de Pernambuco. Cin. Ciência da Computação , 2014.
Inclui referências e apêndices.

1. Engenharia de software. 2. Software - refatoração. 3. Software - reutilização. I. Lima, Ricardo Massa Ferreira. (orientador). II. Título.

005.1 (22. ed.) MEI 2014-79

Tese de Doutorado apresentada por Henrique Emanuel Mostaert Rebêlo à Pós Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "Contract Modularity in Design By Contract Languages" orientada pelo Prof. Ricardo Massa Ferreira Lima e aprovada pela Banca Examinadora formada pelos professores:

Visto e permitida a impressão.
Recife, 11 de março de 2014.

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Aos meus pais e avós.

Acknowledgments

I wish to thank my supervisor, Professor Ricardo Massa, for his great guidance, careful supervision, encouragement, friendship, and supporting me since undergraduate.

I wish to thank Professor Gary T. Leavens for his co-supervision, encouragement, discussions, and refined perception. Most of this work was inspired by his principles and ideas. In fact, he worked as a supervisor. Also, I am grateful to him for hosting me in his home during several visits I made at UCF during my PhD.

I gratefully acknowledge the Professors of the committee, Prof. Augusto Sampaio, Prof. Rohit Gheyi, Prof. Heron Carvalho, Prof. André Santos, and Prof. Fernando Castor, for making important suggestions on how to improve this thesis.

Special thanks to Professor Paulo Borba for listening and making comments, suggestions that helped me to conduct the work described in this thesis.

I am grateful to Shuvendu K. Lahiri who provided me the wonderful experience to conduct a work at Microsoft Research during some months. Among other things, he contributed a lot to this thesis with fruitful discussions. Thanks for spending some time playing ping-pong with me. It was so hard to win a game against him.

Rustan Leino, Tom Ball, Mike Barnett, Thomas Zimmerman, and Manuel Fahndrich, also from Microsoft Research, deserve my gratitude for listening and discussing several aspects of design by contract languages.

I am deeply grateful to Professor Mira Mezini and Professor Alessandro Garcia for the discussions and ideas both gave me during AOSD 2011. Those ideas were very crucial to what is discussed and presented in this thesis.

I wish to thank Professor Hridesh Rajan, Mehdi Bagherzadeh, Thomas Thüm for helpful discussion about modularity, aspect-oriented programming, and modular reasoning.

Special thanks to Professor Ana Lúcia Cavalcanti and Professor Marcel Oliveira for several discussions we had about the design of AspectJML and for using it in their undergraduate and graduate courses. Most of the crucial bugs and some options of the AspectJML compiler, ajmlc, were improved due their feedback. The use of AspectJML in practice is stable thanks for their help.

I wish to thank Professor Rohit Gheyi and Gustavo Soares to run their SafeRefactor tool in some of the refactored systems, with AspectJML, we use in this work.

Thanks to Professors Paulo Borba, Shuvendu K. Lahiri, Chris Hawblitzel, Hridesh Rajan, Yuanfang Cai, Roberta Coelho, Uirá Kulesza, Cláudio Sant'Anna, Alexandre Mota, Marieke, Huisman, Márcio Cornélio, Fernando Castor, and others that I had the good fortune to collaborate with during my PhD.

Special thanks to my grandfather Walter Mostaert, my mother Elizabeth Mostaert, and my grandmother Tereza Mostaert. They supported me in all the ways they could and I needed.

I am deeply grateful to my girlfriend Natália Prado for her encouragement, patience, and love during this work. Thanks for being part of all of this. I promise to dedicate more time to you hereafter. Also, my mother-in-law Socorro Prado and father-in-law Josoé Matias both played important role to make me relax at some crucial moments in this PhD journey.

I wish to thank my friends that I made during the PhD process: Márcio Ribeiro, Leopoldo Teixeira, Rodrigo Andrade, Lais Neves, Paola Accioly, Rodrigo Bonifácio, and

Alberto Costa Neto. Also, thanks to César Lins my friend since the time of the master program at UPE.

I am very glad for studying at Centro de Informática da Universidade Federal de Pernambuco (CIn/UFPE), an excellence in post-graduation in Brazil. Special Thanks for all the UFPE technical staff, specially, Inácia, Socorro, Lilia, and Leila.

The work described in this dissertation was financially supported by FACEPE (Brazilian Agency for Postgraduate Education).

Resumo

Design by Contract (DbC) é uma técnica popular para desenvolvimento de programas usando especificações comportamentais. Neste contexto, pesquisadores descobriram que a implementao de DbC é crosscutting e, portanto, sua implementação é melhor modularizada por meio da Programação Orientada a Aspectos (POA) porém, os mecanismos de POA para dar suporte a modularide de contratos, de fato comprometem sua modularidade e entendidmento. Por exemplo, na linguagem POA AspectJ, o raciocínio da corretude de uma chamada de método requer uma análise global do programa para determinar quais advice aplicam e sobretudo o que esses advice fazem em relação a implementação e checagem DbC. Além disso, quando os contratos so separados das classes o programador corre o risco de quebrar-los inadvertidamente.

Diferentemente de uma linguagem POA como AspectJ, uma linguagem DbC preserva as principais caractersticas DbC como raciocnio modular e documentação. No entanto, pré- e pós-condições recorrentes continuam espalhadas por todo o sistema. Infelizmente esse não o único problema relacionado com modularidade que temos em linguagens DbC existentes, o seu com respectivos verificadores dinâmicos so inconsistentes com as regras de information hiding devido a naturaze overly-dynamic na qual os contratos são checados no lado servidor. Este problema implica que durante a reportagem de erros, detalhes de implementação so expostos para clientes no privilegiados. Portanto, se os programadores cuidadosamente escolherem as partes que devem ser escondidas dos clientes, durante a checagem dinâmica de contratos, as mudanas nessas partes não deveriam afetar nem os clientes dos módulos nem a reportagem de erros de contratos.

Neste trabalho nós resolvemos esses problemas com AspectJML, uma nova liguagem de especificação que suporta contratos crosscutting para código Java. Além disso, nós demonstramos como AspectJML usa as principais caractersticas de uma linguagem DbC como raciocínio modular e documentação dos contratos. Mais ainda, nós mostramos como AspectJML combinado com nossa técnica chamada de client-aware checking permite uma checagem dinâmica de contratos que respeitem os princípios de information hiding em especificações. Neste trabalho usamos JML para fins concretos, mas nossa solução pode ser utilizadas para outras linguagems Java-likee suas respectivas linguagens DbC.

Para concluir, nós conduzimos uma avaliação da nossa modularização dos contratos crosscutting usando AspectJML, onde observamos que seu uso reduz o esforo de escrever pré- e pós-condies, porém com um pequeno overhead em tempo de compilação e instrumentação de código para checagem de contratos.

Palavras-Chave: Design by contract, aspect-oriented programming, crosscutting contracts, JML, AspectJ, AspectJML

Abstract

Design by Contract (DbC) is a popular technique for developing programs using behavioral specifications. In this context, researchers have found that the realization of DbC is crosscutting and fares better when modularized by Aspect-Oriented Programming. However, previous efforts aimed at supporting crosscutting contracts modularly actually compromised the main DbC principles. For example, in AspectJ-style, reasoning about the correctness of a method call may require a whole-program analysis to determine which advice applies and what that advice does relative to DbC implementation and checking. Also, when contracts are separated from classes a programmer may not know about them and may break them inadvertently.

Unlike an AspectJ-like language, a DbC language keeps the main DbC principles such as modular reasoning and documentation. However, a recurrent pre- or postcondition specification remains scattered across several methods in many types. Unfortunately, this is not the only modularity problem we have with existing DbC languages. Such languages along with their respective runtime assertion checkers are inconsistent with information hiding rules because they check specifications in an overly-dynamic manner on the supplier side. This implies that during error reporting, hidden implementation details are exposed to non-privileged clients. Such details should not be used in a client's correctness proof, since otherwise the proof would be invalidated when they change. Therefore, if programmers have carefully chosen to hide those parts "most likely" to change, most changes, in the hidden implementation details, do not affect either module clients nor DbC error reporting.

In this work we solve these problems with *AspectJML*, a new specification language that supports crosscutting contracts for Java code. We also show how AspectJML supports the main DbC principles of modular reasoning and contracts as documentation. Additionally, we explain how AspectJML combined with our client-aware checking technique allows runtime checking to use the privacy information in specifications, which promotes information hiding. We use JML for concreteness, but the solution we propose can also be used for other Java-like languages and their respective DbC languages.

To conclude, we conduct an evaluation to assess the crosscutting contract modularization using AspectJML, where we observe that its use reduces the overall design by contract code, including pre- and postconditions, but introduces a small overhead during compile time and can increase the resulting bytecode due to code instrumentation to check ordinary and crosscutting contracts.

Keywords: Design by contract, aspect-oriented programming, crosscutting contracts, JML, AspectJ, AspectJML

List of Figures

2.1 The Java implementation of the package classes with courier package delivery [MMM02]. 18
2.2 An overview of the JML environment. 20
2.3 An example of the application of the template Desugaring 1 for method setSize. 31
2.4 The JML pre- and postconditions for Package class of Figure 2.1. 49
2.5 Ordering the execution of advice and join points. The darker areas represent the higher-precedence [Lad03]. 58
2.6 The mapping between the traditional and @AspectJ syntax [Lad09]. . . . 61
2.7 The AspectJ implementation for the JML crosscutting contracts presented in Figure 2.4. 62

3.1 The tracing crosscutting concern implementation in AspectJ. 65
3.2 Crosscutting contracts in Package's specifications (see Figure 2.4). Scenario (1) illustrates the crosscutting preconditions, (2) illustrates the crosscutting normal postconditions, and (3) the crosscutting exceptional postconditions. 66

4.1 The crosscutting contract specifications with AspectJML used so far for the type Package related to the crosscutting scenarios illustrated in Figure 3.2. 86
4.2 setSize's client code. 87
4.3 Generated before advice to check the crosscutting preconditions of Package in Figure 4.1. 92
4.4 Generated before advice to check the protected preconditions of the method setWeight in type Package. 92
4.5 The crosscutting contract structure in the Package class using AspectJML/AJDT [KM05]. 94
4.6 An example of a malformed pointcut declaration in AspectJML. 96

5.1 Health Watcher's Design including design by contract concern. 98

List of Tables

2.1 Examples of method signatures that can be used in pointcut designators. 51
2.2 Examples of other useful AspectJ pointcut designators. 52

3.1 Summary of the analysis of modularity for DbC and AOP. 70

5.1 GQM 100
5.2 LOC, DbCLOC, NOPre, NOPo, NOO, and VS metric results for all systems. 101
5.3 ISC, IBC, and CT metric results for all systems. 103
5.4 CT metric results for dbviz, Rhino, and iBATIS systems. 103

Contents

1 Introduction **13**

1.1 Problem Overview . . . 13

1.2 Solution . . . 14

1.3 Organization . . . 15

2 Background **16**

2.1 Definitions . . . 16

2.1.1 Modular Reasoning . . . 16

2.1.2 Crosscutting Structure . . . 17

2.2 A Running Example . . . 17

2.3 Design by Contract . . . 18

2.3.1 Design by Contract with JML and AspectJ . . . 19

2.4 An Overview of JML . . . 20

2.4.1 Behavioral Interface Specification . . . 20

2.4.2 Annotations . . . 20

2.4.3 Assertions and Expressions . . . 21

2.4.4 Null is not the default . . . 22

2.4.5 Method Specifications . . . 23

2.4.6 Method Specification Cases . . . 28

2.4.7 Type Specifications . . . 33

2.4.8 Specification Inheritance . . . 34

2.4.9 Privacy Modifiers and Visibility in Specifications . . . 37

2.4.10 Modularity notes in JML when applying Design By Contract . . . 41

2.4.11 Tool Support . . . 47

2.4.12 Exported JML example . . . 48

2.5 An Overview of AspectJ . . . 48

2.5.1 The Anatomy of an aspect . . . 48

2.5.2 The Join Point Model . . . 49

2.5.3 Pointcut Designators . . . 50

2.5.4 Advice . . . 53

2.5.5 Accessing Join Point Context via Reflection . . . 58

2.5.6 Static Crosscutting . . . 59

2.5.7 Privileged aspects . . . 60

2.5.8 @AspectJ . . . 60

2.5.9 Exported AspectJ example . . . 61

2.5.10 AspectJ Compilers/Weavers . . . 61

2.6 Chapter Summary . . . 62

3 Design by Contract and Modularity Problems **64**
3.1 The Modular Reasoning Criterion . . . 64
3.2 The Documentation Criterion . . . 65
3.3 The Crosscutting Contract Specification Criterion . . . 66
3.4 The Information Hiding and RAC Criterion . . . 67
3.5 Chapter Summary . . . 70

4 The AspectJML Language **71**
4.1 Overview . . . 71
4.2 Design Decisions . . . 71
4.2.1 AspectJML is DbC + Quantification . . . 71
4.2.2 @AspectJ . . . 72
4.2.3 AspectJML Supported Features . . . 72
4.2.4 AspectJML Compatibility . . . 72
4.3 XCS in Action . . . 73
4.3.1 XCS Syntax . . . 73
4.3.2 Specifying Crosscutting Preconditions . . . 74
4.3.3 Specifying Crosscutting Postconditions . . . 74
4.3.4 Multiple Specifications Per Pointcut . . . 75
4.3.5 Pointcut Expressions Without Type Signature Patterns . . . 75
4.3.6 Reusing Pointcuts . . . 76
4.3.7 Specification of Unrelated Types . . . 76
4.3.8 Separate Files for Crosscutting Contracts . . . 80
4.3.9 XCS and Multiple Specification Cases . . . 80
4.3.10 More XCS Examples . . . 82
4.3.11 Exported AspectJML/XCS Examples . . . 85
4.4 AspectJML's Benefits . . . 85
4.5 Enforcing Information Hiding in AspectJML with Client-Aware Checking 88
4.6 Implementation . . . 91
4.6.1 Compilation strategy . . . 91
4.6.2 Contract violation example in AspectJML . . . 94
4.7 Tool Support . . . 94
4.8 Chapter Summary . . . 96

5 Evaluation **97**
5.1 Target Systems . . . 97
5.2 Study Phases . . . 97
5.3 Quantitative analysis . . . 99
5.3.1 System size results . . . 99
5.3.2 Compilation System results . . . 100
5.3.3 What about Scattering and Tangling metrics? . . . 102
5.4 Representative Crosscutting Scenarios . . . 104
5.4.1 Understanding the Crosscutting Contract Structure . . . 104
5.4.2 Modularizing Crosscutting Contracts in HW . . . 106
5.4.3 Reasoning About Change . . . 107

5.4.4 More Crosscutting Scenarios and their Modularization 107
5.4.5 Study Constraints 110
5.5 Chapter Summary 111

6 Related Work 112
6.1 Empirical Evidence About Crosscutting Contracts 112
6.2 Crosscutting Contract Modularization 113
6.3 Crosscutting Modularity and Modular Reasoning with Design Rule Languages 115
6.4 Design by Contract Languages and Generative Programming 116
6.5 Design by Contract Languages and Information Hiding 117
6.6 Design by Contract Languages and Client-Side Checking 118

7 Conclusions 119
7.1 Review of the Contributions 120
7.2 Future Work 120

A AspectJML Grammar Summary 133
A.1 JML-based Reserved Words 133
A.2 @AspectJ-based Reserved Words 133
A.3 Method Specfication 133
A.4 Type Specfication 134
A.5 Predicates and Specification Expressions 134
A.6 Pointcut Expressions 134

Chapter 1

Introduction

Software engineering provides a set of techniques and tools to reduce software costs and improve software correctness [Som01]. Unfortunately, producing error free software artifacts is still a difficult challenge. Problems caused by such defects can vary from simple daily annoyances to the more significant losses of money or even lives. Hence, after software delivery, a great deal of effort is required to fix problems (bugs). This task is know as *corrective maintenance* [Som01]. According to Meyer [Mey00], maintenance is estimated as 70% of a software cost as a whole.

Design by Contract (DbC), originally conceived by Meyer [Mey92a], is a useful technique for developing and improving functional software correctness. It can significantly contribute to reduce the corrective maintenance effort. The key mechanism in DbC is the use of behavioral specifications called "contracts". Checking these contracts against the actual code at runtime has a long tradition in the research community [Mey92b, CR06, Ros95, LBR06, FBL10, TSK+12, RSL+08, BLS03, LTBJ06].

The contracts usually define pre- and postconditions that should be satisfied before and after executing a given piece of code. For instance, a client class must satisfy the preconditions defined in the supplier (declaring) class before invoking its exported routines [Mey92a, Mey92b]. Likewise, the supplier class must satisfy the postconditions, which constitute the supplier class's obligations. However, the runtime checking mechanism reports an error whenever the client class or the supplier class violates the contract. This idea of checking contracts at runtime was popularized by Eiffel [Mey92b] in the beginning of 1990s. In addition to Eiffel, other DbC languages include the Java Modeling Language (JML) [LBR06], Spec# [BLS05], Code Contracts [FBL10], among others.

1.1 Problem Overview

Although contracts can be seen as an important mechanism to specify and document software artifacts, it is claimed in the literature [KHH+01, LL00, LLH02, SL04, MMvD05, BDL05, LKR05, F+06, RLL11, RLK+13] that the contracts of a system are de-facto a crosscutting concern and fare better when modularized with aspect-oriented programming [KLM+97] (AOP) mechanisms such as pointcuts and advice [KHH+01]. The idea has also been patented [LLH02]. However, Balzer, Eugster, and Meyer's study [BEM05] contradicts this intuition by concluding that the use of aspects hinders design by contract specification and fails to achieve the main DbC principles such as documentation and modular reasoning. Indeed, they go further to say that "*no module in a system (e.g., class or aspect) can be oblivious of the presence of contracts*" [BEM05, Section 6.3]. According to them, contracts should appear in the modules themselves and separating such contracts as aspects contradicts this view [Mey92a].

However, plain DbC languages like Eiffel [Mey92b] or JML [LBR06] also have problems when dealing with crosscutting contracts. Although mechanisms such as invariant declarations help avoiding scattering of specifications, the basic pre- and postcondition specification mechanisms do not prevent scattering of crosscutting contracts. For example, there is no way in Eiffel or JML to write a single pre- and postcondition and apply it to several methods of a particular type. Instead, such pre- and postconditions are widely repeated and scattered specification fragments among those methods.

It is clear that we face a dilemma with respect to crosscutting contracts. If we use an AOP language, like AspectJ [KHH$^+$01], to modularize them, the result is a poor contract documentation and compromised modular reasoning. If we go back to a design by contract language such as JML, we face the scattered nature of recurrent contracts that appear throughout the system. This dilemma leads us to the following research question: Is it possible to have the best of both worlds? That is, can we achieve good documentation and modular reasoning while also specifying crosscutting contracts in a modular way?

Another modularity problem with existing DbC languages is related to Parnas [Par72] information hiding. During runtime assertion checking, existing DbC languages expose hidden implementation details in error reporting. The main problem is that such details cannot be fully understood by all clients.

1.2 Solution

To cope with these modularity problems this work proposes *AspectJML*, a simple and practical aspect-oriented extension to JML. It supports the specification of crosscutting contracts for Java code in a modular way while keeping the benefits of a DbC language, like documentation and modular reasoning. Moreover, runtime assertion checking in AspectJML ensures that no implementation details are exposed to non-privileged clients during error reporting. This maintains the benefits of information hiding. We build our proposal in JML for concreteness, but the ideas we propose can also be applied to Eiffel, Spec#, or Code Contracts.

To better evaluate our AspectJML language, we also conduct an empirical study where we refactor existing target systems to modularize crosscutting contracts and compare each one with its corresponding non-AspectJML version (i.e. JML version). Our hypothesis is that, with AspectJML, programmers are able to modularize crosscutting contracts while preserving the main DbC benefits such as documentation and modular reasoning. Also, contract modularization with AspectJML leads to a reduced design by contract lines of code in general (e.g., the number of preconditions). The results suggest that our hypothesis might be true, but it also indicates that we can have a small overhead while using AspectJML. This overhead is in terms of compilation time and code instrumentation (e.g., bytecode size).

1.3 Organization

The remainder of this work is organized as follows:

- Chapter 2 gives some definitions and reviews essential concepts used by this work;
- Chapter 3 discusses the modularity problems we address in this thesis, with existing DbC and AOP languages when dealing with crosscutting contracts. In particular, we discuss modular reasoning, crosscutting modularity, and information hiding about these languages;
- Chapter 4 presents our AspectJML specification language, that defines how to specify crosscutting contracts in a modular way and how error reporting uses information hiding properties in specifications to avoid exposing hidden implementation details to non-privileged clients;
- In Chapter 5, we evaluate the expressiveness of AspectJML in a empirical study. We discuss and show the impact of crosscutting contract modularization using AspectJML through a set of metrics. Also, we illustrate real scenarios, extracted from the studied systems, on how the AspectJML features are used in practice.
- Chapter 6 discusses some related work;
- Chapter 7 presents our final considerations and future work.

Chapter 2

Background

In this chapter we introduce some essential concepts related to this work. Firstly, we provide some definitions of important terms related to this work. Afterwards, we introduce the running example that will be used throughout this work in Section 2.2. Also, we discuss the main DbC features through the DbC language JML (Section 2.3), and through the aspect-oriented language AspectJ (see Section 2.3). Both languages are the focus of this work.

2.1 Definitions

2.1.1 Modular Reasoning

The criteria we use in this work for considering a modular design constraint (i.e., modular contracts) is based on the ideas of Parnas [Par72]. That work argues that the modules into which a system is decomposed should be chosen to provide benefits in three areas. Parnas writes (p. 1054):

> *"The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility— it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility— it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood".*

In relation to these benefits, we begin by defining a notion of *modular reasoning* corresponding to Parna's third benefit.

Definition 2.1.1. Modular reasoning *means being able to make decisions about a module while looking only at the module's own specifications, its own implementation, and the interface specifications (or implementations) of modules that its own implementation references.* □

Not all decisions are amendable to modular reasoning. Sometimes, programmers need extra steps using the modularity mechanisms. We define this extra steps as *expanded modular reasoning.*

Definition 2.1.2. Expanded modular reasoning *means also consulting the specifications or implementations of referenced modules (i.e., supertypes).* □

Based on these definitions, a language supports modular reasoning if the actions of a module say M written in that language can be understood based solely on the code contained in M along with the specifications (if any) of the modules referred to by M. On the other hand, if a programmer, to reason about a module M, also needs to examine all the modules in the system, we say that the programmer needs a non-modular reasoning approach, which we can call as *global reasoning*.

Definition 2.1.3. Global reasoning *means having to provide a whole-program reasoning (all modules examination) in the system.* □

2.1.2 Crosscutting Structure

For more than 30 years, the concept of modularity has been seen as a key enabler for the development of large software systems. Such modularity can be achieved by using a fundamental principle in software design, which is to separate concerns [Par72]. According to this principle, we should decompose a program in such a way that each one of the resulting modules implements one concern.

In the history of programming languages and software engineering, several mechanisms for modularization of concerns have been developed, such as procedures, modules, and classes. These mechanisms enable one to use the form of a block structure or a hierarchical structure as the primary modularization mechanism. In the middle to the end of 1990s, however, emerged the problem of *crosscutting concerns*. Certain concerns are said to be crosscutting when their separation using traditional mechanisms based on block or hierarchical structure seems to be very difficult and challenging [KLM$^+$97].

Definition 2.1.4. *Crosscutting is a structural relationship between the representations of two concerns. It is an alternative to hierarchical and block structure [ABKS13].* □

Classic programming languages suffer from a limitation that is referred to as the *tyranny of the dominant decomposition*, which is the cause of crosscutting [TOHS99]. Considering hierarchical structure, a program can be decomposed in only one way (along one dimension) at a time. This is called dominant decomposition. All concerns that do not align with the dominant decomposition end up in *scattered* and *tangled* code.

Definition 2.1.5. Code scattering *refers to a concern representation that is scattered across representations of multiple other concerns [ABKS13].* □

Definition 2.1.6. Code tangling *refers to the intermingled representation of several concerns within a module [ABKS13].* □

2.2 A Running Example

This section introduces the example that will be used throughout the work.

Figure 2.1 illustrates a simple delivery service system, written in Java, which manages package delivery within a city (destination) [MMM02, pp.100-107]. The example involves a simple set of package classes, including Package, GiftPackage, and DiscountedPackage. Also the Courier class that is responsible for providing the delivery service implementation for the three kinds of packages mentioned.

```
public class Package {
 double width, height, weight;

 void setSize(double width, double height) {
  this.width = width;
  this.height = height;
 }
 void reSize(double width, double height) {
  if((this.width != width) && (this.height != height)){
   this.width = width;
   this.height = height;
  }
 }
 boolean containsSize(double width, double height) {
  if(this.width == width  && this.height == height)
      return true;
  else return false;
 }
 double getSize(){ return this.width * this.height; }

 void setWeight(double weight) { this.weight = weight; }
 ... // other methods
}

public class GiftPackage extends Package{
 void setSize(double width, double height) {...}
 void setWeight(double weight) {...}
}

public class DiscountedPackage extends Package{
 void setSize(double width, double height) {...}
 void setWeight(double weight) {...}
}

public class Courier {
 void deliver(Package p, String destination) {...}
}
```

Figure 2.1: The Java implementation of the package classes with courier package delivery [MMM02].

2.3 Design by Contract

In the in the late 1980's, Bertrand Meyer applied the concept of pre- and postconditions to object-oriented programming and conceived the term Design by Contract (DbC) [Mey92a, Mey00]. To this end, Meyer provided fully built-in support for DbC in his Eiffel programming language [Mey92b]. The main goal was to define a methodology of software construction based on precisely defined contracts that can be checked during runtime [Mey92a, Mey00]. Indeed, checking contracts against the actual code at runtime has a long tradition in the research community [CR06, Ros95, LBR06, FBL10, TSK$^+$12, BLS03, LTBJ06].

A contract constrains the relationship between a supplier class and its clients. These constraints are expressed by method preconditions, method postconditions and object/class invariants. According to Meyer [Mey92a, Mey00], they are defined as follows:

- a **precondition** is an assertion attached to a routine that must be guaranteed by every client prior to calling the routine;
- a **postcondition** is an assertion attached to a routine that must be guaranteed

by the routine's body on return from a call to the routine **iff** its precondition was satisfied on entry.

- an object/class **invariant** is an assertion that must be both satisfied on creation of every instance of a class and preserved by every exported routine of the class; this ensures that the class invariant will hold by all instances of the class whenever they are externally observable.

The benefits of adding contracts to source code include the following [Ver03]:

- more precise description of what the code should do;
- efficient discovery and correction of bugs;
- early discovery of incorrect client usage of classes;
- reduced chance of introducing bugs as the application evolves;
- precise documentation that is always in accordance with application code.

In relation to drawbacks, one recurrent criticism is that some approaches are quite abstract, require a completely new notation to learn or require heavily mathematical notation to be used. Fortunately, DbC languages such as Eiffel [Mey92b], JML [LPC+08, LBR06], or Code Contracts [FBL10] overcome these limitations by using syntax based on the specified programming language. For instance, JML's syntax is similar to that of Java, and its annotations are given in specially formatted comments. Based on that, Leavens, Baker, and Ruby state that JML is easier to use than VDM or Larch. JML was designed by Java developers having only modest mathematical training [LBR99].

2.3.1 Design by Contract with JML and AspectJ

There are numerous mechanisms to specify and check contracts during runtime, such as assertions, executable specifications, pointcuts and advice. Assertions are those simple statements that are placed in an inline fashion within code. Java supports assertion statements. However, in Java, there is no other built-in and expressive way to write contracts in a DbC style. In this context, the JML [LPC+08, LBR06] language was conceived to deal with contracts as Eiffel does. Besides being used for documentation, the JML specifications are executable. In other words, there is a compiler that translate them to runtime checks useful to help ensuring software correctness.

Finally, the later two mechanisms are well-known aspect-oriented programming features [KLM+97] that are used to modularize crosscutting concerns and also used to express DbC contracts. Indeed, according to the literature, these AOP mechanisms fare better in achieving contract modularization [KHH+01, LL00, LLH02, SL04, MMvD05, BDL05, LKR05, F+06, RLL11, RLK+13].

In the following sections, we discuss in more detail two forms to implement/express design by contract at source code level. The first one is the DbC language JML [LPC+08, LBR06], and the second one is the AOP language AspectJ [KHH+01, Lad03]. Both are the main focus of this work.

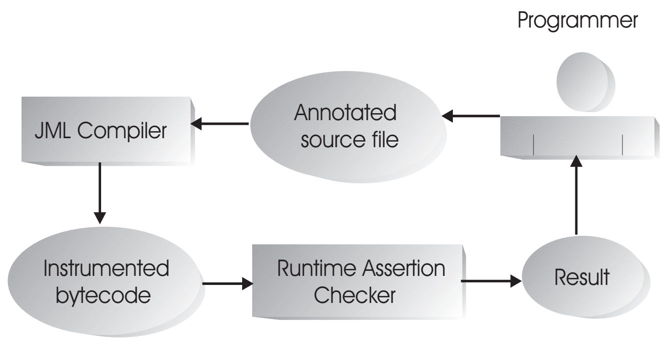


Figure 2.2: An overview of the JML environment.

2.4 An Overview of JML

This section gives an overview of the Java Modeling Language (JML), introducing its major features such as method and type specifications. The presentation is informal and running-example-based.

2.4.1 Behavioral Interface Specification

JML [LPC+08, LBR06], which stands for "Java Modeling Language", is a *behavioral interface specification language* (BISL) tailored to Java. JML combines the design by contract (DbC) technique [Mey92a, Mey00] of Eiffel [Mey92b] with the model-based specification approach typified by VDM [Jon90], Z [Spi89], and Larch/C++ [CL94]. It also adds some elements from the refinement calculus [BvW98]. As in Eiffel, JML uses Java's expression syntax in assertions. As mentioned earlier, this makes JML's notation easier for programmers to learn than notations based on an independent specification language, such as Z [Spi89] or OCL [WK99].

As a BISL, JML is used to specify Java modules (classes and interfaces). Concerning Java modules, JML takes into account two issues during the specification process:

- syntactic interface — consists of the names and static information (e.g., method names, modifiers, arguments, return type) found in Java declarations;
- functional behavior — describes how the module works when used.

Therefore, BISLs are languages that describe interface details for clients. For example, Larch/C++ [CL94] describes how to use a module in a C++ program, just as JML specifies how to use a module in a Java program.

2.4.2 Annotations

JML specifications are written in special *annotation comments*, which start with an `@` sign, that is, comments in the form: `//@` <JML specification> or `/*@` <JML specification> `@*/`. These annotations work as simple comments for a Java compiler, whereas they are interpreted as specifications by the JML compiler [Che03, Reb08].

It is important to note that the at-sign (`@`) must be right next to the start of comment characters. A comment starting with `// @` will be ignored by JML. In other words, such a comment is not processed as a specification by the JML compiler. This happens because JML tools do not currently warn the programmer about comments that use such mistaken annotation markers.

Figure 2.2 depicts an overview of the JML environment. A programmer includes annotations in the Java source file in the form of comments. Then, the JML compiler translates the annotated Java source file into instrumented bytecode that check whether the Java program respects the specification.

2.4.3 Assertions and Expressions

Assertions and expressions of specifications in JML are written using Java's expression syntax. However, they must be pure. This means that side-effects cannot appear in JML assertions or expressions [LCC+05]. But Java assertions and expressions do allow side-effects. Regarding the prevention of side-effects, the following Java operators are not allowed within JML specifications:

- assignment — assignment operators (such as `=`, `+=`, `-=`) are not allowed;
- increment and decrement operators — all forms of increment and decrement operators (`++` and `--`) are not allowed.

In addition, only pure methods can be used in JML expressions and assertions — a method is pure if it does not have any side-effects on the program state. In other words, the method does not modify the state (e.g., by assigning any fields of objects). The pureness of methods is expressed by using the JML modifier **pure** when declaring a method.

```
/*@ pure @*/ boolean containsSize (double width, double height){...}
```

This method declaration denotes a method with no side-effects; it is not allowed to modify the program state.

Besides the **pure** annotation, JML provides a rich set of constructs, some of which make extensions to the Java's expression syntax to provide more expressive power in JML specification; they can be used in JML assertions and expressions. For example, `\old`(E) represents the pre-state value of expression E. A pre-state value refers to the value before method execution. The `\result` construct specifies the return value of a method. Note that in JML assertions, such constructs start with a backslash (\) in order to avoid interfering with identifiers present in a user program. JML also provides the use of logical connectives such as *conjuction* (`&&`), *disjunction* (`||`), *negation* (`!`), *forward* (`==>`) and *reverse implications* (`<==`), *equivalence* (`<==>`), and *inequivalence* (`<=!=>`). Regarding quantifiers, JML supports several kinds such as *universal quantifier* (`\forall`), *existential quantifier* (`\exists`), and *generalized quantifiers* (`\sum`, `\product`, `\min`, and `\max`). The quantifiers `\sum`, `\product`, `\min`, and `\max` are generalized quantifiers that return respectively the sum, product, minimum, maximum of the values present in JML expressions. For example, an expression (`\sum int x; 1 <= x && x <= 5; x`) denotes the sum of values from 1 to 5.

Another feature provided by JML is that one can use *informal descriptions* when specifying a Java module. Informal descriptions are useful for producing an informal documentation of the Java code. JML also allows informal descriptions when specifying a Java module.

```
(* some text describing a boolean-valued predicate *)
```

Since informal descriptions are not-executable, they are ignored by the JML compiler.

Expression Evaluation

Since JML is based on standard Java expressions, it should be aware of exceptions that can arise during evaluation. In JML, an assertion is considered to be valid if and only if its evaluation: (1) does not cause an exception to be thrown, and (2) yields the value **true**. Similarly, an assertion is taken to be invalid iff its evaluation: (1) does not cause an exception to be thrown, and (2) yields the value **false**. An assertion to be considered either valid or invalid is the standard two-valued logic expected in general. However, as mentioned, Java provides a third one; when expressions become invalid due to an exception thrown. Hence, JML handles a three-valued logic during assertion evaluation.

This three-valued logic semantics adopted by JML is discussed in detail by Chalin [Cha07]. Among other things, he discusses that when an assertion becomes invalid by and exception, the runtime assertion checking stops immediately to signal an invalid expression evaluation. For instance, in JML, we have the JMLEvaluationError thrown to signal such invalid assertions during runtime checking.

In-line assertions

JML provides the use of a specific kind of assertion known as *in-line* assertions (also called *intraconditions*). These assertions can be specified in the method body. In other words, they are interwoven with Java code. For example, consider while setting the size of packages (using the method setSize from Package of Figure 2.1), we want to ensure that the dimension of a package is greater than zero and does not exceed 400 square centimeters:

```
void setSize(double width, double height) {
  //@ assert width > 0 && height > 0;
  //@ assert width * height <= 400; // max dimension
  this.width = width;
  this.height = height;
 }
```

Here, when the execution of method setSize reaches the assertions, the above expressions must be satisfied; otherwise a JMLAssertError is raised to signal the assertion violation.

JML provides several kinds of in-line assertions, such as `assert` and `assume` statements, `loop invariant` among others. Additional information about the kinds of in-line assertions and their implementations can be found in [LPC$^+$08].

2.4.4 Null is not the default

Null pointer exceptions are one of the common faults raised by components written in object-oriented languages such as Java. For example, if x is null then x.f and x.m() result both in a NullPointerException. As described above, such null pointer exceptions make expression evaluations to be invalid in assertions.

One can prevent this problem by explicitly declaring every reference type field, return type, and parameter type as **non_null**.

```
void deliver(/*@ non_null @*/Package p, /*@ non_null @*/String destination){...}
```

The deliver method has two parameter types that are reference types. Hence, we can add the **non_null** annotation and any call to deliver with null arguments will raise a contract error in JML. If the method does allow null references, the programmer can declare the type do be **nullable**:

```
void deliver(/*@ nullable @*/Package p, /*@ non_null @*/String destination){...}
```

If the implementation of the deliver method handles internally null references of packages, we can add the **nullable** annotation to the parameter type Package of deliver.

Previously, the JML semantics considered reference types to be implicitly null (like using the **nullable** annotation for every reference type). However, Chalin *et al.* [CR05a, CR05b, CJ07, CJR08] stated that programmers want more than 50% of declarations of reference types to be non-null. That said, in the old versions of JML, programmers needed to explicitly add the **non_null** annotations to most of the declarations. As a result, programmers may forget to add some **non_null** annotations to some reference type, thus leading clients to call methods with null arguments resulting in NullPointerException. Therefore, based on this evidence, Chalin *et al.* proposed to change the JML semantics by allowing reference type declarations to be **non_null** by default. Since then, JML adopted the **non_null** semantics by default.

2.4.5 Method Specifications

In JML, method specifications contain pre- and postconditions based on Hoare-style [Hoa69, Hoa72], but with many extensions like implications (see Subsection 2.4.3). It also offers some features that are not standard. For example, JML makes a clear distinction between normal and exceptional postconditions.

Requires Clauses

A requires clause specifies a precondition of method or constructor. Preconditions are predicates that must hold before method (or constructor) execution. The general form of a precondition definition is as follows:

```
Requires-Clause ::= Requires-Keyword [!] Pred ;
      | Requires-Keyword \same;
Requires-Keyword ::= requires | pre
Pred ::= Predicate | \not_specified
```

The predicate in a **requires** clause can refer to any visible fields and to method parameters. See Section 2.4.9 for more details on visibility in JML.

Any number of requires clauses can be included in specification. Multiple requires clauses in a specification denotes the same as a single requires clause whose precondition predicate is the conjunction of these preconditions. For example:

```
requires P;
requires Q;
```

means the same thing as:

```
requires P && Q;
```

When a requires clause is omitted in a specification, JML assumes a default precondition with the meaning **\not_specified**. In terms of runtime assertion checking, in JML this works exactly as checking **requires true**. In relation to **\same**, we explain its benefit in Section 2.4.10 after describing JML specification cases, inheritance, and privacy of specifications.

For a concrete example of JML requires clause, please recall the method setSize from Package (see Figure 2.1). Previously we showed how to use the JML **assert** statement to ensure that the dimension of package is greater than zero and does not exceed 400 square centimeters. Now, let us use the JML requires clauses for this design constraint:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
void setSize(double width, double height) {...}
```

Instead of being within the method's body (like JML inline assertions), note that the preconditions are added in the method's header. If we consider other package's methods such as reSize and containsSize, we note that they have the same design constraint on their input parameters. Thus, we also have the same requires clauses for them:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
void reSize(double width, double height) {...}

//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
boolean containsSize(double width, double height) {...}
```

In relation to the keyword **pre**, its meaning is the same as the **requires** keyword. In other words, **pre** is a syntactic sugar for **requires**.

Ensures Clauses

As aforementioned, in JML we have a way to specify two kinds of postconditions. One responsible for normal termination and one responsible to specify exceptional behavior. The JML ensures clauses are used to specify a normal postcondition. Normal postconditions are predicates that must hold after method (or constructor) execution without throwing any exception. The general syntax is as follows:

```
Ensures-Clause ::= Ensures-Keyword [!] Pred ;
Ensures-Keyword ::= ensures | post
Pred ::= Predicate | \not_specified
```

The predicate in an **ensures** clause can refer to any visible fields (See Section 2.4.9 for more details on visibility in JML), the method parameters, and the return of a method when it is non-void.

The following ensures clause

```
ensures Q;
```

means if the method execution terminates normally(i.e., without throwing an exception), then the predicate Q must hold.

Any number of ensures clauses can be included in specification. Multiple ensures clauses in a specification denotes the same as a single ensures clause whose postcondition predicate is the conjunction of these postconditions. For example:

```
ensures P;
ensures Q;
```

means the same thing as:

```
ensures P && Q;
```

When an ensures clause is omitted in a specification, JML assumes a default normal postcondition with the meaning **not_specified**. In terms of runtime assertion checking, in JML this works exactly as checking **ensures true**.

For concreteness, recall again the method setSize from Package (see Figure 2.1). After the method setSize's execution, we need to ensure that the Package's fields be properly updated. Hence, we have the following ensures clauses:

```
//@ ensures this.width == width;
//@ ensures this.height == height;
void setSize(double width, double height) {...}
```

These ensures clauses guarantee that the fields width and height be properly assigned to the input parameters.

Consider now the following normal postcondition for the Package method getSize:

```
//@ ensures \result == this.width * this.height;
double getSize(){...}
```

Note the use of the **result** expression. It stands for the result that is returned by the non-void method getSize. Hence, the getSize's ensures clause states that the method must return the Package's dimension after execution.

In relation to the keyword **post**, its meaning is the same as the **ensures** keyword. In other words, **post** is a syntactic sugar for **ensures**.

Signals Clauses

The signals clauses are the second kind of JML postconditions. It is used to specify the exceptional behavior of methods. Exceptional postconditions are predicates that must hold when a method (or constructor) throws an exception. The syntax is as follows:

```
Signals-Clause ::= Signals-Keyword (reference-type [ident]) [!] Pred ;
Signals-Keyword ::= signals | exsures
```

Consider the general form:

```
signals (E e) P;
```

E has to be a subclass of java.lang.Exception and the variable e is bound in *P*. If *E* is a checked exception, it needs to be one of the exceptions listed in the method or constructor's throws clause, or a subclass or a superclass of *E*. That said, when the constrained method (or constructor) terminates by throwing an exception of type *E*, then the predicate *P* must hold. The predicate in *P* can refer to any visible fields (See Section 2.4.9 for more details on visibility in JML) and to method parameters.

As with requires and ensures clauses, multiple signals clauses are allowed in specification. For example:

```
signals (E1 e) R1;
signals (E2 e) R2;
```

means the same as:

```
signals (Exception e) ((e instanceof E1) ==> R1)
                      && ((e instanceof E2) ==> R2);
```

If an exception is thrown that is both of type $E1$ and $E2$, then both predicates $R1$ and $R2$ must hold.

It is important to note that signals clauses specify conditions that must hold when certain exceptions are thrown. Signals clauses cannot be used to force a specific exception to be thrown.

When a signals clause is omitted in a specification, JML assumes a default exceptional postcondition that allows only the exceptions in throws clause and the instances (including subtypes) of java.lang.RuntimeExcetpion to be thrown in a method or constructor execution (when terminates abnormally).

As a concrete example, consider again the setSize from Package (see Figure 2.1). Consider further that this method can throw an exception say SizeDimensionException if the package dimension exceeds its permitted dimension. In this context, consider the following signals clause:

```
//@ signals (SizeDimensionException) width * height > 400;
void setSize(double width, double height) throws SizeDimensionException {
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

This means that when the exception SizeDimensionException is thrown, the condition width $*$ height > 400 should be satisfied; otherwise an exceptional postcondition error in JML is raised to signal this contract violation.

Imagine now that we want to forbid any instance (including subtypes) of java.lang.-RuntimeException to be thrown by the method setSize. Hence, we have the following augmented signals specification:

```
//@ signals (SizeDimensionException) width * height > 400;
//@ signals (RuntimeException) false;
void setSize(double width, double height) throws SizeDimensionException {
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

Now if we want to forbid any exception except SizeDimensionException to be thrown, we can write the following signals clause in JML for the method setSize:

```
//@ signals (SizeDimensionException) width * height > 400;
//@ signals (Exception e) e instanceof SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException {
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

In JML, any Java error cannot be mentioned in the signals clause. JML does not constrain instances (or any subtype) of java.lang.Error. When errors occur, they should stop program execution and no JML contracts can interfere in this behavior.

In relation to the keyword **exsures**, its meaning is the same as the **signals** keyword. In other words, **exsures** is a syntactic sugar for **signals**.

Signals Only Clauses

The JML **signals_only** clause is just a syntactic sugar/abbreviation for a standard **signals** clause. It is used to specify what exceptions may be thrown by a method (or constructor), and thus, implicitly, which exceptions may not be thrown. The general form of a **signals_only** clause is as follows:

```
Signals-Only-Clause ::= Signals-Only-Keyword Reference-Type [ , Reference-Type] ... ;
        | Signals-Only-Keyword \nothing ;
Signals-Only-Keyword ::= signals_only
```

All of the reference-types named in a **signals_only** clause must be subtypes of java.lang.Exception. Each reference-type that is a checked exception type must either be one of the exceptions listed in the method (or constructor's) throw clause, or a subclass or a superclass of such specified reference-type.

The following general form:

```
signals_only E1, E2, ..., En ;
```

is a syntactic sugar for the following

```
signals (Exception e)
            e instanceof E1
        ||  e instanceof E2
        ||  ...
        ||  e instanceof En
```

That is, if a method or constructor throws an exception, it must be an instance of one of the types specified in the **signals_only** clause.

JML also allows multiple **signals_only** clause in specification. So, the following clauses

```
signals_only E1;
signals_only E2;
```

is equivalent to the following:

```
signals_only E1, E2 ;
```

Therefore, in the end, the JML **signals_only** clauses can be thought of a set of exceptions that can be thrown during a method (or constructor) execution.

Consider again the following signals clauses for method setSize from Package (see Figure 2.1), previously discussed:

```
//@ signals (SizeDimensionException) width * height > 400;
//@ signals (Exception e) e instanceof SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException {
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

We can rewrite it by using a JML **signals_only** clause:

```
//@ signals (SizeDimensionException) width * height > 400;
//@ signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException {
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

This is equivalent of the previous signals clause one and does only allow instances of SizeDimensionException exception to be thrown when setSize ends in an abnormal form.

If we consider now the discussed preconditions to protect the maximum Package's size dimension to be set, we can change our specification to forbid exceptions to be thrown. In this case we have the following updated specification for setSize:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ signals_only \nothing;
void setSize(double width, double height) {...}
```

With this specification, all exceptions (including runtime exceptions) are forbidden to be thrown. If we decide to go back and allow at least runtime exceptions to be thrown, we can refine the above specification to the following:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ signals_only RuntimeException;
void setSize(double width, double height) {...}
```

2.4.6 Method Specification Cases

Until now, we discussed several JML specification clauses (e.g., **requires**) that can be used to describe behavior of Java methods or constructors. The overall syntax is described below:

```
Method-Specification ::= Specification
Specification ::= Requires-Clause [Requires-Clause] ...
      | Ensures-Clause [Ensures-Clause] ...
      | Signals-Clause [Signals-Clause] ...
      | Signals-Only-Clause [Signals-Only-Clause] ...
```

As an example, consider the following specification clauses for the method setSize in Package (Figure 2.1):

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
void setSize(double width, double height) {...}
```

As discussed, these specifications constrain the package's dimension (precondition), ensures that the Package's fields width and height be properly assigned to the corresponding input parameters (normal postconditions), and guarantees that no exception (that is an instance of java.lang.Exception) is thrown after the method execution. In JML, we can consider this method specification as a single specification case. JML language gives more expressiveness by allowing several specification cases for a single method declaration. Hence, we have the following revised syntax for method (or constructor) specification:

```
Method-Specification ::= Specification
Specification ::= Spec-Case [also Spec-Case] ...
Spec-Case ::= Requires-Clause [Requires-Clause] ...
      | Ensures-Clause [Ensures-Clause] ...
      | Signals-Clause [Signals-Clause] ...
      | Signals-Only-Clause [Signals-Only-Clause] ...
```

JML uses the keyword **also** to combine one specification case to another. Each specication case has a precondition that tells when that specication case applies to a method (or constructor). JML's **also** joins together specication cases in a way that makes sure that, whenever a specication case's precondition holds for a method or constructor execution, its corresponding postcondition must also hold. That is, in general a JML method specication may consist of several specication cases, and all these specication cases must be satised by a correct implementation.

One reason for using **also** and separate specication cases is to make distinct execution scenarios clear to the specication's reader. Consider now the two specification cases for the method setSize:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
//@ also
//@ requires width > 0 && height > 0;
//@ requires width * height > 400; // exceeding allowed dimension
//@ ensures this.width == \old(this.width);
//@ ensures this.height == \old(this.height);
//@ signals(SizeDimensionException) width * height > 400;
//@ signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException{
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

Each specification case handles distinct scenarios during a method execution. The first specification case (just as before the **also**) provides the same specification we showed before. It denotes the expected behavior in a successful scenario when the size dimension is in its expected limits. The main difference now is that we also provided another specification case (guarded by a different precondition) for a different behavior; that is, if we exceed now the package's size dimension, we expect the fields to keep their old values/states (see the JML **old**) expression, and this scenario must finish by throwing the SizeDimensionException exception. The **old** operator is often used in the postconditions for methods that change the state of an object [Mey00].

Another way to refine this specification is to handle scenarios where one tries to pass negative or zero valued arguments for the package's size dimension. Instead of throwing a JML precondition error, we just allow the method to receive and properly handle the values. Of course that negative arguments will not be accepted to set a valid size dimension. See the following refined version for method setSize's specification.

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
//@ also
//@ requires width > 0 && height > 0;
//@ requires width * height > 400; // exceeding allowed dimension
//@ ensures this.width == \old(this.width);
//@ ensures this.height == \old(this.height);
//@ signals(SizeDimensionException) width * height > 400;
//@ signals_only SizeDimensionException;
//@ also
//@ requires width <= 0 && height <= 0;
//@ ensures this.width == \old(this.width);
//@ ensures this.height == \old(this.height);
//@ signals(SizeDimensionException) width <= 0 && height <= 0;
//@ signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException{
 if(width * height > 400)
   throw new SizeDimensionException();
 else if(width <= 0 && height <= 0)
   throw new SizeDimensionException();
 ...
}
```

Therefore, when the third specification has its precondition satisfied, the rest of the specification case must also be satisfied. Note that the method implementation was adapted to match the desired behavior, which is now in accordance with method's specification; otherwise a violation will be detected and either the specification or the implementation should be fixed.

Leavens refer to the combination of two method specications with **also** as their "join", since it is technically the join with respect to the renement scenarios on method specications [Lea06].

Regarding preconditions and postconditions for specification cases, we use the following simpler definition based on Leavens [Lea06, Definition 1] notation:

Definition 2.4.1. Join of JML method specications. *Let* $T \rhd (pre, post)$ *and* $T \rhd (pre', post')$ *be specification cases of an instance method* m *in type* T*. Then the join of* $(pre, post)$ *and* $(pre', post')$ *for* T*, written* $(pre, post) \sqcup^T (pre', post')$*, is the specification* $T \rhd (p,q)$ *with precondition* p*:*

$$pre \ \texttt{||} \ pre' \tag{2.1}$$

and postcondition q*:*

$$(\texttt{\textbackslash old(}pre\texttt{)} \texttt{==>} post) \ \texttt{\&\&} \ (\texttt{\textbackslash old(}pre'\texttt{)} \texttt{==>} post') \tag{2.2}$$

□

In this definition, the precondition of the join of two specification cases is their disjunction (with || as in Java). The postcondition of the join is a conjunction of implications (written ==> in JML's notation), such that when one of the preconditions holds, then the corresponding postcondition must hold. The template of Desugaring 1 illustrates the mechanics of Definition 2.4.1. Figure 2.3 shows a concrete example of the application of the template Desugaring 1.

Desugaring 1. ⟨*the join of specification cases*⟩

```
class T {
//@ requires pre;
//@ ensures post;
//@ also
//@ requires pre';
//@ ensures post';
 void m() {
   body
 }
}
```

$\Rightarrow$

```
class T {
//@ requires pre || pre';
//@ ensures (\old(pre) ==> pos)
//@ && (\old(pre') ==> pos');
 void m() {
   body
 }
}
```

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400;
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ also
//@ requires width > 0 && height > 0;
//@ requires width * height > 400;
//@ signals(SizeDimensionException) width * height > 400;
void setSize(double width, double height)
      throws SizeDimensionException {...}
```

$\Downarrow$

```
//@ requires ((width > 0 && height > 0)
//@             && (width * height <= 400))
//@         || ((width > 0 && height > 0)
//@             && (width * height > 400))
//@ ensures \old(((width > 0 && height > 0)
//@             && (width * height <= 400)))
//@          ==>
//@            (this.width == width)
//@            && (this.height == height)
//@ signals(SizeDimensionException)
//@            \old(((width > 0 && height > 0)
//@             && (width * height > 400)))
//@          ==> (width * height > 400)
void setSize(double width, double height)
      throws SizeDimensionException {...}
```

Figure 2.3: An example of the application of the template Desugaring 1 for method setSize.

Lightweight and Heavyweight specifications

In JML one is not required to specify behavior completely. Indeed, JML has a style of method specification case, called lightweight. In this style, programmers only need to specify what interest them. All the specifications discussed so far are lightweight.

In a heavyweight specification case, on the other hand, JML expects that the programmer works with a “complete” specification that includes both normal and exceptional situations or at least either normal or exceptional. Programmers distinguish between such cases of method specifications by using different syntaxes. In essence, in a method specification case in heavyweight mode, the programmer uses one of the behavior keywords, such as **normal_behavior**, **exceptional_behavior**, or **behavior**. The absence of these keywords denotes a lightweight specification, as mentioned, the ones we showed until now.

The revised syntax for method specification is as follows:

```
Method-Specification ::= Specification
Specification ::= Spec-Case [also Spec-Case] ...
Spec-Case ::= Lightweight-Spec-Case | Heavyweight-Spec-Case
Lightweight-Spec-Case ::= Generic-Spec-Clause
Heavyweight-Spec-Case ::= Behavior-Spec-Case
      | Normal-Behavior-Spec-Case
      | Exceptional-Behavior-Spec-Case
Behavior-Spec-Case ::= [privacy] Behavior-Keyword
                            Generic-Spec-Clause
Behavior-Keyword ::= behavior | behaviour
Normal-Behavior-Spec-Case ::= [privacy] Normal-Behavior-Keyword
                                   Generic-Spec-Clause
Normal-Behavior-Keyword ::= normal_behavior | normal_behaviour
Exceptional-Behavior-Spec-Case ::= [privacy] Exceptional-Behavior-Keyword
                                   Generic-Spec-Clause
Exceptional-Behavior-Keyword ::= exceptional_behavior | exceptional_behaviour
Generic-Spec-Case ::= Requires-Clause [Requires-Clause] ...
      | Ensures-Clause [Ensures-Clause] ...
      | Signals-Clause [Signals-Clause] ...
      | Signals-Only-Clause [Signals-Only-Clause] ...
```

The code below is an example of a JML heavyweight of kind **behavior**.

```
//@ behavior
//@   requires width > 0 && height > 0;
//@   requires width * height <= 400; // max dimension
//@   ensures this.width == width;
//@   ensures this.height == height;
//@   signals_only \nothing;
void setSize(double width, double height) {...}
```

The **behavior** spec is useful to describe a complete behavior including normal and exceptional ones. We can refine the above specification to the following **normal_behavior** specification case:

```
//@ normal_behavior
//@   requires width > 0 && height > 0;
//@   requires width * height <= 400; // max dimension
//@   ensures this.width == width;
//@   ensures this.height == height;
void setSize(double width, double height) {...}
```

Note that now we removed the **signals_only** clause. Since it is a **normal_behavior** specification it includes only detailed design about normal termination. Hence, such a **normal_behavior** specification is a sugar for a **behavior** specification with an implicit **signals_only \nothing**. Therefore, in a normal behavior mode, the method is not allowed to throw exceptions; this includes runtime exceptions as well.

Consider now the following refinement:

```
//@ normal_behavior
//@  requires width > 0 && height > 0;
//@  requires width * height <= 400; // max dimension
//@  ensures this.width == width;
//@  ensures this.height == height;
//@ also
//@ exceptional_behavior
//@  requires width > 0 && height > 0;
//@  requires width * height > 400;
//@  signals(SizeDimensionException) width * height > 400;
//@  signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException{
 if(width * height > 400) throw new SizeDimensionException();
 ...
}
```

Contrasting to a **normal_behavior** specification case, when the precondition of an **exceptional_behavior** specification case is satisfied, the method must terminate abnormally by throwing an exception. Thus **exceptional_behavior** specification cases are used to specify properties when a method throws exceptions. Since a method or constructor having a satisfied exceptional specification case cannot terminate in a normal form, it has an implicitly **ensures false** clause. Hence the above specification provides two distinct scenarios where the normal one cannot throw any exception and the second cannot finish without throwing an exception.

Note that an **exceptional_behavior** specification case cannot force a specific exception to be thrown, but it can enforce to have an exception thrown at least; otherwise an exceptional postcondition error in JML is signalled during runtime checking.

In the revised syntax for method specification shown above, we can see the use of the optional [*Privacy*] syntax. We describe it in Section 2.4.9.

2.4.7 Type Specifications

In addition to field and method declarations and specifications, a type specification in JML may also contain invariants, history constraints, and **initially** clauses. In the following we discuss invariant specifications. For information about other type specifications and even more details about invariants in JML, please refer to [Lea06, LPC$^+$08, LBR06].

Invariants

Invariants are predicates that must hold in all visible states during a method or constructor execution. A state is a visible state for an object o if it is the state that occurs at one of these moments in a program's execution:

- after the execution of a non-helper constructor that is initializing o;
- before the execution of a non-helper finalizer that is finalizing o;
- before and after the execution of a non-helper non-static non-finalizer method with o as the receiver;

These visible states are enforced to instance invariants in JML. However, in JML we can have both instance and static invariant declarations[1]. A static invariant can only refer to static members of an object. An instance invariant, on the other hand, may refer to both static and non-static members. We say members since we can also refer to the result of a method as a part of the invariant condition.

In relation to the visible states in which static invariants must hold, we list them below:

- before and after the execution of a non-helper of a non-helper constructor that is initializing *o*;
- before and after the execution of a non-helper static method in *o*'s class or some subclass of *o*'s class.

Note that these visible states exclude executions of helper methods or constructors declared with the JML **helper** modifier. For more information about helper methods or constructors, please refer to [LPC+08].

The JML invariant syntax is as follows:

```
Invariant ::= [Privacy] [static] Invariant − Keyword Predicate ;
Invariant − Keyword ::= invariant
```

For a concrete example of invariants, consider the two instance invariants below for the type Package in Figure 2.1:

```
public class Package {
    double width, height;
    //@ invariant this.width >= 0 && this.height >= 0;
    double weight;
    //@ invariant this.weight >= 0;
   ...
}
```

The first invariant restricts package's dimension to be always greater than or equal to zero and the second one has the same design constraint on package's weight.

As can be seen above in the invariant's syntax, we can use a visibility modifier (privacy information) on an invariant declaration. We discuss this privacy feature about JML specifications in Section Section 2.4.9.

Like pre- and postconditions in JML, we can have multiple invariant declarations like those above. Multiple invariant declarations are desugared into a single one. All invariants are combined by conjunction.

2.4.8 Specification Inheritance

In JML, there are various ways to inherit specifications: subclassing, interface extension and implementation, and refinement [LBR06]. A subtype inherits not only fields and methods from its supertypes, but also specifications such as pre- and postconditions,

[1]JML carefully uses the standard names like object or class invariants. Object invariants are actually related to the instances of a class, whereas class invariants denote properties of the class itself (no instance involved). Hence, object invariants in JML is also called instance invariants. Class invariants in JML is also known as static invariants.

and invariants. To provide the effect of specification inheritance, JML also employs the **also** construct (already discussed in Section 2.4.6), which denotes a combination (join) of specification cases (see Definition 2.4.1), which consist of clauses including pre-, postconditions, and so forth. The main difference is that the two specification cases connected by **also** is not local in the same method declaration; one is located in the overridden method in a supertype and the other is located in an overriding method in a subtype.

For a concrete example of a specification inheritance in JML, consider the following specifications based on Figure 2.1:

```
public class Package {
 double width, height;
 //@ invariant this.width >= 0 && this.height >= 0;
 double weight;
 //@ invariant this.weight >= 0;

 //@ requires weight > 0;
 //@ requires weight <= 5;
 //@ ensures this.weight == weight;
 //@ signals_only \nothing;
 void setWeight(double weight) {...}
 ...
}
public class GiftPackage extends Package {
 //@ also
 //@ requires weight > 0;
 //@ requires weight <= 8;
 //@ ensures this.weight == weight;
 //@ signals_only \nothing;
 void setWeight(double weight) {...}
 ...
}
```

In this example, the method setWeight in Package is only allowed to set a package's weight with a maximum of 5kg. In the overriding method setWeight in subtype GiftPackage, we refined the specification to allow heavier packages of 8kg int total permitted. The stronger precondition is located in the specification case in Package type, whereas the weaker precondition is placed in the GiftPackage type. These specification cases are connected using the **also** construct in JML. It means that we should consider all the local specification cases in GiftPackage.setWeight in addition to ones in its overridden method in Package.setWeight.

All overriding methods automatically inherit specifications from its supertypes. So if one forgets to use the **also** construct, a JML tool should issue a warning about it. The warning is to make the programmer aware that even if there is no **also**, specifications of supertypes are still been considered.

Besides method's pre- and postconditions, subtypes inherit other kinds of specifications like invariants. In this context, the above two invariants declared in type Package are also inherited by GiftPackage. Hence, any call to setWeight of type GiftPackage must establish those invariants both before and after its execution.

The Definition 2.4.1 (described in Section 2.4.6) describes the join of method specification cases (local or inherited) in JML. To define the join of (inherited) invariants, Leavens introduces the notation $added\_inv^T$ [Lea06, Definition 2].

See the definition below:

Definition 2.4.2. Extended specification for invariant. *the extended invariant of T is the conjunction of all added invariants in T and its proper supertypes:*

$$ext\_inv^T = \bigwedge\{added\_inv^U | U \in supers(T)\} \tag{2.3}$$

□

2.4.9 Privacy Modifiers and Visibility in Specifications

Information hiding [Par72] (also known as black-box abstraction) is a widely accepted principle that aids in software development. It advocates that a module should expose its functionality but hide its implementation behind an interface. This supports modular reasoning and independent evolution/maintenance of the hidden parts of a module. If programmers have carefully chosen to hide the parts “most likely” to change [Par72], most changes, in the hidden implementation details, do not affect client modules.

Information hiding and its benefits apply not only to code but also to other artifacts, such as documentation and specifications [LM07, Par11]. In this context, a specification declaration in JML can be **public**, **protected**, **private**, or default (package) visibility. However, JML imposes some extra rules for specifications in addition to the usual Java visibility rules [LBR06, LM07] to preserve information hiding. We discuss them below.

Rule 1

The first rule is based on Leavens and Müller’s Rule 1 [LM07]. It states that a specification cannot refer to members that are more hidden than the specification own visibility. The reason for this restriction is that the programmers/users who are allowed to see the specifications should be able to see each of the members referred in those specifications; otherwise, they might not understand it [LM07, Mey00].

In practical terms, this information hiding rule is fundamental while a programmer is performing, for example, runtime checking. Suppose that the programmer gets a precondition or postcondition violation where both are part of a public specification mentioning hidden (e.g., private) fields in those clauses. The problem is that the programmer would see a precondition or postcondition violation referring to hidden members that are not visible to public clients. Thus such contract violations, involving hidden fields, are not meaningful to all clients. As a consequence, clients confront an issue that the interface claimed to hide [Kic96].

Therefore, public clients should be able to see all the declarations of public members referred within the specification. However, public specifications cannot contain protected, default access, or private members mentioned on them.

Consider the following invariant example for the type Package (Figure 2.1):

```
 1 public class Package {
 2  protected double width;
 3  private double height;
 4  double weight;
 5
 6  //@ public invariant this.width >= 0;       // illegal!
 7  //@ public invariant this.height >= 0;      // illegal!
 8  //@ public invariant this.weight >= 0;      // illegal!
 9
10  //@ protected invariant this.width >= 0;    // legal!
11  //@ protected invariant this.height >= 0;   // illegal!
12  //@ protected invariant this.weight >= 0;   // illegal!
13
14  //@ invariant this.weight >= 0;             // legal!
15  //@ invariant this.height >= 0;             // illegal!
16
17  //@ private invariant this.height >= 0;     // legal!
18
19  ...
20 }
```

To illustrate the JML rules on visibility, we declare each field with a different privacy modifier, and then we write several invariant clauses with also different privacy levels. However, in the above example, only the invariants declared on lines 10, 14, and 17 are valid. The others violate our first rule described here.

In relation to method specification cases, the same JML rule applies. For example, consider the following heavyweight (normal) specification case for the method setSize in Package:

```
public class Package {
 protected double width, height;

//@ public normal_behavior
//@  requires width > 0 && height > 0;
//@  requires width * height <= 400;
//@  ensures this.width == width;      // illegal!
//@  ensures this.height == height;    // illegal!
void setSize(double width, double height) {...}

...
}
```

As observed, the protected fields width and height are both mentioned in **ensures** clauses of a **public** specification case for the method setWeight. Thus, both usage are illegal. The question that arises now is how to export a valid public specification case for method setSize? To answer this question, JML has two ways to handle this situation.

The first one is to use **spec_public** modifier as follows:

```
public class Package {
 protected /*@ spec_public @*/ double width, height;

//@ public normal_behavior
//@  requires width > 0 && height > 0;
//@  requires width * height <= 400;
//@  ensures this.width == width;     // legal!
//@  ensures this.height == height;   // legal!
void setSize(double width, double height) {...}

...
}
```

This JML modifier allows one to declare both fields width and height as public for specification purposes. This way, the above public specification case is now valid for method setSize. For the similar purpose, JML also offers the **spec_protected** modifier (please refer to [LBR06, LPC$^+$08] for more information).

It is important to note that when a method uses a lightweight specification case where there is no privacy modifier, then the spec case has the same visibility as the method itself.

The second way to handle hidden fields in specifications is by using model fields [CLSE05, Lea06]. Model fields are specification-only fields that give an abstraction of some concrete state. Concrete state refers to state variables, which is one or more fields in a type. Consider the following example:

```
public class Package {
 //@ public model double width;
 protected double _width;
 //@ protected represents width = _width;

 //@ public model double height;
 protected double _height;
 //@ protected represents height = _height;

//@ public normal_behavior
//@  requires width > 0 && height > 0;
//@  requires width * height <= 400;
//@  ensures this.width == width;     // legal!
//@  ensures this.height == height;   // legal!
void setSize(double width, double height) {...}

...
}
```

In this example, the value of the public model fields width and height are determined directly by the value of the corresponding protected (Java) fields _width and _height. This relationship is specified by JML **represents** clauses. Those **represents** clauses are protected, since it mentions protected fields. In the end, the public specification case is valid again.

If we consider the example and the two forms used to make a field visible in a specification, we can observe that the second form only provided the benefit of field name encapsulation. So if we refactor and change the concrete field _width to something else, public clients are not affected due to the benefit of information hiding. However, model fields could be used to hide more things than just a single name. A single model field can be used to abstract several fields or even their concrete type representation. For example, we can use the following refined specification to hide the fields involved in package dimension:

```
public class Package {
 //@ public model double dimension;
 protected double width, height;
 //@ protected represents dimension = width * height;
...
}
```

Now we have a model field with no direct corresponding Java field. Thus the model field dimension is an abstract representation for the concrete Java fields width and height. We can even change the type to a more abstract representation using the embedded JML model Types. For example,

```
public class Package {
 //@ public model JMLDouble dimension;
 protected double width, height;
 //@ protected represents dimension = new JMLDouble (width * height);
...
}
```

the type JMLDouble now hides what representation is actually being used by the implementation. One does not know if we use a Java primitive type **double** or the wrapper class Double.

Rule 2

The second rule for visibility in specifications prohibits a specification to constrain fields that are more visible than the specification itself (see Leavens and Müller's Rule 6 [LM07] for more information). In particular this rule applies to JML type specifications like invariants and history constraints.

For example, a private invariant cannot mention a public field, since clients could see the public field without seeing the invariant, and thus would not know when they might violate the private invariant by assigning to the public field.

In order to illustrate how this rule works in JML, please consider the following Package's invariant:

```
public class Package {
 public double width, height;
 protected double weight;

 //@ protected invariant this.width >= 0;    // illegal!
 //@ protected invariant this.height >= 0;    // illegal!

 //@ invariant this.width >= 0;   // illegal!
 //@ invariant this.height >= 0;   // illegal!
 //@ invariant this.weight >= 0;   // illegal!

 //@ private invariant this.width >= 0;     // illegal!
 //@ private invariant this.height >= 0;     // illegal!
 //@ private invariant this.weight >= 0;     // illegal!

 ...
}
```

All of these invariants are illegal since they violate rule 2. In other words, all of these invariants are more hidden than the constrained fields. This situation leads to unsoundness on program correctness.

JML Syntax for visibility involving fields

The privacy syntax in JML involving fields is as follows:

```
JML-Field-Decl ::= Model-Field-Decl | Represents-Clause | Field-Decl
Model-Field-Decl ::= [Privacy] [Modifiers] Model-Keyword FieldType Field-Ident ;
Model-Keyword ::= model
Represents-Clause ::= [Privacy] [static] Represents-Keyword Field-Ident = Exp ;
Represents-Keyword ::= represents
Field-Decl ::= [Privacy] [Modifiers] [JML-Modifier] FieldType Field-Ident [= Exp] ;
Exp ::= FieldAccessExp | MethodCallExpr | ConstantExp
...
```

2.4.10 Modularity notes in JML when applying Design By Contract

As an expressive design by contract language for Java, JML offers some mechanisms that improve contract modularity while specifying classes and methods. In the following we discuss these modularity mechanisms offered by JML for contract specification. For the discussion recall the running example illustrated in Figure 2.1.

Pre- and postconditions as invariants

When a programmer is writing specifications, the tendency is to provide pre- and postconditions for methods. Afterwards type specifications like invariants are also considered. After all pre- and postconditions are met, the programmer can analyze if there are pairs of a pre- and postconditions that are recurrent scattered across all the methods in a particular type. If so, we can extract those pre- and postconditions and modularize them using a quantified statement like an invariant declaration.

For example, consider the following pre- and postconditions for the methods in type Package:

```
public class Package {
 double width, height, weight;

 //@ requires this.width >= 0;
 //@ requires this.height >= 0;
 //@ requires this.weight >= 0;
 //@ requires ...
 //@ ensures this.width >= 0;
 //@ ensures this.height >= 0;
 //@ ensures this.weight >= 0;
 //@ ensures ...
 void setSize(double width, double height) {...}

 //@ requires this.width >= 0;
 //@ requires this.height >= 0;
 //@ requires this.weight >= 0;
 //@ requires ...
 //@ ensures this.width >= 0;
 //@ ensures this.height >= 0;
 //@ ensures this.weight >= 0;
 //@ ensures ...
 void reSize(double width, double height) {...}

 //@ requires this.width >= 0;
 //@ requires this.height >= 0;
 //@ requires this.weight >= 0;
 //@ requires ...
 //@ ensures this.width >= 0;
 //@ ensures this.height >= 0;
 //@ ensures this.weight >= 0;
 //@ ensures ...
 boolean containsSize(double width, double height) {...}

 //@ requires this.width >= 0;
 //@ requires this.height >= 0;
 //@ requires this.weight >= 0;
 //@ requires ...
 //@ ensures this.width >= 0;
 //@ ensures this.height >= 0;
 //@ ensures this.weight >= 0;
 //@ ensures ...
 double getSize() {...}

 //@ requires this.width >= 0;
 //@ requires this.height >= 0;
 //@ requires this.weight >= 0;
 //@ requires ...
 //@ ensures this.width >= 0;
 //@ ensures this.height >= 0;
 //@ ensures this.weight >= 0;
 //@ ensures ...
 void setWeight(double weight) {...}
 ... // other methods
}
```

As observed, there is a set of pre- and postconditions that are repeated in the specification for all methods in Package. In order to avoid such annotation burden (scattering), we write an invariant per pair of repeated pre- and postcondition. Hence, we have now the specifications more modular as follows:

```
public class Package {
 double width, height, weight;
 //@ invariant this.width >= 0;
 //@ invariant this.height >= 0;
 //@ invariant this.weight >= 0;

 //@ requires ...
 //@ ensures ...
 void setSize(double width, double height) {...}

 //@ requires ...
 //@ ensures ...
 void reSize(double width, double height) {...}

 //@ requires ...
 //@ ensures ...
 boolean containsSize(double width, double height) {...}

 //@ requires ...
 //@ ensures ...
 double getSize() {...}

 //@ requires ...
 //@ ensures ...
 void setWeight(double weight) {...}
 ... // other methods
}
```

With quantified statements like invariants, we were able to remove the recurrent pre- and postconditions and reduce the overall 40 specification clauses to only 13. Three of these 13 specifications are now invariant declarations and provide the same design constraints as before.

The drawback of this approach is that we can only refactor pre- and postconditions to invariants iff they apply to all methods. That is, if we have just one method without the recurrent pre- and postconditions, we cannot modularize them by using an invariant declaration.

Reusing specifications with specification inheritance

Another way to reuse contracts is by using specification inheritance. Usually, expressive DbC languages like JML or Eiffel provide this feature. So instead of duplicating the specifications of an overridden method in an overriding method like below

```
public class Package {
 //@ requires weight > 0;
 //@ requires weight <= 5;
 //@ ensures this.weight == weight;
 void setWeight(double weight) {...}
 ...
}
public class GiftPackage extends Package {
 //@ requires weight > 0;
 //@ requires weight <= 5;
 //@ ensures this.weight == weight;
 void setWeight(double weight) {...}
 ...
}
```

we just leave the overridden method specifications as follows

```
public class Package {
 //@ requires weight > 0;
 //@ requires weight <= 5;
 //@ ensures this.weight == weight;
 void setWeight(double weight) {...}
 ...
}
public class GiftPackage extends Package {
 void setWeight(double weight) {...}
 ...
}
```

In the overriding method, we leave no specifications because the ones in overridden method are automatically inherited and enforced with a modularization using a hierarchical structure.

Reusing preconditions with the same predicate

Often in writing a method specification in a subtype, one wants the precondition of the overriding method to be the same as that of the specification of the method being overridden. This often occurs for a method in a subclass that does something extra. JML's predicate **\same** can be used in the precondition of such a specification to say that the method's precondition is the same as that of the method being overridden [LPC+08].

For example, the overriding method (in type GiftPackage) below

```
public class Package {
 //@ requires weight > 0;
 //@ requires weight <= 5;
 //@ ensures this.weight == weight;
```

```
  void setWeight(double weight) {...}
  ...
}
public class GiftPackage extends Package {
  //@ also
  //@ requires weight > 0;
  //@ requires weight <= 5;
  //@ ensures this.weight == weight;
  //@ signals_only \nothing;
  void setWeight(double weight) {...}
  ...
}
```

has the same specifications of the overridden method except that it adds a **signals_only** clause (exceptional postcondition). For situations like that, we can use the JML **same** predicate to avoid duplicating the same preconditions. In the following we have the refined version, of the above example, using the JML **same** predicate.

```
public class Package {
  //@ requires weight > 0;
  //@ requires weight <= 5;
  //@ ensures this.weight == weight;
  void setWeight(double weight) {...}
  ...
}
public class GiftPackage extends Package {
  //@ also
  //@ requires \same;
  //@ ensures this.weight == weight;
  //@ signals_only \nothing;
  void setWeight(double weight) {...}
  ...
}
```

We are allowed to add more **requires** clauses if needed. However, in JML there is no similar construct to reuse postconditions.

Note that the **same** predicate is an exclusive construct of the JML language. No other DbC language has a similar feature.

Reusing preconditions with nested specifications

Nested specifications is another (exclusive) feature in JML that can be used to modularize recurring preconditions in local (non-inherited) specification cases. For a concrete example, consider the following specification cases for the method setSize:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ also
//@ requires width > 0 && height > 0;
//@ requires width * height > 400;
//@ signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException {...}
```

As observed, the first **requires** clause in each specification case is the same. In JML, when we have a set of recurring preconditions in each (local and non-inherited) specification case, we can write out a nested specification case by separating all recurring preconditions (**requires** clauses). The refined example of the above specifications is:

```
//@ requires width > 0 && height > 0;
//@ {|
//@   requires width * height <= 400; // max dimension
//@   ensures this.width == width;
//@   ensures this.height == height;
//@ also
//@   requires width * height > 400;
//@   signals_only SizeDimensionException;
//@ |}
void setSize(double width, double height) throws SizeDimensionException {...}
```

Now the specification cases are placed between the nested specification markers {| and |}. Note that on top of such nested specification we have the modularized **requires** clause that was repeated in each specification case before nesting.

Unfortunately, this JML feature is only available to modularize preconditions. Postconditions like **ensures** clauses are not allowed to be reused like preconditions.

Non-null as default

As discussed in Section 2.4.4, null is not the default value for objects. Instead of writing several non−**null** annotations, JML implicitly write them for you. So the non-null semantics in JML is a form to avoid scattering of annotations. However, it is no perfect. For example, assume we have a type with a hundred reference types as fields, parameter types, etc. Assume further that 50% of these reference types can be null. Hence, we need to write out **nullable** for all of the nullable types. On the other way around, if we consider that we have turned-off the non-null by default semantics to be nullable by default, and we have 50% of type declarations to be non-null, then we need to scatter such non−**null** annotations, and therefore having the same problem as before.

Untangling specifications in JML with separate files

Usually, JML specifications are written as annotation comments within .java files. However, there are some situations in which one may wish to separate specifications from source code. The first reason for writing specifications in separate files is to prevent specifications to be quite tangled and scattered (i.e., making it hard to see all of the code at once) within the source code. Hence, instead of writing specifications in a .java file, we add them in a corresponding .jml file.

Another important use of such separate files with .jml suffix is when one does not own the sources for the Java code, but want to specify them. This way, we need to put the specifications in a different file (i.e., .jml file).

Note that methods cannot contain body in such .jml files. Only the syntactic interfaces must be declared. These .jml files work just like Java interfaces.

Crosscutting contract specification support

Even though JML is an expressive DbC language for Java that can also be used to modularize recurrent specification clauses such as **requires** (preconditions), there is no support crosscutting contract specification. For example, there is no way to write the same pre- and postconditions with a quantified statement and apply to some (specific) methods in a type or set of types.

We discuss this problem in detail in Chapter 3 and solutions with AspectJ (Section 2.5) and AspectJML (Section 4), respectively.

2.4.11 Tool Support

In the following we discuss some of the main JML tools available for users.

Static verification

Esc/Java2 [FLL+02] performs compile time verifications to check some common errors in Java code, such as null dereferences, casting to incompatible types, or indexing an array out of its bounds. With Esc/Java2 one can check the consistency between the Java code and the given JML specifications. It issues a list of possible errors after the program verification. The Esc/Java2 tool uses the theorem prover Simplify [DNS05], which translates a given JML annotated program into logical formulas [LSS99].

Similar to Esc/Java2 [FLL+02], the Loop tool [vdBJ01] performs a static verification of the Java programs annotated with JML specifications. The Loop tool translates Java classes into high order logic for two theorem provers: PVS [ORS92] and Isabelle [Pau94]. The Jack tool provides an environment for Java and Java Card program verification using JML annotations. As with the Loop tool, the Jack tool translates the annotated Java class with JML into high order logic for different theorem provers such as PVS [ORS92]. The Krakatoa [MMU04] tool uses JML as specification language and produces proof obligations for the theorem prover Coq [Tea08].

Runtime assertion checking compilers

The jmlc compiler. The JML compiler (*jmlc*) [Che03] was developed at Iowa State University. It is a runtime assertion checking compiler, which converts JML annotations into runtime checks. Jmlc is built on top of the MultiJava compiler [CLCM00, CMLC06]. It reuses the front-end of existent JML tools [BCC+05] to verify the syntax and semantics of the JML annotations and produces a typechecked abstract syntax tree (AST). The compiler introduces two new compilation passes: the "runtime assertion checker (RAC) code generation"; and the "runtime assertion checker (RAC) code printing". The *RAC code generation* creates the assertion checker code from the AST. It modifies the abstract syntax tree to add nodes for the generated checking code. The *RAC code printing* writes the new abstract syntax tree to a temporary file.

The *wrapper approach* [Che03, Section 4.1.3] is a strategy used by the JML compiler to implement the assertion checking. Each method is redeclared as private with a new name. Then, a method known as *wrapper method* is generated with the name of the original method. Its surrounds the original method (now with a new name) with the assertion methods. Hence, client method calls the wrapper method, which is responsible for calling the original method with appropriate assertion checks (e.g., precondition checking). The JML compiler is responsible for controlling the order of execution of assertion methods.

The ajmlc compiler. The ajmlc compiler [RSL+08][RLL+13c] is an alternative to the classical jmlc compiler which uses aspect-oriented programming to instrument pre- and postconditions and check them during runtime. Similar to Cheon [Che03], ajmlc reuses the front-end of the JML Type Checker [BCC+05]. It traverses the typechecked AST generating *Aspect Assertion Methods* (AAM) for each Java method. Then, the AAM are compiled through the AspectJ compiler ajc or abc which weaves the AAM with the Java code. The main difference between ajmlc and jmlc is that the former does not use reflection to implement specification inheritance. This results in an instrumented bytecode compliant to both Java SE and Java ME applications. Putting in other words, with ajmlc it is possible to use the main JML features and apply them to other platforms like Java ME.

Since ajmlc uses AspectJ under the hood, we can say that the wrapper approach used in ajmlc is automatically performed by the AspectJ weaver/compiler, whereas in the standard jmlc, it is managed manually by the code generation. Moreover, the order of execution and checking of assertions are controlled by the AspectJ weaver during code instrumentation. In jmlc, it is done by the infrastructure of the JML compiler itself.

2.4.12 Exported JML example

In order to have an homogenous example that can be used in the discussion hereafter, we provide JML specifications (with some features discussed so far in this chapter) for the type Package (see our running example in Figure 2.1) in Figure 2.4.

2.5 An Overview of AspectJ

This section gives an overview of the AspectJ language, a general-purpose aspect-oriented extension to Java. It provides support for implementing crosscutting concerns (e.g., design by contract) in a modular way. In the following, we present the major features of AspectJ such as pointcuts and advice.

2.5.1 The Anatomy of an aspect

The main construct of AspectJ is an aspect. Each aspect defines a specific function that intercepts several classes in a system. Similarly to a class, an aspect can also define fields and methods and a hierarchy of aspects, through the definition of subaspects.

Aspects may change the static structure of Java programs. With aspects one can introduce new methods and fields to an existing class. In addition, aspects can be used to convert checked exceptions into unchecked ones. Also, aspects can change the hierarchy of a class, for example, making it to extend an existing class with another one. These features are part of a broader concept called static crosscutting.

Besides the static structure, aspects can also affect the dynamic behavior of a system. For instance, one can explicitly intercept certain points in the execution flow (called

```
public class Package {
 //@ public model JMLDouble dimension;
 //@ public model JMLDouble weight;
 protected double width, height, _weight;
 //@ protected represents dimension = new JMLDouble(this.width * this.height);
 //@ protected represents weight = new JMLDouble(this._weight);

 //@ public invariant this.dimension.doubleValue() >= 0;
 //@ public invariant this.weight.doubleValue() >= 0;

 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 400; // max dimension
 //@ ensures this.dimension.doubleValue() == width * height;
 //@ signals_only \nothing;
 public void setSize(double width, double height){...}

 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 400; // max dimension
 //@ requires this.dimension.doubleValue() != width * height;
 //@ ensures this.dimension.doubleValue() == width * height;
 //@ signals_only \nothing;
 public void reSize(double width, double height){...}

 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 400; // max dimension
 //@ signals_only \nothing;
 public boolean containsSize(double width, double height){...}

 //@ signals_only \nothing;
 public double getSize(){...}

 //@ requires weight > 0;
 //@ requires weight <= 5;
 //@ ensures this.weight.doubleValue() == weight;
 //@ signals_only \nothing;
 public void setWeight(double weight) {...}
}
```

Figure 2.4: The JML pre- and postconditions for Package class of Figure 2.1.

join points), and add behavior before, after, or around (instead of) the behavior of the intercepted join point.

Commonly an aspect composes several join points by means of a *pointcut*. Pointcuts select join points and context (values) at these join points. Once pointcuts are identified, the aspect define *advice* that execute when the join points (intercepted via pointcuts) are reached.

In the following sections, we discuss the main AspectJ constructs and exemplify how they can be used in practice. More detail can be found elsewhere [KHH+01, Lad03].

2.5.2 The Join Point Model

The most fundamental concept in the design of any aspect-oriented language is the join point model. According to Kiczales [KHH+01], the join point model provides the common frame of reference that makes it possible to define the structure of crosscutting concerns.

In AspectJ, join points are well-defined points in the execution of the program. AspectJ provides several kinds of join points, but this work discusses only two of them: method (or constructor) execution and method (or constructor) call join points. The former is on the declaration/server side itself. In other words, the execution join point is on the method (or constructor) body. On the other hand, the latter is located at many parts of the program, which are the places that are calling the methods. In terms of AspectJ and for the purposes of this thesis, these join points represent the most useful and commonly used points to inject the crosscutting behavior.

All join points also have a context associated with them. For example, a call to a join point in a method has the caller object, the target object, and the arguments of the method available as the context.

2.5.3 Pointcut Designators

In AspectJ, pointcut designators are used to identify collections of join points in the program flow. For example, the pointcut designator:

```
call(void Package.setSize(double, double))
```

identifies all calls to the method setSize available on Package objects (Figure 2.1), whereas the pointcut designator:

```
execution(void Package.setSize(double, double))
```

identifies all the executions of the method setSize defined in Package class.

Pointcuts designators can also be composed using pointcut operators, so for example:

```
call(void Package.setSize(double, double)) ||
call(void Package.setWeight(double))
```

identifies all calls to be either setSize or setWeight methods of Package class.

In summary, AspectJ provides one unary operation (!) and two binary operators (|| and &&) to form complex matching rules while combining several kinds of pointcuts. The unary operator ! allows the matching of all join points except those specified by the pointcut itself. For example, the following pointcut composition:

```
call(void Package.set*(..)) &&
!call(void Package.setWeight(double)) ||
```

identifies all calls to any set-like method of Package object, but excluding method setWeight. We discuss AspectJ wildcarding using * or **..** later in this section.

In relation to binary operators, the combination of two pointcuts with the || operator causes the selection of join points that match either of the pointcuts, whereas combining them with an && operator selects join points matching both pointcuts.

In AspectJ, pointcuts can be either *anonymous* or *named*. Anonymous pointcuts, like anonymous classes, are defined at the place of their usage, such as a part of the advice declaration (we discuss AspectJ's advice after pointcuts).

Programmers can define their own pointcut designators by a name:

```
pointcut change(): call(void Package.setSize(double, double)) ||
                   call(void Package.setWeight(double))
```

This defines a pointcut named change that designates calls to methods that changes Package's dimension or weight.

Note that the name of the pointcut is at the left of the colon. Also, a pointcut can be declared with any Java access modifier such as **public** or **private** as illustrated bellow:

Table 2.1: Examples of method signatures that can be used in pointcut designators.

Signature Pattern	Matched Methods
public void Package.set*(*)	Any public method in the Package class with the name starting with set that returns void and takes a single argument of any type.
public void Package.*()	Any public method in the Package class that returns void and takes no arguments.
public * Package.*()	Any public method in the Package class that takes no arguments and return any type.
public * Package.*(..)	Any public method in the Package class that takes any number (including zero) and type of arguments and returns any type
* Package.*(..)	Any method in the Package class. This matches methods with public, protected, private, and default access.
* *.*(..)	Any method regardless or return type, defining type, method name, and arguments.
public Package.new()	A public constructor of the Package class taking no arguments.
public Package.new(..)	A public constructor of the Package class taking any number and type of arguments.

```
public pointcut change(): call(void Package.setSize(double, double)) ||
                          call(void Package.setWeight(double))
```

```
private pointcut setSizeCall(): call(void Package.setSize(double, double))
```

The general form of a named pointcut syntax is as follows:

```
[Privacy] pointcut Pointcut-Ident ([Args]) : Pointcut-Expression
```

Named and anonymous pointcuts

In AspectJ, pointcuts can be either anonymous or named. Named pointcuts are declarations that can be referenced/reused from several places. Anonymous pointcuts, on the other hand, cannot be reused. In order to see how anonymous pointcuts are used in practice, please see Section 2.5.4.

Property-based primitive pointcut designators

The previous pointcut designators are all based on the explicit enumeration of a set of method signatures (named-based primitive pointcut designators). AspectJ also provides an expressive way to select join points in terms of method properties instead of its fully qualified-signature. As an example, consider the following pointcut definition:

```
call(void Package.set*(..))
```

It identifies calls to any **void** method defined in Package, whose name begins with "set" like setSize and setWeight. More options of method signature patterns that can be used within pointcuts are illustrated in Table 2.1.

Table 2.2: Examples of other useful AspectJ pointcut designators.

Pointcut	**Description**
args(double, double)	Any method or constructor join point where the first and second arguments are of type double.
args(double, .., double)	Any method or constructor where the first and last arguments are of type double.
args(double, *)	Any method or constructor where the first argument is of type double and the second is of any type.
args(..)	Any method or constructor that takes any number (including zero) and type of arguments.
within(Package)	Any join point inside the Package class's lexical scope including inside any nested classes.
within(Package+)	Any join point inside the lexical scope of the Package class and its subclasses including inside any nested classes.
within(*)	Any join point inside the lexical scope of any type and its subclasses including inside any nested classes.
withincode(* Package.setSize(..))	Any join point inside the lexical scope of any setSize() method of the Package class.
this(Package)	Any join point where this instanceof Package evaluates to true. This selects all join points, such as methods calls where the current execution object is Package or one of its subtypes, like GiftPackage.
target(Package)	Any join point where the object on which the method is being called is instanceof Package!. This selects all join points such as method calls, where the target object is Package or its subtypes like GiftPackage.
cflow(call(* Package.setSize(..)))	All join points in the control flow of call of any setSize() method in Package, including the execution of the setSize method itself.
cflowbelow(call(* Package.setSize(..)))	All join points in the control flow of call of any setSize() method in Package, but excluding the execution of the setSize method itself.
@annotation(AnnotationType)	Any join point inside the lexical scope of any type marked with the AnnotationType annotation including nested classes.

Exposing context in pointcuts

In AspectJ, pointcut designators can also be used to expose part of the context associated with the intercepted join points. In the following code, the pointcut setSizeCall exposes three values from calls to setSize: the target object denoted by the **target** pointcut designator, and the two **double** arguments width and height denoted by the **args** pointcut designator.

```
pointcut setSizeCall(Package p, dobule w, double h):
    call(void Package.setSize(double, double))
   && target(p) && args(w, h);
```

In Table 2.2, we describe both **args** and **target** pointcuts, among other useful pointcut designators in AspectJ.

Matching static methods

AspectJ pointcuts also select static methods, but trying to match a call to a static method using a combination of **call** and **target** pointcuts (or **execution** and **this** pointcuts) will fail. The problem is that there is no target object in this case; the **target** designator becomes unbound. What is needed is just referring to the name of the declaring type in **call** or **execution** pointcut. For example, the following pointcut

```
pointcut pc(): execution(* Package.*(..));
```

will match both static and instance methods in Package class. If only static methods are wanted to be matched, the following refined pointcut

```
pointcut pc(): execution(static * Package.*(..));
```

will do that. Note the **static** modifier used within the **execution** pointcut. This serves to filer the join points to be static methods in Package class.

If we want to exclude static methods, we can use the unary operator ! to negate static methods. Hence, the following pointcut

```
pointcut pc(): execution(!static * Package.*(..));
```

will exclude static methods from being selected.

2.5.4 Advice

The pointcuts discussed so far are useful to pick out join points. But they do not perform anything else besides selecting join points. In order to actually implement crosscutting behavior, AspectJ uses a special construct called *advice*. AspectJ has several different kinds of advice that define additional code that should run at join points. Each form of advice declaration follows the same basic structure:

```
[strictfp]Advice-Specification [throws TypeList] : Pointcut-Expression {
  body of advice
}
```

In AspectJ advice, no other modifier besides **strictfp** is allowed. For example, visibility modifiers are not allowed since advice cannot be called explicitly. The optional **throws** clause indicates the exceptions that the body of the advice may throw. Advice may only throw exceptions compatible with the join points which are advised. For instance, advice cannot throw a checked exception that the client executing at the join point is not expecting.

In the following we discuss each kind of AspectJ advice. The examples will be carried out by design by contract. DbC is one of the recurring crosscutting concerns that AspectJ can modularize and deal with [KHH+01].

Before advice

Before advice executes before the action of the advised join points. So, if the before advice is associated with a **call** pointcut, then the advice runs just before the "call" of the advised join points. For an **execution** pointcut, the advice takes action just before the "execution" of the advised join points.

As an example, recall the preconditions for the method setSize (see Figure 2.4).

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
public void setSize(double width, double height) {...}
```

Consider further the following pointcut and before advice declarations in AspectJ used to check the same above preconditions:

```
pointcut setSizePre(double w, double h):
         execution(void Package.setSize(double, double)) && args(w, h);
before(double w, double h): setSizePre(w, h){
  boolean pred = (w > 0 && h > 0) && (w * h <= 400);
  Checker.Precondition(pred);
}
```

This before advice intercepts executions of method setSize and checks its preconditions before its execution. If we take a closer look at Figure 2.4, we can observe that besides setSize, the methods reSize and containsSize have the same preconditions. Although JML is an expressive DbC language with constructs to offer modularity opportunities to deal with contracts, such recurrent preconditions cannot be written only once and applied to these three methods. In AspectJ, the above pair of before advice and pointcut can be rewritten to do this job as follows:

```
pointcut sizes(double w, double h):
         execution(void Package.*Size(double, double)) && args(w, h);
before(double w, double h): sizes(w, h){
  boolean pred = (w > 0 && h > 0) && (w * h <= 400);
  Checker.Precondition(pred);
}
```

Now a pointcut named sizes selects all methods ending with “Size” and have two arguments of type **double**. This pointcut is used by the before advice in order to check the recurrent preconditions for setSize, reSize, and containsSize methods.

After returning advice

After returning advice executes after the successful completion of advised join points. In terms of DbC, such advice can be used to check JML normal postconditions. Recall the following normal postcondition, specification in JML, for the method setSize:

```
//@ ensures this.dimension.doubleValue() == width * height;
public void setSize(double width, double height) {...}
```

The following pointcut and after-returning advice declarations in AspectJ used to check the above normal postcondition specification:

```
pointcut setSizeNPost(Package obj, double w, double h):
         execution(void Package.setSize(double, double))
        && args(w, h) && this(obj);
after(Package obj, double w, double h) returning :
  setSizeNPost(obj, w, h){
   boolean pred = obj.dimension.doubleValue() == w * h;
   Checker.nPostcondition(pred);
}
```

This is the AspectJ code used to check the JML normal postcondition specification of method setSize. Note that if the method throws an exception, this advice does not execute. Also, if we need to use return value of and advised join point, AspectJ offers the following syntax:

```
after() returning (<ReturnType returnObjectIdent>)
```

This is useful for non-void methods. In addition, AspectJ uses this feature to check JML's result expression.

The method reSize contains the same normal postcondition. Thus, the following AspectJ pointcut-advice can be used to check the same normal postcondition specification for both setSize and reSize methods in Package.

```
pointcut sizeChange(Package obj, double w, double h):
         (execution(void Package.setSize(double, double))
         execution(void Package.reSize(double, double)))
         && args(w, h) && this(obj);
after(Package obj, double w, double h) returning :
  sizeChange(obj, w, h){
   boolean pred = obj.dimension.doubleValue() == w * h;
   Checker.nPostcondition(pred);
}
```

After throwing advice

AspectJ offers the use of after throwing advice to take action when the advised join point throws an exception. In terms of DbC, such advice can be used to check JML exceptional postconditions. Consider the following JML **signals_only** clause for setSize:

```
//@ signals_only \nothing;
public void setSize(double width, double height) {...}
```

The following AspectJ pair of pointcut and advice can be used to forbid the method setSize to throw exceptions like the above JML **signals_only** clause:

```
pointcut setSizeXPost(Package obj, double w, double h):
         execution(void Package.setSize(double, double))
         && args(w, h) && this(obj);
after(Package obj, double w, double h) throwing :
  setSizeXPost(obj, w, h){
   boolean pred = false;
   Checker.xPostcondition(pred);
}
```

Besides the method setSize, all methods in Package have this constraint about not throwing exceptions. Hence, in AspectJ, we can write just a single pair of pointcut-advice to enforce that:

```
pointcut allMeth(): execution(* Package.*(..));
after() throwing : allMeth(){
  boolean pred = false;
  Checker.xPostcondition(pred);
}
```

Note that if the method terminates normally without throwing any exception, this advice does not execute. Also, if needed, we can access the exception thrown; AspectJ offers the following syntax in order to do that:

```
after() throwing (<ExceptionType exceptionObjectIdent>)
```

To exemplify, consider the following JML exceptional postcondition for method setSize

```
//@ signals (SizeDimensionException) width * height > 400;
public void setSize(double width, double height)
  throws SizeDimensionException {...}
```

and its implementation using AspectJ:

```
pointcut setSizeXPost(Package obj, double w, double h):
        execution(void Package.setSize(double, double))
        && args(w, h) && this(obj);
after(Package obj, double w, double h)
   throwing (SizeDimensionException e) : setSizeXPost(obj, w, h){
   boolean pred = w * h > 400;
   Checker.xPostcondition(pred);
}
```

This advice will be executed only if the method setSize throws a SizeDimensionException; otherwise, the advice will not be executed.

Around advice

Around advice is the unique kind of advice that has the ability to surround the intercepted join points. That is, an around advice can bypass the execution of the intercepted join point completely, or to execute the join point with the same or different arguments and with the same or different behavior. In summary, typical usages of around advice are as follows:

- perform additional logic before and after the advised join point (e.g., pre- and postcondition checking);
- bypass the original operation and perform some alternative logic with the same or different arguments;
- surround the advice code with a **try/catch** block to perform an exception handling functionality.

With these features, we can observe that around advice is the most powerful form of advising that can always be used instead of before or after advice or even replacing both.

If within the around advice one want to execute the original behavior behind the intercepted join point, the use of a special keyword **proceed** in the body of the advice is needed. If the call to **proceed** is missing, the original behavior of the intercepted join point will be bypassed.

To illustrate the use of around advice, please consider first the following JML pre- and postconditions in JML for method setSize:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.dimension.doubleValue() == width * height;
public void setSize(double width, double height) {...}
```

In AspectJ, one can implement such pre- and (normal) postcondition in JML using the following code in AspectJ:

```
pointcut setSizePreNPost(Package obj, double w, double h):
          execution(void Package.setSize(double, double))
          && args(w, h) && this(obj);
void around(Package obj, double w, double h) :
    setSizePreNPost(obj, w, h){
     boolean prePred = w > 0 && h > 0;
     Checker.precondition(prePred);
     proceed(obj, w, h);
     boolean nPostPred = obj.dimension.doubleValue() == w * h;
     Checker.nPostcondition(nPostPred);
}
```

Note that the call to **proceed** method separates the code related to pre- and post-condition. The code before **proceed** denotes the precondition checking, whereas the code after **proceed** denotes normal postcondition checking, respectively.

Anonymous pointcut in advice

All pointcuts used so far are named, we can refer to them or reuse in other pointcuts. In AspectJ, we can use a second category of pointcuts that are unnamed or anonymous. Consider the following before advice using an anonymous pointcut:

```
before(double w, double h):
  execution(void Package.setSize(double, double)) && args(w, h){
   boolean pred = (w > 0 && h > 0) && (w * h <= 400);
   Checker.Precondition(pred);
}
```

In this before advice, there is no named pointcut referred within it. So if we write another advice that uses the same join points captured by the above **execution** pointcut, we need to write out them entirely again. The programmer is free to choose which kind of pointcuts to use, named or anonymous. But the former clearly offers maintainability advantages over the latter.

Ordering of advice

As observed, we can have multiple advice declarations intercepting the same set of join points. When this happens, AspectJ uses the following precedence rules to determine the order in which the advice is applied:

- the aspect with higher precedence executes its before advice intercepting the same set of join points before the aspect with lower precedence;
- the aspect with higher precedence executes its after advice intercepting the same set of join points after the aspect with lower precedence;

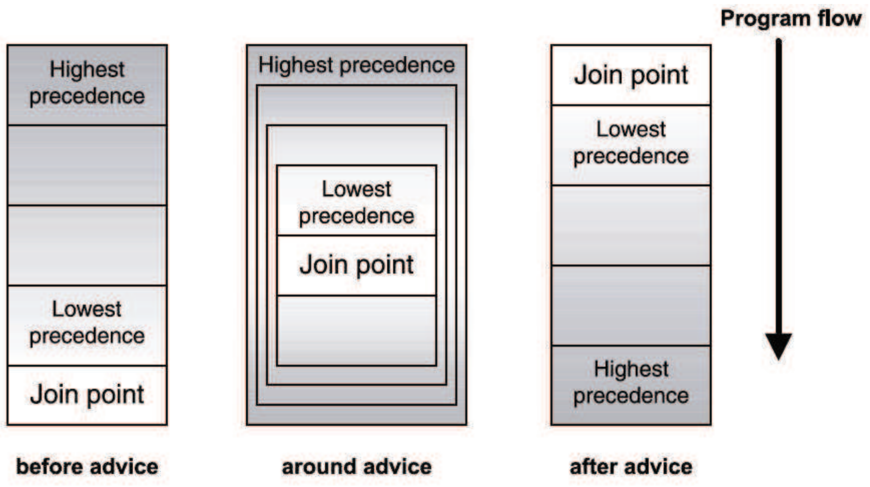


Figure 2.5: Ordering the execution of advice and join points. The darker areas represent the higher-precedence [Lad03].

- the around advice in the higher-precedence aspect encloses the around advice.

Figure 2.5 illustrates the precedence rules. When we have multiple advice in one aspect that apply to the same join points, the precedence is determined lexically. That is, before advice that appears first has the precedence, whereas the after advice that appears latter has the highest precedence. However, we can have multiple advice advising the same set of join points that are declared in different aspect declarations. In this case, AspectJ provides a special construct **declare precedence** for explicitly aspect precedence controlling. For instance, in the below **declare precedence**, the aspect A1 has a higher precedence over A2.

```
declare precedence : A1, A2;
```

2.5.5 Accessing Join Point Context via Reflection

AspectJ provides reflective access to join point context through three special constructs available in each advice body. The special variables are: **thisJoinPoint**, **thisJoinPointStaticPart**, and **thisEnclosingJoinPointStaticPart**. The constructs behave like the special variable **this** in plain Java. Through reflective access, one can get information such as the name of the current advised method, source location (line number). Also, we call methods or access fields of the object type owner of the advised method.

```
pointcut setSizePreNPost(double w, double h):
        execution(void Package.setSize(double, double)) && args(w, h);
void around(Package obj, double w, double h) :
   setSizePreNPost(obj, w, h){
    Package obj = (Package)thisJoinPoint.getThis();
    boolean prePred = w > 0 && h > 0;
    Checker.precondition(prePred);
    proceed(obj, w, h);
    boolean nPostPred = obj.dimension.doubleValue() == w * h;
    Checker.nPostcondition(nPostPred);
}
```

Instead of exposing the Package instance through parameterization as before, we access it now through the **thisJoinPoint** special variable.

For more information about the reflective API of AspectJ, please refer to [Lad03].

2.5.6 Static Crosscutting

Besides modifying the dynamic behavior of a program using advice, AspectJ can also affect the static structure of a program. This mechanism is known as *static crosscutting mechanism*. We quickly introduce some static crosscutting features here. For more information, please refer to [Lad03].

Member introduction

Member introduction, also known as *inter-type declaration*, is the ability to add members to an existing type without being invasive. Suppose we want to add the following method to type Package that is a part of the design by contract concern:

```
private void Package.precondition (boolean pred) {
  if(!pred){
    throw new JMLPreconditionError();
  }
}
```

Note that preceding the name of the method, we add the target type name which will be declared. Physically, this is located within any aspect declaration, but in practical terms, we can use such method just like if it was declared within type Package. AspectJ also supports field introduction.

Modifying the class hierarchy

With AspectJ, we can also change the hierarchy structure of a class. To this end, AspectJ provides the **declare parents** feature. Suppose that the GiftPackage in Figure 2.1 has no supertype (no **extends** clause). Suppose further that we want to make this type to be a subtype of Package, but without being invasive (changing the GiftPackage for that). Hence, we can use the following declaring **parents** to do the job:

```
declare parents: GiftPackage extends Package;
```

Introducing compile-time errors and warnings

AspectJ provides special constructs in a static crosscutting manner to declare compile-time errors and warnings. With this mechanism, one can implement behavior similar to the #error and #warning preprocessor directives supported in C/C++ language preprocessors.

The AspectJ **declare** error construct provides a way to declare a compile-time error. The error is issued when a given pointcut expression is satisfied. To exemplify this feature, let us consider that in Package class no methods can assign to fields except set-like methods of constructors. The following AspectJ **declare** error can implement this rule:

```
declare error: set(* Package.*) && !withincode(* Package.set*(..))
       && !withincode(Package.new(..)): "assignment forbidden!";
```

The pointcut within **declare** error states that any Package field assignment (denoted by the AspectJ set pointcut [Lad03]) that is not within a set-like method or constructor (expressed using **withincode** pointcut [Lad03]), should issue the compile-time error message “assignment forbidden”.

To issue a warning instead of a compile-time error, we just need to adapt the above declaration to **declare** warning. More details can be found in [Lad03].

2.5.7 Privileged aspects

The Java visibility rules are also applied to aspects. However, there are situations where an aspect needs to access private or protected members of types. Hence, to allow this kind of access, the AspectJ aspects must be declared privileged. In order to illustrate that, consider the following aspect declaration accessing the **protected** fields width and height of Package class (see Figure 2.4):

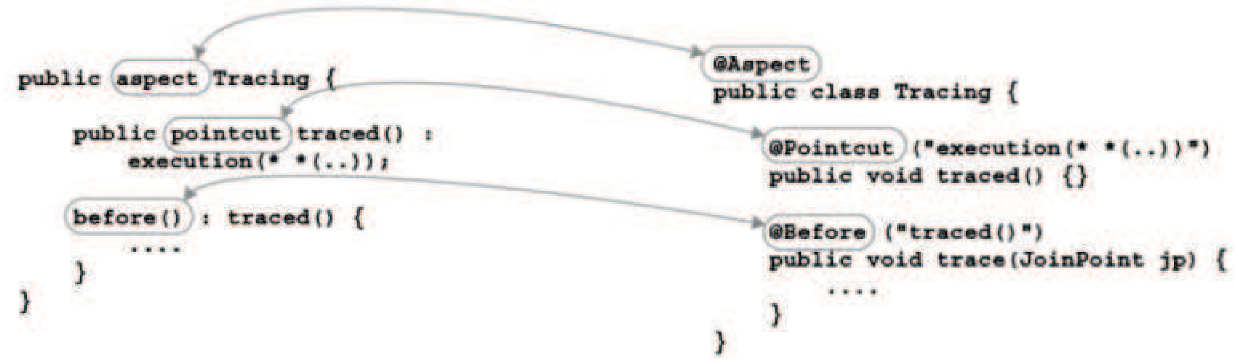
```
privileged public aspect PrivateAccess {
 pointcut sizeChange(Package obj, double w, double h):
          (execution(void Package.setSize(double, double))
          execution(void Package.reSize(double, double)))
          && args(w, h) && this(obj);
 after(Package obj, double w, double h) returning :
  sizeChange(obj, w, h){
   boolean pred = (obj.width == w) && (obj.height == h);
   Checker.nPostcondition(pred);
 }
}
```

The above privileged aspect can access the fields width and height, even though they are protected members of the Package class. The used `after` advice checks if such fields are equal to those values passed as arguments (which are exposed by the advice).

It is important to note that privileged aspects can even access private members.

2.5.8 @AspectJ

In relation to AspectJ, what is new today is the presence of the @AspectJ alternative syntax. The @AspectJ (often pronounced as “at AspectJ”) syntax was conceived as a part of the merge of standard AspectJ with AspectWerkz [Bon]. This merge enables crosscutting concern implementation by using constructs based on the metadata annotation facility of Java 5. The main advantage of this syntactic style is that one can compile a program using a plain Java compiler, allowing the modularized code using AspectJ to work better with conventional Java IDEs and other tools that do not understand the traditional AspectJ syntax. The Spring Framework is an example that heavily uses the alternative AspectJ syntax [Lad09].



```
public aspect Tracing {

    public pointcut traced() :
        execution(* *(..));

    before() : traced() {
        ....
    }
}
```

```
@Aspect
public class Tracing {

    @Pointcut ("execution(* *(..))")
    public void traced() {}

    @Before ("traced()")
    public void trace(JoinPoint jp) {
        ....
    }
}
```

Figure 2.6: The mapping between the traditional and @AspectJ syntax [Lad09].

Figure 2.6 illustrates the general mapping between the traditional and the @AspectJ syntax. The general idea behind the design of the @AspectJ syntax is to find a suitable Java counterpart and annotate it to express the traditional AspectJ feature. For instance, instead of using the **aspect** keyword, we use an ordinary Java class annotated with an **@Aspect** annotation. This tells the AspectJ/ ajc compiler to treat the class as an aspect declaration. Similarly, the **@Pointcut** annotation marks the empty method traced as a pointcut declaration. The expression specified in this pointcut is the same as the one used in the standard AspectJ syntax. The name of the method serves as the pointcut name. Finally, the **@Before** annotation marks the method trace as a **before** advice. The body of the method is used to modularize the crosscutting concern/advising code. This code is executed just before the matched join point's execution.

Note that the method body for a concrete pointcut is empty, because the method is just a placeholder, for the **@Pointcut** annotation, without any significance for the code inside it. Therefore, any attempt to write a body for a pointcut method in @AspectJ style raises a compile-time error.

2.5.9 Exported AspectJ example

Figure 2.7 presents the AspectJ code, using the feature discussed in this section, for implementing the crosscutting contracts (identified in this section) illustrated in Figure 2.4).

2.5.10 AspectJ Compilers/Weavers

This section briefly presents the two well-known AspectJ compilers used by the AspectJ community (academic and industry).

ajc compiler

The ajc compiler is the standard (official) compiler of the AspectJ language [KHH$^+$01]. In early versions, ajc was a pre-compiler written in Java. It used to compile aspects and classes together and to produce Java source code. Such a Java code was usually referred as a "weave", then ajc used to invoke javac to actually compile the "weave" into bytecode. Nowadays, the ajc compiler generates Java bytecode directly from weaving.

```
public aspect PackagePreconditions {
 pointcut sizes(double w, double h):
         execution(void Package.*Size(double, double)) && args(w, h);
 before(double w, double h): sizes(w, h){
  boolean pred = (w > 0 && h > 0) && (w * h <= 400);
  Checker.Precondition(pred);
 }
}
public privileged aspect PackageNPostconditions {
 pointcut sizeChange(Package obj, double w, double h):
         (execution(void Package.setSize(double, double))
         execution(void Package.reSize(double, double)))
         && args(w, h) && this(obj);
 after(Package obj, double w, double h) returning :
  sizeChange(obj, w, h){
   boolean pred = obj.dimension.doubleValue() == w * h;
   Checker.nPostcondition(pred);
 }
}
public aspect PackageXPostconditions {
 pointcut allMeth(): execution(* Package.*(..));
 after() throwing : allMeth(){
  boolean pred = false;
  Checker.xPostcondition(pred);
 }
}
```

Figure 2.7: The AspectJ implementation for the JML crosscutting contracts presented in Figure 2.4.

The ajc compiler is also integrated in the Eclipse environment through the AJDT plugin [AJD14] (AspectJ Development Tools, as an extension of the JDT, Java Development tools).

Hilsdale and Hugunin [HH04] provide more details about how the ajc compiler weaves aspects together with classes.

abc compiler

The AspectBench Compiler (abc) is an academic compiler [ACH+05] that implements the full AspectJ language [KHH+01]. The compiler was conceived as a workbench to facilitate easy experimentation with new language features and implementation techniques. In particular, new features for AspectJ have been proposed that require extensions in many dimensions: syntax, type checking and code generation, as well as data flow and control flow analyses. Experiments conducted in this dissertation demonstrated that abc produces code of better quality if compared with the ajc compiler.

For more details about the abc compiler please refer to [ACH+05].

2.6 Chapter Summary

In this chapter, we presented the main concepts related to this thesis. As presented, JML is a formal behavioral interface specification language conceived to Java. As with Eiffel [Mey92b], JML employs the design by contract style to the Java programming language. JML supports a variety of features like method and type specifications, specification inheritance and syntax for visibility modifiers to achieve information hiding in pre- and postcondition specifications. Afterwards, we discussed the aspect-oriented programming concepts using the most well-known AOP language called AspectJ. We showed how AspectJ tackles the crosscutting concern modularization problem using by means of pointcuts and advice. We discussed in practice how AspectJ can be used to check DbC features like pre- and postconditions. Also, we discussed the new AspectJ syntax that is based on Java 5 annotations. This new syntax is called @AspectJ.

Chapter 3

Design by Contract and Modularity Problems

In this chapter we discuss the existing modularity problems when using design by contract in practice. To this end, we use the following criteria:

- modular reasoning;
- documentation;
- crosscutting contract specification;
- information hiding and RAC.

We use this criteria to analyze DbC/JML [LBR06] and AOP/AspectJ [KLM+97, KHH+01] languages in terms of design by contract and modularity. This criteria is extracted from the works [Ste06, BRLM11, BEM05, Par72, Par11].

In the following, we discuss the main modularity problems related to the design by contract languages and using the criteria listed above.

3.1 The Modular Reasoning Criterion

If we consider plain JML/Java without AspectJ, the example in Figure 2.4 supports modular reasoning [Mey00, Lea06, LN13, RLL+13b] (see our modular reasoning definitions 2.1.1 and 2.1.2). For example, suppose one wants to write code that manipulates objects of type Package. One could reason about Package objects using just Package's contract specifications in addition to ones inherited (expanded modular reasoning) from its supertypes (if any) [DL96, Lea06, LN13].

Consider now the AspectJ crosscutting contract implementation of the Package class (see Figure 2.7). In plain AspectJ, advice declarations are applied by the compiler without explicit reference to aspects from a module or a client module; therefore by definition, modular reasoning about the Package module does not consider any of the advice declarations in the PackageContracts aspect. The aspect behavior is only available via non-modular reasoning or global reasoning (see our Definition 2.1.3). That is, in AspectJ, a programmer must consider every aspect that refers to the Package class in order to reason about the Package module. So the answer to the question "Which

```
aspect Tracing {
 void around(): execution(void Package.*(..)) {
  proceed();
  System.out.println("Exiting"+thisJoinPoint);
 }
}
```

Figure 3.1: The tracing crosscutting concern implementation in AspectJ.

advice/contract applies to the method setSize or setWeight in Package?" cannot (in general) be answered modularly. Therefore, a programmer cannot study the system one module at a time [Par72, BEM05, ITB11, SGR$^+$10, BRLM11, RLL$^+$13b].

3.2 The Documentation Criterion

In a design by contract language like JML, the pre- and postconditions and invariant specifications are typically placed directly in or next to the code they are specifying. Hence, contracts increase system documentation [Mey00, BEM05, Par11]. In AspectJ, however, the advising code (that checks contracts) is separated from the code it advises and this forces programmers to consider all aspects in order to understand the correctness of a particular method. In addition, the physical separation of contracts can be harmful in the sense that an oblivious programmer can break a method's pre- or postconditions when these are only recorded, through syntactic constructs, in aspects [Mey00, BEM05, Par11].

Consider now another crosscutting concern, say tracing. Figure 3.1 shows the modularization of tracing by the aspect Tracing. It prints a message after the successful execution of any **void** method in the Package class (see Figure 2.4) when called. For this concern, different orders of composition with other aspects (that check contracts) lead to different behaviors/outputs. Hence, the **around** advice (Figure 3.1) could execute either before or after the **after−returning** or **after−throwing** advice we have in Figure 2.7. Without either documentation or the use of AspectJ's **declare precedence** [KHH$^+$01] to enforce a specific order on aspects, it is quite difficult–perhaps impossible–to understand the order in which pre- and postconditions will be executed until they are actually executed.

Another problem caused by the lack of documentation implied by separating contracts as aspects is discussed by Balzer, Eugster, Meyer's work [BEM05]. They argue that programmers become aware of contracts only when using special tools like AJDT [KM05]; they are more likely to forget to account for the contracts when changing the classes.

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400;
public void setSize(..){...}

//@ requires width > 0 && height > 0;
//@ requires width * height <= 400;
public void reSize(..){...}

//@ requires width > 0 && height > 0;
//@ requires width * height <= 400;
public boolean containsSize(..){...}
```

(1)

```
//@ signals_only \nothing;
public void setSize(..){...}

//@ signals_only \nothing;
public void reSize(..){...}

//@ signals_only \nothing;
public boolean containsSize(..){...}

//@ signals_only \nothing;
public double getSize(){...}

//@ signals_only \nothing
public void setWeight(..) {...}
```

(3)

```
//@ ensures this.dimension.doubleValue() == width * height;
public void setSize(..){...}

//@ ensures this.dimension.doubleValue() == width * height;
public void reSize(..){...}
```

(2)

Figure 3.2: Crosscutting contracts in Package's specifications (see Figure 2.4). Scenario (1) illustrates the crosscutting preconditions, (2) illustrates the crosscutting normal postconditions, and (3) the crosscutting exceptional postconditions.

3.3 The Crosscutting Contract Specification Criterion

Balzer, Eugster, and Meyer's study [BEM05] helped to crystallize our thinking about the goals of a DbC language and about the parts of such languages that provide good documentation, modular reasoning, and contracts in general without obliviousness. One straightforward way to avoid the previous problems discussed is to use a plain DbC language like JML without AspectJ. As discussed in Section 2.4.10, a DbC language like JML can be used to modularize some contracts. For example, the invariant clauses (declared in Package) can be viewed as a form of built-in modularization. That is, instead of writing the same pre- and postconditions for all methods in a type and its subtypes, we declare a single invariant that modularizes those pre- and postconditions. Also, specification inheritance is another form of modularization. In JML, an overriding method inherits method contracts and invariants from the methods it overrides (see Sections 2.4.8 and 2.4.10 for more details).

However, DbC languages (like JML) do not capture all forms of crosscutting contract structure [KHH$^+$01, HK02] that can arise in specifications. For example, consider the JML specifications illustrated in Figure 2.4. In this example there are three scenarios (summarized in Figure 3.2) in which crosscutting contracts are not properly modularized with plain JML constructs:

(1) we cannot write preconditions constraining the input parameters on the methods setSize, reSize, and containsSize (in Package) to be greater than zero and less than or equal to 400 (the package dimension) only once and apply them to these or other

methods with the same design constraint;

(2) the two normal postconditions of the methods setSize and reSize of Package are the same. They ensure that the dimension model field is equal to the multiplication of the method argument width with argument height; however, we cannot write a simple and local quantified form of these postconditions and apply them to the constrained methods; and

(3) the exceptional postcondition **signals_only** **nothing** must be explicitly written for all methods in Package which forbid exceptions; there is no way to modularize such a JML contract in one place and apply it to all constrained methods.

In relation to AspectJ, as discussed in Section 2.5 and observed in Figure 2.7, it supports crosscutting contract implementation in a modular way. By means of pointcuts and advice combined with quantification, we can apply a crosscutting contract to several join points.

3.4 The Information Hiding and RAC Criterion

Although there is a set of rules for information hiding in specifications [LM07] and although JML [LBR06] supports visibility modifiers, existing runtime assertion checker (RAC) tools (e.g., Eiffel's RAC [Mey92b]) violate those information hiding rules during runtime assertion checking and error reporting.

The problem is the way contemporary RACs implement runtime assertion checks for method calls. Such RAC compilers operate by injecting code at the supplier side, thus checking each method's precondition at the beginning of its code, and injecting code to check the method's postcondition at the end of its code. Since supplier-side checks do not take into account what kind of client (i.e., privileged or non-privileged) is calling a method, a RAC checks all the specifications regardless of visibility. Therefore, we say that a RAC that checks all the specifications at supplier side as *overly-dynamic*. For example, consider the following specification cases for method setSize in Package (we refined the specs for setSize shown in Figure 2.4 to add a protected specification case):

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.dimension.doubleValue() == width * height;
//@ signals_only \nothing;
//@ also
//@ protected behavior
//@ requires width > 0 && height > 0;
//@ requires width * height <= 600; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
public void setSize(double width, double height){...}
```

If we use the standard JML compiler (jmlc) to compile these specification cases for setSize method to runtime checks, we have the following code instrumentation:

```
public void setSize(double width, double height) {
  checkPre$setSize$Package(width, height))
  this.width = width;
  this.height = height;
  checkNPost$setSize$Package(width, height))
}
public void checkPre$setSize$Package(double width, double height){
   boolean pred = ((width > 0 && height > 0) && (width * height <= 400))
                || ((width > 0 && height > 0) && (width * height <= 600));
   if(!pred)
    throw new JMLEntryPreconditionError();
}
public void checkNPost$setSize$Package(double width, double height) {
 // check NPost when corresponding precondition hold
  if((width > 0 && height > 0) && (width * height <= 400)){
   boolean pred = this.dimension.doubleValue() == width * height;
   if(!pred)
    throw new JMLExitNormalPostconditionError();
  }
  if((width > 0 && height > 0) && (width * height <= 600)){
   boolean pred = (this.width == width) && (this.height == height);
   if(!pred)
    throw new JMLExitNormalPostconditionError();
  }
}
```

As mentioned, due to the overly-dynamic checking, in this code instrumentation, all the specification cases are checked during runtime regardless of visibility rules.

According to the rules discussed in Section 2.4.6, the first specification case is of kind lightweight and the visibility is the same as the method itself; public in this case. The second specification case is of kind heavyweight with a protected visibility, hence it is allowed to mention protected fields in **ensures** clauses. This is intended for privileged clients such as subtypes. Also, the protected specification case weakens the precondition to handle bigger packages until 600cm^2 as a maximum dimension. Of course subtypes of Packages can refine this inherited precondition to handle even bigger packages, but by default any Package's subtype can at least handle packages until 600cm^2 as the maximum dimension.

Consider now the following public client code:

```
public class PackageClient {
   public void setSizePackageClient(Package p) {
      p.setSize(200, 2);
   }
}
```

Consider further that the implementation of setSize mistakenly increments the width by 1. In this scenario, we got the following normal postcondition error in the classical JML RAC, when using the above JML specifications for setSize:

```
Exception in thread "main"
```

```
org.jmlspecs.ajmlrac.runtime.JMLExitNormalPostconditionError:
by method Package.setSize regarding specifications at
File "Package.java", line 23 (Package.java:23), when
   'this.width' is 201.0
   'this.height' is 2.0
   'this.dimension' is 402.0
   'width' is 200.0
   'height' is 2
   ...
```

As can be seen, in this error output, the protected fields width and height are mentioned (see the shadowed part). However, these fields are not meaningful to public clients, as they are not visible. So, supplier side checking in JML makes it difficult to implement client-oriented error reporting that only shows clients error reports that make sense at the call site. This problem also occurs in other specification languages like Eiffel [Mey92b], Spec# [BLS05], and Code Contracts [FBL10].

Indeed, another problem is that some expected violations are missed due to the overly-dynamic checking. According to Leavens and Müller's rules [LM07], only the public specification case should be checked against the method call p.setSize(200,2) in a public client. As such, suppose now that we do not have the implementation mistake anymore in setSize and that the public call has the following inputs:

```
 p.setSize(300, 2);
```

This leads to no violation at all in JML using an overly-dynamic checking. The problem is that the hidden protected precondition which handles larger packages hold, then no precondition violation is raised to a $600cm^2$ package's dimension. But the protected precondition was designed for privileged clients such as subytpes. This way only the public precondition should be checked against the above method call. Hence, a precondition violation should be raised by doing that, but it was neglected by the nature that JML/RAC checks specifications during runtime.

The problem is evident when we change the call to

```
 p.setSize(301, 2);
```

and we got the following precondition error:

```
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLExitNormalPostconditionError:
by method Package.setSize regarding specifications at
File "Package.java", [spec-case]: line 15 (Package.java:15),
line 16 (Package.java:16), and [spec-case]: line 21 (Package.java:21),
line 22 (Package.java:22), when
   'width' is 301.0
   'height' is 2
   ...
```

The four shadowed lines in the error reporting denotes the two public **requires** clauses and the two protected **requires** clauses, respectively.

In relation to AspectJ it is possible to check contracts respecting information hiding. For example, consider the following AspectJ advice to check the protected normal postcondition previously discussed:

```
after (p.Package obj, double width, double height) returning():
 call(void p.Package.setSize(double, double))
 && within(p.*) || within(p.Package+)
 && target(obj) && args(width, height) {
 if((width > 0 && height > 0) && (width * height <= 600)){
  boolean pred = (this.width == width) && (this.height == height);
  JMLChecker.checkNormalPostcondition(pred, "errorMsg");
 }
}
```

The main issue is the use of the **within** pointcut. The shadowed part illustrates the protected visibility enforcement. That is, the **within** pointcut restricts where the **after returning** advice will be checked; it will be executed **iff** clients refer to type Package within package p or subtypes of Package. For the public normal postcondition, we have a similar advice (not shown) but without the **within** pointcut. This ensures that the postcondition will be checked to any call to setSize method regardless the context of visibility.

In the next chapter, we discuss how our AspectJML DbC language combined with a technique called *client-aware checking* enables the runtime checking in conformance with information hiding. Since the instrumented code with runtime checks is injected at the site of each method call, it properly checks only the visible specifications associated with the clients.

3.5 Chapter Summary

In this chapter, we presented the main modularity problems with existing design by contract and aspect-oriented languages. So, for AspectJ-like languages, although they can physically modularize crosscutting contracts or preserve information hiding during runtime, they are not suitable to ensure key DbC properties like modular reasoning and documentation. On the other hand, a DbC language like JML can offers good documentation and provide modular reasoning. However, such languages are not suitable to deal with the crosscutting structure nature of contracts. This situation leads to a dilemma/trade-off between AOP and DbC. Another problem discussed here is that existing DbC tools do violate information hiding principles during runtime assertion checking and error reporting. This modularity analysis is summarized in Table 3.1.

Table 3.1: Summary of the analysis of modularity for DbC and AOP.

Criterion	**JML**	**AspectJ/@AspectJ**
modular reasoning	√	x
documentation	√	x
crosscutting contract specification	x	√
information hiding and RAC	x	√

Chapter 4

The AspectJML Language

In this chapter, we present AspectJML [RLB+14], a simple and practical aspect-oriented extension to JML. We describe how the AspectJML features can be used to overcome all the modularity problems discussed in the previous chapter.

4.1 Overview

AspectJML extends JML [LBR06] with support for crosscutting contracts [MMvD05]. It allows programmers to define additional constructs (in addition to those of JML) to modularly specify pre- and postconditions and check them at certain well-defined points in the execution of a program. We call this the *crosscutting contract specification* mechanism, or XCS for short.

AspectJML also enables runtime checking on client side. We call this feature as *client-aware checking*, or CAC for short. AspectJML with CAC respects information hiding while checking contracts during runtime.

In the following, we present the main features of our AspectJML language. The presentation is informal and running-example-based.

4.2 Design Decisions

In this section, we discuss the main design decisions of AspectJML.

4.2.1 AspectJML is DbC + Quantification

As we discussed in Chapter 3, DbC languages like JML and AOP languages like AspectJ both have advantages and disadvantages when employed for specification and runtime checking of contracts. The main discussion is that neither can achieve crosscutting contract modularity nor the main DbC principles at the same time such as modular reasoning and documentation.

In this context, to tackle this dilemma, we conceive the AspectJML language that combines the advantages of both DbC and AOP techniques. Hence, all the benefits of JML are kept as usual. The key difference is the addition of the AspectJ's pointcut language in AspectJML. We decided to include only this portion of an AspectJ-like language because it is the main core of AOP. The core of an AspectJ-like language is the quantification mechanism [FF00, VCFS10] that is responsible for selecting all the join points of interest that are part of the crosscutting concern structure (DbC in our case). Hence, we added in AspectJML this quantification mechanism of AspectJ, which is represented by the pointcut language.

Therefore to be precise, the formula that AspectJML uses to provide crosscutting contract modularity and keep documentation and modular reasoning is "DbC + Quantification". This formula is achieved by the AspectJML's XCS feature.

4.2.2 @AspectJ

XCS in AspectJML is based on a subset of AspectJ's constructs [KHH$^+$01]. However, since JML is a design by contract language tailored for plain Java, we would need special support to use the traditional AspectJ syntax. To simplify the adoption of AspectJML, we employ AspectJ constructs based on the alternative @AspectJ syntax [Bon]. Thus, the AspectJML compiler, called ajmlc [BCC$^+$05, RSL$^+$08, RLL$^+$13c], needs only to process standard Java features. We already discussed @AspectJ syntax in Section 2.5.8.

4.2.3 AspectJML Supported Features

For the language scope, AspectJML supports all the JML features (excluding inline assertions[1]) presented in Section 2.4. In relation to AspectJ, we already mentioned that AspectJML only inherits the pointcut language. For crosscutting contract specification and runtime checking purposes, the pointcut designators that AspectJML supports are **call** and **execution**. They are expressive enough to handle crosscutting contracts. Other useful pointcuts that can be used together with **call** and **execution** are illustrated in Table 2.2. With respect to this table, we exclude the pointcut designators **withincode**, **cflow**, **cflowbelow**, **this**, and **target**. The first three we exclude because there is no example until now that one really needs it to perform contract modularization. The last two are forbidden to use. The compiler will complain if a programmer tries to use the pointcut **this** or **target**. This is to avoid the problem of obliviousness and is discussed in more detail in Section 4.4. AspectJML also supports the use of the special AspectJ variable **thisJoinPoint**. In rest of this chapter, we discuss a special scenario where **thisJoinPoint** is useful for crosscutting contract specification.

4.2.4 AspectJML Compatibility

One of the goals of this work is to support a substantial user community. To make this concrete, we have chosen to design crosscutting contract specification in AspectJML as a compatible extension to JML using AspectJ's pointcut language. This takes advantage of AspectJ's familiarity among programmers. Our goal is to make programming and specifying with AspectJML feel like a natural extension of programming and specifying with Java and JML. The AspectJML/ajmlc compiler has the following properties:

- all legal JML annotated Java programs are legal AspectJML programs;

[1]Since we use AOP/AspectJ to instrument JML contracts we cannot intercept and instrument method's local variables related to inline assertions. Hence, this is a well-known limitation of AspectJML/ajmlc compiler.

- all legal AspectJ/@AspectJ programs are legal AspectJML programs;
- all legal Java programs are legal AspectJML programs; and
- all legal AspectJML programs run on standard Java virtual machines.

4.3 XCS in Action

In AspectJML only two mechanisms are necessary to modularize crosscutting contracts at the source code level. Recall that a *pointcut designator* enables one to select well-defined points in a program's execution, which are known as *join points* [KHH+01]. Optionally, a pointcut can also include some of the values in the execution context of intercepted join points. In AspectJML, we can compose these AspectJ pointcuts combined with JML specifications. Hence, pointcuts and specifications are the two basic mechanisms needed in AspectJML to specify crosscutting contracts in a modular way.

The major difference, in relation to plain AspectJ, is that a specified pointcut is always processed when using the AspectJML compiler (ajmlc). In standard AspectJ, a single pointcut declaration does not contribute to the execution flow of a program unless we define some AspectJ advice that uses such a pointcut. In AspectJML, we do not need to define an advice to check a specification in a crosscutting fashion. This makes AspectJML simpler and a programmer only needs to know AspectJ's pointcut language in addition to the main JML features supported.

4.3.1 XCS Syntax

This is the AspectJML's program syntax for crosscutting contract specification. Note that both JML and @AspectJ syntax were discussed in Chapter 2.

```
Method-Specification ::= Specification
Specification ::= Spec-Case [also Spec-Case] ...
Spec-Case ::= Lightweight-Spec-Case | Heavyweight-Spec-Case
Lightweight-Spec-Case ::= Generic-Spec-Clause
Heavyweight-Spec-Case ::= Behavior-Spec-Case
      | Normal-Behavior-Spec-Case
      | Exceptional-Behavior-Spec-Case
Behavior-Spec-Case ::= [privacy] Behavior-Keyword
                            Generic-Spec-Clause
Behavior-Keyword ::= behavior | behaviour
Normal-Behavior-Spec-Case ::= [privacy] Normal-Behavior-Keyword
                                    Generic-Spec-Clause
Normal-Behavior-Keyword ::= normal_behavior | normal_behaviour
Exceptional-Behavior-Spec-Case ::= [privacy] Exceptional-Behavior-Keyword
                                    Generic-Spec-Clause
Exceptional-Behavior-Keyword ::= exceptional_behavior | exceptional_behaviour
Generic-Spec-Case ::= Requires-Clause [Requires-Clause] ...
      | Ensures-Clause [Ensures-Clause] ...
      | Signals-Clause [Signals-Clause] ...
      | Signals-Only-Clause [Signals-Only-Clause] ...
Requires-Clause ::= Requires-Keyword [!] Pred ;
      | Requires-Keyword \same;
Requires-Keyword ::= requires | pre
Pred ::= Predicate | \not_specified
Signals-Clause ::= Signals-Keyword (reference-type [ident]) [!] Pred ;
Signals-Keyword ::= signals | exsures
Signals-Only-Clause ::= Signals-Only-Keyword Reference-Type [, Reference-Type] ... ;
       | Signals-Only-Keyword \nothing ;
```

```
Signals − Only − Keyword ::= signals_only
@Pointcut("Pointcut − Expression")
[Privacy] void Pointcut − Ident ([Args]) {}
```

A more complete description of the AspectJML features and syntax can be found in Appendix A.

4.3.2 Specifying Crosscutting Preconditions

Recall our first crosscutting contract scenario described in Section 3.3. It consists of two preconditions for any method, in Package (Figure 2.4) with a name ending with Size that returns **void** and takes two arguments of type **double**. For this scenario, consider the JML annotated pointcut with the following preconditions:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
@Pointcut("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
void sizes(double width, double height) {}
```

The pointcut sizes matches all the executions of methods ending with “Size” of class Package like setSize and reSize. As observed, this pointcut is exposing the intercepted method arguments of type **double**. This is done in @AspectJ by listing the formal parameters in the pointcut method. We bind the parameter names in the pointcut's expression (within the annotation **@Pointcut** [Bon]) using the argument-based pointcut **args** [KHH+01].

The main difference between this pointcut declaration and the standard pointcut declarations in @AspectJ is that we are specifying two JML preconditions (using the **requires** clause). In this example the JML says to check the declared preconditions before the executions of intercepted methods.

4.3.3 Specifying Crosscutting Postconditions

We discuss now how to properly modularize crosscutting postconditions in AspectJML. JML supports two kinds of postconditions: normal and exceptional. Normal postconditions constrain methods that return without throwing an exception. To illustrate AspectJML's design, we discuss scenarios (2) and (3) from Section 3.3. For scenario (2), we use the following specified pointcut:

```
//@ ensures this.dimension.doubleValue() == width * height;
@Pointcut("(execution(* Package.setSize(double, double))"+
 "|| execution(* Package.reSize(double, double)))"+
 "&& args(width, height)")
public void setOrReSize(double width, double height) {}
```

This pointcut constrains the executions of the setSize and reSize methods in Package to ensure that, after their executions, the model field dimension is equal to the multiplication of arguments width and height. Recall that this model field is a representation of the protected fields width and height depicted in Figure 2.4. To modularize the crosscutting postcondition of scenario (3), we use the following JML annotated pointcut declaration:

```
//@ signals_only \nothing;
@Pointcut("execution(* Package+.*(..))")
public void allMeth() {}
```

The above specification forbids the executions of any method in Package (or in subtypes, such as GiftPackage) to throw an exception. If any intercepted method throws an exception (even a runtime exception), a JML exceptional postcondition error is thrown to signal the contract violation. In this pointcut, we do not expose any intercepted method's context.

4.3.4 Multiple Specifications Per Pointcut

All the crosscutting contract specifications discussed above consist of only one kind of JML specification per pointcut declaration. However, AspectJML can include more than one kind of JML specification in a pointcut declaration. For example, assume that the Package type in Figure 2.4 does not contain the containsSize method or its JML specifications. In this scenario, we can write a single pointcut to modularize the recurrent pre- and postconditions of methods setSize and reSize. Therefore, instead of having separate JML annotated pointcuts for each crosscutting contract, we specify them in a new version of the pointcut sizeMeths:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.dimension.doubleValue() == width * height;
@Pointcut("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
public void sizeMeths(double width, double height) {}
```

This pointcut declaration modularly specifies both preconditions and normal postconditions of the same intercepted size methods (setSize and reSize) of Package.

4.3.5 Pointcut Expressions Without Type Signature Patterns

In AspectJ, a pointcut expression can be defined without using a type signature pattern. A type signature pattern is a name (or part of a name) used to identify what type contains the join point. For example, the following AspectJ pointcut:

```
pointcut sizes(): execution(* *Size(double, double));
```

selects any method ending with "Size" and has two arguments of type **double**. In AspectJ, this pointcut matches any type in a system. Since we omit the type signature pattern, any type is candidate to expose the join points of interest. In AspectJ, although not required, we can also use a wildcard (*) to represent a type signature pattern that intercepts any type in the system. The pointcut looks like as follow:

```
pointcut sizes2(): execution(* *.*Size(double, double));
```

However AspectJML has a different semantics compared with AspectJ. For example, recall the previous pointcut method sizes in AspectJML:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
```

```
@Pointcut("execution(* *Size(double, double))"+
 "&& args(width, height)")
void sizes(double width, double height) {}
```

this pointcut method still selects the same methods ending with "Size" and that has two arguments of type **double**. The main difference is that even with the absence of the target type, AspectJML restricts the join points to the type (Package in this case) enclosing the pointcut declaration (see Figure 4.1). AspectJML works in this manner to avoid the obliviousness problem (see Section 4.4 for more details). Due to this restriction, it does not matter if we write a general pointcut expression like:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
@Pointcut("execution(* *.*(..))"+
 "&& args(width, height)")
void sizes(double width, double height) {}
```

4.3.6 Reusing Pointcuts

As with AspectJ, programmers can reuse pointcut declarations in AspectJML. The main advantage of reusing a pointcut is when we want to select the same join points already captured by another pointcut, but combined with more join points. For instance, consider the following two pointcut declarations:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
@Pointcut("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
void sizeMethsPre(double width, double height) {}

//@ ensures this.dimension.doubleValue() == width * height;
@Pointcut("sizeMethsPre(width, height) &&" +
  "(execution(* Package.*Size2(double, double))" +
  "&& args(width, height))")
void sizeMethsPost(double width, double height) {}
```

The first one defines the pointcut sizeMethsPre used to specify two crosscutting preconditions. The second pointcut, named sizeMethsPost, reuses the pointcut sizeMethsPre to apply one crosscutting normal postcondition. The main difference is that the second pointcut besides reusing the pointcut sizeMethsPre, it also selects any method ending in "Size2" with two arguments of type **double**. As a result, the reuse of pointcuts helps programmers to write less verbose pointcuts in AspectJML.

4.3.7 Specification of Unrelated Types

Another issue to consider is whether or not AspectJML can modularize inter-type[2] crosscutting specifications. All the crosscutting contract specifications we discuss are

[2] Inter-types here are not the AspectJ feature [KHH$^+$01] that allows adding methods or fields with a static crosscutting mechanism. Instead, they are unrelated modules in a system; that is, types that are not related to each other but can present a common crosscutting contract structure.

related to one type (intra-type) or its subtypes. However, AspectJ can advise methods of different (unrelated) types in a system. This quantification property of AspectJ is quite useful [VCFS10] but can also be problematic from the point of view of modular reasoning, since one needs to consider all the aspect declarations to understand the overall system behavior [Ste06, SGR+10, SPAK10, ITB11, BRLM11, RLL+13b]. Instead of ruling this completely out, the design of AspectJML allows the specifier to use specifications that constrain unrelated inter-types, but in an explicit and limited manner (see Section 4.4 for more details about non-obliviousness in AspectJML).

As an example, recall the JML specifications in Figure 2.4. We know that all the methods declared in Package are forbidden to throw exceptions (see the **signals_only** specification). Suppose now that the methods declared in type Courier and Package's subtype (GiftPackage and DiscountedPackage) (see Figure 2.1) also has this constraint. Note that the type Courier is not a subtype of Package. They are related in the sense that the method deliver depends on the Package type due to the declaration of a formal parameter. Consider further that Courier contains many methods that are not dependent on Package in any way. Consider the following type declaration:

```
interface ExceptionSignallingConstraint {
 @InterfaceXCS
 class ExceptionSignallingConstraintXCS {
  //@ signals_only \nothing;
  @Pointcut("execution(* ExceptionSignallingConstraint+.*(..))")
  public void allMeth() {}
 }
}
```

This type declaration illustrates how we specify crosscutting contracts for interfaces. In @AspectJ, pointcuts are not allowed to be declared within interfaces. We overcome this problem by adding an inner class that represents the crosscutting contracts of the outer interface declaration. As a part of our strategy, the pointcut declared in the inner class refers only to the outer interface (see the reference in the pointcut predicate expression). Now any type that wants to forbid its method declarations to throw exceptions need only to implement the interface ExceptionSignallingConstraint. Such an interface acts like a marker interface [HU03]. This is important to avoid obliviousness and maintain modular reasoning. (according to our definition 2.1.1).

Note that the inner class is marked with the **@InterfaceXCS** annotation. This is to distinguish from any other inner class that could also be declared within our crosscutting interface. Without this mechanism, the AspectJML compiler will not be able to find the crosscutting contracts for the interface ExceptionSignallingConstraint.

Design decision behind crosscutting contract interfaces

One question that can arise is why inner classes for interface contracts? To justify our decision let us discuss the reason why we cannot declare a pointcut within an interface. The reason is that a pointcut in AspectJ can be declared with any privacy modifier. Java interfaces, on the other hand, can only be declared as public. This is the first reason we adopt an inner class for interface contracts.

Another reason to design interface contracts with inner classes was discussed in Section 2.4. In JML one can specify contracts with different visibility access. As such, suppose that we want to add a protected specification to methods in several unrelated types. If we consider a plain method declaration in a Java interface, we cannot write such a specification. We cannot write a protected specification case in a public method in an interface type. Recall that in lightweight specifications, the visibility is defined by the method own visibility. Therefore, due to these two limitations (both related to visibility access), we decided to design crosscutting contracts in interfaces through inner classes.

This solution is very similar to the one used by the DbC language Code Contracts [FBL10]. Code Contracts add their clauses within a method body. Since the C# language (whose Code Contracts language is used for) will not let one put method bodies within an interface, writing contracts for interface requires creating a separate class to hold them. The interface and its contract class are related/linked via a pair of annotations in C# style. See the example below:

```
[ContractClass(typeof(IFooContract))]
interface IFoo {
 int Count f get; g
 void Put(int value );
}
[ContractClassFor(typeof(IFoo))]
abstract class IFooContract : IFoo {
 int IFoo.Count {
  get {
   Contract.Ensures( 0 <= Contract.Result<int>() );
   return default(int ); // dummy return
  }
 }
 void IFoo.Put(int value){
  Contract.Requires( 0 <= value );
 }
}
```

What if we have partial quantification?

The crosscutting contract interface discussed above is useful to modularize some crosscutting contract that occurs in several types. As observed, the ExceptionSignallingConstraint interface once implemented constrains all the methods in implementing types. What if we want to specify contracts that affect only some methods of some unrelated types? Basically, we have two ways to handle this situation in AspectJML.

Solution 1. One way is to negate some recurrent pattern. For example, suppose we want to constrain all methods except setters and getters. Thus, we have the following refined crosscutting contract interface:

```
 interface ExceptionSignallingConstraint {
  @InterfaceXCS
  class ExceptionSignallingConstraintXCS {
   //@ signals_only \nothing;
   @Pointcut("(execution(* ExceptionSignallingConstraint+.*(..)) && "+
             "!execution(void ExceptionSignallingConstraint+.set*(..))"+
              "!execution(* ExceptionSignallingConstraint+.get*()))")
   public void allMeth() {}
  }
 }
```

We exclude the setters and getters by composing the negation of two **execution** pointcuts with a wildcard; one for setters and one for getters, respectively.

Solution 2. If the excluded join points can vary to the extent that we cannot use a property-based pointcut (which is part of a method name combined with a wildcard), we can use the AspectJ's **@annotation** pointcut (see Table 2.2) to exclude specific join points (methods). Consider the following version of our crosscutting contract interface:

```
 interface ExceptionSignallingConstraint {
  @InterfaceXCS
  class ExceptionSignallingConstraintXCS {
   //@ signals_only \nothing;
   @Pointcut("execution(* ExceptionSignallingConstraint+.*(..)) && "+
              "!@annotation(ExcludeMarker)")
   public void allMeth() {}
  }
  @Retention(RetentionPolicy.RUNTIME)
  @Documented
  public @interface ExcludeMarker {}
 }
```

With this solution, we need to mark/annotate in the implementing types all the methods that are not part of the crosscutting contract concern. So in the following Package type, the crosscutting contract will be excluded from both setSize and reSize methods.

```
public class Package implements ExceptionSignallingConstraint{
 ...

 @ExcludeMarker
 public void setSize(double width, double height){...}

 @ExcludeMarker
 public void reSize(double width, double height){...}

 public boolean containsSize(double width, double height){...}

 public double getSize(){...}

 public void setWeight(double weight) {...}
}
```

The second solution is good when we have few join points to exclude. If we have a type with 200 methods and only 50 methods are part of the crosscutting contract, maybe it would be better to modularize such a crosscutting structure in the type itself by listing the join points in the pointcut expression and trying as much as possible to use property-based pointcuts to capture this situation. Some examples of property-based matching were discussed in Section 2.5.

It is important to note that even if we discuss the above two solutions for crosscutting contract interfaces, we can apply those solutions for classes as well. In other words, we can compose pointcuts within classes using the patterns described by the above solutions. Another point to stress is that if we want to avoid implementing the crosscutting contract interfaces for the solely purpose to inherit specifications, we can use annotation pattern style. In other words, instead of activating crosscutting contracts by implementing an interface, we just annotate a type by using an annotation type.

Crosscutting contract interfaces and separate files

There is a way to sidestep the disadvantage of having inner classes to declare interface contracts. We discuss this workaround in the next section.

4.3.8 Separate Files for Crosscutting Contracts

As discussed, in JML, we can write specifications in separate files (see Section 2.4.10). In AspectJML, we can also untangle specifications by putting them in a separate file. Since AspectJML is an extension of JML, the extension of the separate file could be either .jml or .ajml. Note that a method cannot contain an implementation (body) in such .jml or .ajml files. Only the syntactic interface signatures must be declared.

Using separate files is beneficial when we want to specify a contract for an interface. As we discussed in the previous section, we need to add an inner class to write contracts for an interface. If we use a separate file, we can avoid such inner classes when we deal with public specifications. Please, consider the following crosscutting contract interface:

```
// declared in the file ExceptionSignallingConstraint.jml
interface ExceptionSignallingConstraint {
  //@ signals_only \nothing;
  @Pointcut("execution(* ExceptionSignallingConstraint+.*(..))")
  public void allMeth();
}
```

This interface constraint looks like more natural than the solution with inner classes. The AspectJML compiler handles such an interface in the same manner as it does with ordinary Java source files. The disadvantage of this approach is that since we are writing pointcut expressions in other files than .java, we do not get immediate feedback about a malformed pointcut. We need to compile to get an error (if any). With Eclipse/A-JDT [KM05], we get early feedback before calling the AspectJML compiler. Note that this solution only works for public specifications. If we need protected specifications only, we need the inner class solution again.

4.3.9 XCS and Multiple Specification Cases

As we discussed in Section 2.4.6, JML has the notion of method specification cases. This feature is useful to make distinct execution scenarios, which are separated as blocks with the JML keyword **also**. Recall the following specification cases for the method setSize in Package:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
//@ also
//@ requires width > 0 && height > 0;
//@ requires width * height > 400; // exceeding allowed dimension
//@ ensures this.width == \old(this.width);
//@ ensures this.height == \old(this.height);
//@ signals(SizeDimensionException) width * height > 400;
//@ signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException{...}
```

As noted, the two specification cases begin with a recurrent precondition. One way to modularize this precondition is by means of JML nested specifications (as shown in Section 2.4.10). Another way to modularize such a precondition is illustrated below:

```
 //@ requires width > 0 && height > 0;
 @Pointcut("execution(* Package.*Size(double, double))"+
  "&& args(width, height)")
 void sizes(double width, double height) {}
```

Among other join points, this pointcut intercepts the method setSize. As before, we have the precondition checking for method setSize. The main difference is that the crosscutting precondition crosscuts the two specification cases to enforce the modularized recurrent precondition. With the modularization, we have the following refined specification cases for setSize method:

```
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
//@ also
//@ requires width * height > 400; // exceeding allowed dimension
//@ ensures this.width == \old(this.width);
//@ ensures this.height == \old(this.height);
//@ signals(SizeDimensionException) width * height > 400;
//@ signals_only SizeDimensionException;
void setSize(double width, double height) throws SizeDimensionException{...}
```

However, if the preconditions of the specification cases are different (without an intersection between them), the previous pointcut cannot be used to check preconditions since it will inject the precondition for all specification cases. Hence, if we have two methods with two specification cases, where we have a recurrent precondition between the methods but only occurring in one specification case for each method, we cannot use AspectJML/XCS to modularize them.

Still considering method specification cases, one question that can arise is what if we have recurrent specification cases? Does AspectJML handle this situation? The answer is yes. AspectJML does handle crosscutting specification cases in JML that are scattered among methods. As an example, let us assume that the following methods setSize and reSize have two specification cases each and the first specification is duplicated among these methods:

```
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
//@ also
//@ ...
void setSize(double width, double height) throws SizeDimensionException{...}

//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
//@ also
//@ ...
void reSize(double width, double height) throws SizeDimensionException{...}
```

In this context, the following AspectJML pointcut can be used to modularize the recurrent specification case:

```
//@ public behavior
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
//@ signals_only \nothing;
@Pointcut("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
void sizes(double width, double height) {}
```

The key difference between this specified pointcut and others presented until now is that the specification case denotes a heavyweight specification in JML. In AspectJML, we decided that whenever a programmer add a heavyweight specification in a pointcut, it will be applied as a join of specification cases instead of checking the contracts at every specification case of an intercepted join point.

In summary, in AspectJML, lightweight specifications are checked at all the specification cases that an intercepted method has. On the other hand, heavyweight specifications are conjoined as a regular JML specification case, which separates one to each other through an **also** construct. This design decision in AspectJML gives more expressiveness to handle more situations a specifier can face by using JML specifications.

4.3.10 More XCS Examples

We already discussed the main AspectJML features in relation to crosscutting contract specifications. We discuss now two more scenarios where AspectJML is more expressive than a conventional DbC language like JML.

Scenario 1

Consider again the implementation (without JML specifications) of the Package classes illustrated in Figure 2.1. Consider further that each package type has a specific constraint on its dimension. For instance, suppose that the dimension of a package of static type Package should be between 0 and 400 cm^2 and all methods (in Package) should establish this constraint. To enforce this design constraint, we can write the following invariant:

```
//@ invariant this.width * this.height > 0
//@      && this.width * this.height <= 400;
```

Suppose now that the dimension of a GiftPackage should be between 0 and 600 cm^2 and all methods (in GiftPackage) should establish this constraint. Moreover, the dimension of a DiscountedPackage should be between 0 and 800 cm^2 and all methods (in DiscountedPackage) should establish this constraint. Hence, the following invariants enforce these constrains, respectively:

```
//@ invariant this.width * this.height >=0
//@      && this.width * this.height <= 600;
//@ invariant this.width * this.height >=0
//@      && this.width * this.height <= 800;
```

Intuitively, the above invariants are the ones a programmer may write when trying to enforce such design constraints. But the first invariant related to the type Package has a problem. Since in JML an instance invariant is inherited by subtypes, the types GiftPackage and DiscountedPackage should enforce such restrictions. The main problem is that by inheriting the first invariant, a programmer cannot write the other two since they are trying to weaken the inheriting constraint over a package's dimension.

To solve this problem in plain JML, a programmer may write the following invariant on top of the hierarchy, which is the type Package in this case:

```
//@ invariant this.width * this.height > 0;
```

This invariant will be inherited and enforced by all package types in the hierarchy. Moreover, to fulfill the Package's constraints, we need to write the following pair or pre- and postcondition, for every method, to ensure that the dimension does not exceed 400 cm:

```
//@ requires this.width * this.height <= 400;
//@ ensures this.width * this.height <= 400;
```

For type GiftPackage, we need to write the following pair or pre- and postcondition for every method:

```
//@ requires this.width * this.height <= 600;
//@ ensures this.width * this.height <= 600;
```

This ensures that the package's dimension does not exceed 600 cm^2. Remember that we do not need to write an invariant since it was inherited from type Package. Analogously, we need to write the following pair of pre- and postcondition for every method in type DiscountedPackage:

```
//@ requires this.width * this.height <= 800;
//@ ensures this.width * this.height <= 800;
```

This ensures that the package's dimension does not exceed 800 cm^2. As with GiftPackage, we do not need to write an invariant since it was inherited from Package.

Considering that we have five methods per type, we need to write one invariant and fifteen pairs or pre- and postcondition to fulfill the design constraints of the package types. Unfortunately, there is no way to get rid of the scattering of pre- and postconditions if we use JML as our DbC language.

The good news is that a programmer can enhance this scenario if he/she considers to use AspectJML/XCS feature. With AspectJML he/she needs to write the same invariant constraint, but instead of fifteen pairs of pre- and postcondition, he/she only needs three accompanying corresponding pointcuts. These pointcuts and specifications are the following:

```
//@ requires this.width * this.height <= 400;
//@ ensures this.width * this.height <= 400;
@Pointcut("execution(* Package.*(..)) && within(Package)")
void dimensionLimitPackage() {}

//@ requires this.width * this.height <= 400;
//@ ensures this.width * this.height <= 400;
@Pointcut("execution(* Package.*(..)) && within(GiftPackage)")
void dimensionLimitGiftPackage() {}

//@ requires this.width * this.height <= 800;
//@ ensures this.width * this.height <= 800;
@Pointcut("execution(* DiscountedPackage.*(..))" +
      "&& within(DiscountedPackage)")
void dimensionLimitDiscountedPackage() {}
```

Each pointcut should be placed in its corresponding type. So the first is for type Package, the second is for type GiftPackage, and the last one is for type DiscountedPackage. In order to avoid intercepting any method of a subtype, each pointcut includes the pointcut designator **within**. For example, the pointcut method dimensionLimitPackage intercepts all methods in type Package. It avoids intercepting any method on subtypes due to **within**(Package). This ensures that the selected join points should be lexically contained in type Package.

Scenario 2

For our scenario 2, let us consider that all methods (in type Package of Figure 2.1) that have arguments of type **double** should be greater than zero. In JML, however, there is no way to write this constraint only once and apply for all methods in Package. For instance, the following preconditions are needed to enforce this constraint for the methods setSize and reSize in Package:

```
//@ requires width > 0;
//@ requires height > 0;
void setSize(double width, double height){...}

//@ requires width > 0;
//@ requires height > 0;
void reSize(double width, double height){...}
```

In AspectJML, on the other hand, we can write only one precondition to enforce our design constraint over the arguments of type **double** contained in methods of type Package. Here is the pointcut in AspectJML:

```
/*@ requires (\forall int i; 0 <= i && i < thisJoinPoint.getArgs().length;
  @    (thisJoinPoint.getArgs()[i] instanceof Double)
  @      ==> ((Double)thisJoinPoint.getArgs()[i]).doubleValue() > 0);
  @*/
@Pointcut("execution(* Package.*(..,double,..))")
void doubleParamXCS(JoinPoint thisJoinPoint){}

void setSize(double width, double height){...}

void reSize(double width, double height){...}
```

The pointcut doubleParamXCS is responsible for checking our design constraint. Note that it uses a JML **requires** clause with a **forall** quantifier that inspects each parameter's value of the intercepted methods (which includes all the methods in Package). We access each parameter's value through the AspectJ's **thisJoinPoint** variable that is exposed via an argument type in the pointcut doubleParamXCS. This is the standard way to access the reflective variable **thisJoinPoint** in AspectJ, which is the same in AspectJML.

If the discussed design constraint should be applied to other types, we can move the pointcut doubleParamXCS to a crosscutting contract interface. This way, the constraint in automatically applied to the implementing types.

4.3.11 Exported AspectJML/XCS Examples

The crosscutting contract specifications used to modularize the crosscutting contract scenarios, discussed in Section 3.3, are illustrated in Figure 4.1 (the shadowed part illustrates the XCS in AspectJML's pointcuts and specifications).

4.4 AspectJML's Benefits

In this section we discuss the main AspectJML benefits when used for crosscutting contract specification.

Enabling modular reasoning

Recall that our notion of modular reasoning (Definition 2.1.1) means that one can soundly verify a piece of code in a given module, such as a class, using only the module's own specifications, its own implementation, and the interface specifications of modules that it references [DL96, Mey00, Lea06, LN13, RLL+13b].

With respect to whether or not AspectJML supports modular reasoning like a DbC language such as JML, consider the client code, which we will imagine is written by Cathy, shown in Figure 4.2.

To verify the call to setSize, Cathy must determine which specifications to use. If she uses the definition of modular reasoning, she must use the specifications for setSize in Package. Let us assume that she uses the JML specifications of Figure 2.4. Hence, she uses:

(1) the pre- and postconditions located at the method setSize (lines 11-14);

(2) the first invariant definition on line 8, which constrains the Package's dimension; and

```
1  public class Package {
2   //@ public model JMLDouble dimension;
3   //@ public model JMLDouble weight;
4   protected double width, height, _weight;
5   //@ protected represents dimension = new JMLDouble(this.width * this.height);
6   //@ protected represents weight = new JMLDouble(this._weight);
7
8   //@ public invariant this.dimension.doubleValue() >= 0;
9   //@ public invariant this.weight.doubleValue() >= 0;
10
11  //@ requires width > 0 && height > 0;
12  //@ requires width * height <= 400; // max dimension
13  @Pointcut("execution(* Package.*Size(double,double))"+
14   "&& args(width, height)")
15  public void sizes(double width, double height) {}
16
17  //@ ensures this.dimension.doubleValue() == width * height;
18  @Pointcut("(execution(* Package.setSize(double,double))"
19   + "|| execution(* Package.reSize(double, double)))"+
20   "&& args(width, height)")
21  public void sizeChange(double width, double height) {}
22
23  //@ signals_only \nothing;
24  @Pointcut("execution(* Package+.*(..))")
25  public void packageMeths() {}
27
28  public void setSize(double width, double height){...}
29
30  //@ requires this.dimension.doubleValue() != width * height;
31  public void reSize(double width, double height){...}
32
33  public boolean containsSize(double width, double height){...}
34
35  public double getSize(){...}
36
37  //@ ...
38  public void setWeight(double weight) {...}
39 ... // other methods
40 }
```

Figure 4.1: The crosscutting contract specifications with AspectJML used so far for the type Package related to the crosscutting scenarios illustrated in Figure 3.2.

(3) the second invariant (line 9) related to the Package's weight.

Cathy only needs these three steps, including six JML pre- and postcondition, and invariant specifications, when using plain JML reasoning. (Package has no supertype; otherwise, she would also need to consider specifications inherited from such supertypes.) After obtaining these specifications, she can see that there is a precondition violation regarding the width value of 0 passed to setSize (in Figure 4.2).

Suppose now that Cathy wants to perform again the same modular reasoning task, but using the AspectJML specifications in Figure 4.1 instead of the JML specifications defined in Figure 2.4. In this case she needs to find the following pieces of specified code:

(1) the first invariant definition on line 8, that constrains the Package's dimension;

(2) the second invariant (line 9) related to the Package's weight;

```
// written by Cathy
public class ClientClass {
 public void clientMeth(Package p)
  { p.setSize(0, 1); }
}
```

Figure 4.2: setSize's client code.

(3) the preconditions of the pointcut (lines 11-12) sizes, since it intercepts the execution of method setSize;

(4) the normal postcondition (line 17) located at the pointcut sizeChange; and

(5) the exceptional postcondition (line 23) of pointcut packageMeths.

As before, this involves only modular reasoning and she can still detect the potential precondition violation related to the Package's width. In this case, Cathy needed the same six specifications, but with two more steps (five in total) to reason about the correctness of the call to setSize (see our expanded modular reasoning Definition 2.1.2). So, although AspectJML supports modular reasoning, Cathy must follow a slightly more indirect process to reason about the correctness of a call. This confirms that the obliviousness issue present in AspectJ-like languages [FF00] does not occur in this example. Cathy is completely aware of the contracts of Package class, though it does take her longer to determine them.

In summary, as with JML invariants, annotated pointcuts are quantified statements that the way we reason about is the same as invariants in plain JML. The extra steps we need during reasoning is related to our Definition 2.1.2.

Enabling documentation

This example shows that, despite the added indirection, reasoning with AspectJML specifications does not necessarily have a modularity difference compared to reasoning with JML specifications. Only the location where these specifications can appear can be different, due to the use of pointcut declarations in AspectJML.

Our conclusion is that an inherent cost of crosscutting contract modularization and reuse is the cost of some indirection in finding contract specifications (expanded modular reasoning), which is necessary to avoid scattering (repeated specifications). However, using AspectJML, users also have the choice to what extent we should modularize or even not modularize at all crosscutting contracts.

Taming obliviousness

Since AspectJML allows pointcut declarations in AspectJ-style, one can argue that a programmer can specify several unrelated modules in one single place. This phenomenon brings into focus again whether AspectJML allows the controversial obliviousness property of AOP [Ste06, SGR+10, SPAK10, ITB11, BRLM11, RLL+13b].

The answer is no. AspectJML rules out this possibility. If one tries to write such pointcuts, they will have no effect with respect to crosscutting specification and runtime checking. This happens because AspectJML associates the specified pointcut with the type in which it was declared (see the discussion in the next section and the generated code in Figure 4.3). Hence, only join points within the given type or its subtypes are allowed. The cross-references generated by AspectJML (see Section 4.7) can help visualize the intercepted types.

Even though there is no way in AspectJML to specify unrelated modules anonymously, the declared pointcuts can still be used within aspect types that can crosscut unrelated types. Those pointcuts can be used to modularize other kinds of crosscutting concerns using the standard AspectJ pointcuts-advice mechanisms [KHH+01].

4.5 Enforcing Information Hiding in AspectJML with Client-Aware Checking

One goal of AspectJML is to ease proper checking of visibility rules of JML specifications. We call our technique for doing this client-aware checking (CAC). CAC aims to runtime check contracts from the client's point of view. We describe CAC in more detail below.

The most important feature of CAC is that it checks method specifications on the client side of each call (i.e., at the call site). Doing this allows CAC to be consistent with information hiding rules, by checking only the visible pre- and postconditions for each call. This also avoids the not meaningful error reporting that may arise from overly-dynamic RACs.

To see how these checks are made, consider again the public and protected specification cases for method setSize discussed and shown in Section 3.4:

```
 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 400; // max dimension
 //@ ensures this.dimension.doubleValue() == width * height;
 //@ signals_only \nothing;
 //@ also
 //@ protected behavior
 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 600; // max dimension
 //@ ensures this.width == width;
 //@ ensures this.height == height;
 //@ signals_only \nothing;
 public void setSize(double width, double height){...}
```

Also, consider the same public client code also discussed in Section 3.4:

```
public class PackageClient {
  public void setSizePackageClient(Package p) {
    p.setSize(200, 2);
  }
}
```

In order to reason about the correctness (using the above JML specifications) of method call to setSize in class PackageClient, we use the proof rule for method calls that allows one to derive

$$\{pre_m^T[\vec{a}/\vec{f}]\}\ p.m(\vec{a})\ \{post_m^T[\vec{a}/\vec{f}]\}$$

from a specification $(T \triangleright pre_m^T, post_m^T)$ associated with the receiver p's the static type T. (The notation $[\vec{a}/\vec{f}]$ means the substitution of the actual parameters $\vec{a}$ for m's formals $\vec{f}$.) An automated static verifier that uses weakest precondition semantics can modularly replaces a call $p.m(\vec{a})$ by the sequence of "**assert** $pre_m^T[\vec{a}/\vec{f}]$; **assume** $post_m^T[\vec{a}/\vec{f}]$" [BL05].

Hence, by using **assert** and **assume** statements in JML-style, we have the following way to reason about the method setSize in class PackageClient:

```
public class PackageClient {
  public void setSizePackageClient(Package p) {
    // public preconditions reasoned as assert statements
    //@ assert 200 > 0 && 2 > 0;
    //@ assert 200 * 2 <= 400;
    p.setSize(200, 2);
    //@ assume p.dimension.doubleValue() == 200 * 2;
    // public normal postconditions reasoned as assume statements
  }
}
```

In AspectJML, the runtime checks are instrumented just like the above **assert** and **assume** statements used for reasoning about the correctness of method setSize. That is, in the CAC technique enabled, the AspectJML compiler injects runtime checks around each method call to check the pre- and postconditions of the statically-visible specifications for the call.

As before, let us assume that the implementation of setSize mistakenly increments the width by 1. Therefore, we got the following normal postcondition error, when employing AspectJML with CAC enabled, for the method setSize:

```
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLExitPublicNormalPostconditionError:
by method Package.setSize regarding specifications at
File "Package.java", line 23 (Package.java:23), when
  'this.dimension' is 402.0
  'width' is 200.0
  'height' is 2
  ...
```

As observed, this time the error output does not mention the protected (hidden) fields width or height anymore (as shown in Section 3.4). The public client now is aware of the specifications that are being checked during runtime. In other words, information hiding is working as expected and the client has all the benefits expected from runtime assertion checking. Also, CAC provides a specific postcondition violation for public postconditions (as shadowed above).

The protected postcondition should be omitted from the error reporting since specifications of wider visibility should be refined by their counterparts of narrower visibility (hence public method preconditions should imply the protected preconditions, and protected postconditions should be implied by public postconditions, when the public preconditions hold [DL96]). The reason for this is that a client would be surprised if they encountered assertion violation errors with invisible assertions [Kic96].

Another problem that AspectJML with CAC avoids, in contrast to overly-dynamic checking, is the missing expected violations. As discussed in Section 3.4, the following public client code

```
p.setSize(300, 2);
```

misses the expected precondition violation when considering the specifications shown above for method setSize. Since the overly-dynamic checking also considers the weaken protected precondition, the above client does not break any precondition.

Fortunately, with AspectJML/CAC, a public client calling setSize(300, 2), will result in a precondition error, as expected:

```
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLEntryPublicPreconditionError:
by method Package.setSize regarding specifications at
File "Package.java", line 15 (Package.java:15),
line 16 (Package.java:16), when
  'width' is 300.0
  'height' is 2
  ...
```

4.6 Implementation

We have implemented the AspectJML crosscutting contract specification and client aware checking techniques in the AspectJML compiler called ajmlc [RLB+14]. It also has various code optimizations [RLL+13b] in relation to early versions of ajmlc and in relation to the classical jmlc compiler [Che03]. AspectJML is an open source project and is broadly available for download and for modication under GNU General Public License. This is the first runtime assertion checking compiler to support crosscutting contract specifications and information hiding during runtime.

The ajmlc compiler itself was described in previous works [RSL+08, Reb08]. The main difference between AspectJML/ajmlc and JML/ajmlc is that the latter was designed to support only JML features we described in Section 2.4 and to also generate instrumented code compatible to Java ME applications [RSL+08, Reb08], which was not possible otherwise. Both compiler versions generate AspectJ code to check contracts during runtime. So the new ajmlc supports everything as before with the addition to crosscutting contract specification using some @AspectJ constructs and the ability to check contracts at client side.

4.6.1 Compilation strategy

In this section, we illustrate examples of instrumented code generated to check crosscutting contracts written in XCS and instrumented code generated to check contracts at client side (CAC). In addition, we discuss how the precedence order of generated advice is enforced in AspectJML.

Implementation of XCS

Figure 4.3 shows the **before** advice generated by the ajmlc compiler to check the crosscutting preconditions of class Package defined in Figure 4.1.[3] The variable rac\$b denotes the precondition to be checked. This variable is passed as an argument to JMLChecker.checkPrecondition, which checks such preconditions; if it is not true, then a precondition error is thrown. As discussed in Section 4.4, note that the exposed object

[3] The ajmlc compiler provides a compilation option that prints all the checking code as aspects instead of weaving them.

```
/** Generated by AspectJML to check the precondition of
  * method(s) intercepted by sizeMeths pointcut. */
before (Package object$rac, final double width,
  final double height) :
  (execution(* p.Package.*Size(double,double))
  && this(object$rac) && args(width, height)) {
  boolean rac$b = (((width > +0.0D) && (height > +0.0D))
   && ((width * height) <= 400.0D));
  JMLChecker.checkPrecondition(rac$b, "errorMsg");
 }
```

Figure 4.3: Generated before advice to check the crosscutting preconditions of Package in Figure 4.1.

```
/** Generated by JML to check the protected
  * precondition of method setSize. */
before (p.Package object$rac, double width, double height):
 call(void p.Package.setSize(double, double))
 && within(p.*) && within(p.Package+)
 && target(object$rac) && args(width, height) {
 if ((   &&   ){
  boolean rac$b = (((width > +0.0D) && (height > +0.0D)) &&
                 ((width * height) <= 600.0D));
  JMLChecker.checkPrecondition(rac$b, "...");
}
```

Figure 4.4: Generated before advice to check the protected preconditions of the method setWeight in type Package.

type is Package. Hence, this precondition can only be checked to join points of Package or its subtypes like GiftPackage.

Implementation of CAC

We also implemented the CAC technique in our AspectJML RAC compiler (ajmlc) [RLB+14]. Figure 4.4 shows the **after returning** advice generated by the ajmlc compiler to check the protected preconditions of method setWeight discussed in Section 4.5.

Let us assume that the type Package is declared in a Java package, say p. The package information is used for protected precondition checking. To properly check protected preconditions, we rely on the AspectJ **within** pointcut in the generated advice (see the shadowed part in Figure 4.4). It guarantees the effective protected precondition to check all calls to method setSize that lexically occurs from within package p (denoted by **within**(p.∗)) or, if those calls are made from outside p, then they must occur in a subclass of p.Package (e.g., GiftPackage) denoted by **within**(p.Package+).

For public preconditions, ajmlc uses the **within**(∗) pointcut because any call to the method setSize, regardless its client visibility, must respect the specified public precondition. For the other privacy modifiers, ajmlc uses an appropriate variation of the **within** pointcut.

For handling multiple specification cases, which is possible in JML, ajmlc generates a dedicated **before** advice for each JML specification case with different privacy modifiers. For instance, the following generated **before** advice is responsible for checking the public preconditions of the method setSize:

```
/** Generated by JML to check the public
  * precondition of method setSize. */
before (p.Package object$rac, double width, double height) returning():
 call(void p.Package.setSize(double, double))
 && within(*) &&
 target(object$rac) && args(width, height) {
 if ((  &&  ){
  boolean rac$b = ((((width > +0.0D) && (height > +0.0D)) &&
          ((width * height) <= 400.0D));
  JMLChecker.checkPrecondition(rac$b, "...");
}
```

Ordering of checks

As acknowledged, AspectJML can also compile AspectJ/@AspectJ programs. As AspectJML/ajmlc generates AspectJ aspects to check contracts during runtime, we need precedence enforcement to avoid conflicts with other aspects in the system responsible for other crosscutting concerns.

For example, if we compile a class named Package with ajmlc, it generates an AspectJ aspect called AspectJMLRAC_Package. So to ensure the precedence rules between AspectJMLRAC_Package and other aspects, the former declares the following AspectJ construct:

```
declare precedence : AspectJMLRAC_*, *;
```

This precedence ensures that any aspect in the system that starts with "AspectJMLRAC_" has higher precedence that any other aspect in the system (denoted by the second wildcard). The **declare precedence** construct is discussed itself in Section 2.5. In terms of precedence rules, the above construct ensures that all AspectJML preconditions , which are implemented with **before** advice, to be checked first. Hence, once preconditions are satisfied, the other aspects are allowed to run their respective **before** advice. Analogously, the AspectJML postconditions, which is implemented by **after** advice, are only checked after all the **after** advice of other aspects are executed first.

The use of **declare precedence** guarantees that no other code (i.e., within **before** advice) runs if we have a precondition violation. In the same manner, the precedence rule prevents undetected postcondition violations, which could happen if postconditions were checked before the execution of any other **after** advice.

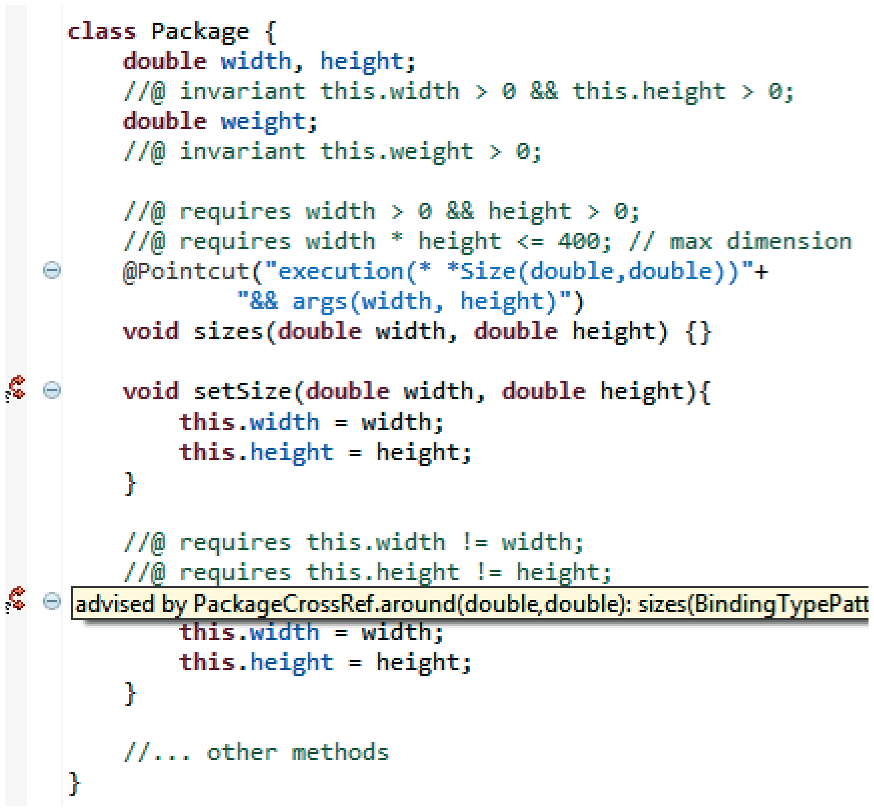
In summary, AspectJML generated aspects that have the higher precedence in relation to other aspects in the system. The precedence rules that govern the runtime checking are the same ones described in Section 2.5 and summarized by Figure 2.5.

4.6.2 Contract violation example in AspectJML

As an example of runtime checking using AspectJML/ajmlc, recall the client code illustrated in Figure 4.2. In this scenario, we got the following precondition error in the AspectJML RAC:

```
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLEntryPreconditionError:
by method Package.setSize regarding code at
File "Package.java", line 13 (Package.java:13), when
  'width' is 0.0
  'height' is 1.0
  ...
```

As can be seen, in this error output, the shadowed input parameter width is displaying 0.0. But the precondition requires a package's width to be greater than zero. As a result, this precondition violation occurs during runtime checking when calling such client code.



```
class Package {
    double width, height;
    //@ invariant this.width > 0 && this.height > 0;
    double weight;
    //@ invariant this.weight > 0;

    //@ requires width > 0 && height > 0;
    //@ requires width * height <= 400; // max dimension
    @Pointcut("execution(* *Size(double,double))"+
            "&& args(width, height)")
    void sizes(double width, double height) {}

    void setSize(double width, double height){
        this.width = width;
        this.height = height;
    }

    //@ requires this.width != width;
    //@ requires this.height != height;
advised by PackageCrossRef.around(double,double): sizes(BindingTypePatt
        this.width = width;
        this.height = height;
    }

    //... other methods
}
```

Figure 4.5: The crosscutting contract structure in the Package class using AspectJML/AJDT [KM05].

4.7 Tool Support

In aspect-oriented programming, development tools like AJDT [KM05], allow programmers to easily browse the crosscutting structure of their programs. For AspectJML, we are developing analogous support for browsing crosscutting contract structure. We use the existing functionality of AJDT to this end.

For example, consider the crosscutting contract structure of the Package class using AspectJML/AJDT (see Figure 4.5). Note the arrows indicating where the crosscutting contracts apply. In plain AspectJ/AJDT this example shows no crosscutting structure information, because it has only pointcut declarations without advice. In AspectJ, we need to associate the declared pointcuts to advice in order to be able to browse the crosscutting structure of a system. We have implemented an option (that is enabled by default) in AspectJML that generates the cross-references information for crosscutting contracts, thus allowing one to visualize the crosscutting structure.

To enable the crosscutting contract structure view, AspectJML generates an **around** advice in AspectJ, without effect in the base code, to associate with the corresponding pointcut in AspectJML pointcuts. For instance, considering the Package type (Figure 4.5), AspectJML generates an AspectJ aspect called PackageCrossRef. Figure 4.5 illustrates this in practice. Once compiled, we can see that the method reSize in Package is intercepted by the pointcut sizes from PackageCrossRef. Through the cross-references, we go to the **around** advice (in the aspect PackageCrossRef) that actually activates the arrow through AJDT. But the cross-reference code we generate for AspectJML allows the programmers from a javadoc-link to point to the right pointcut sizes in type Package. The generated code looks as follows:

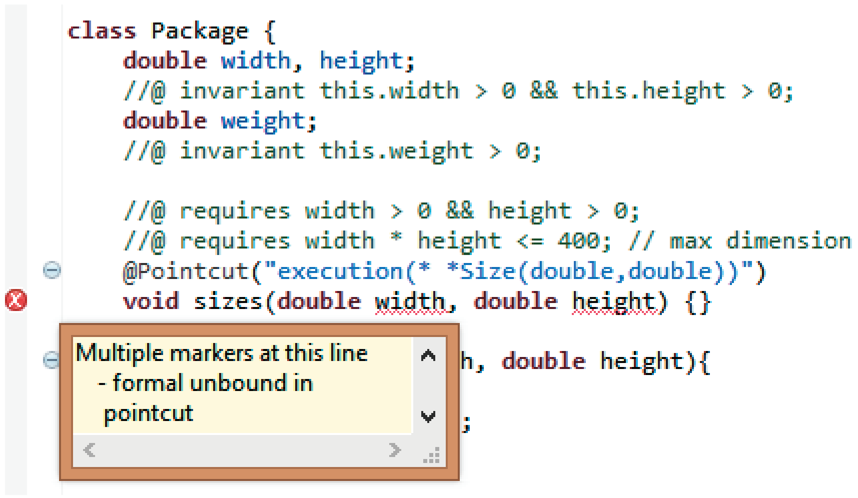
```
public privileged aspect PackageCrossRef {
  static boolean advise = false;
 /** Generated by AspectJML to enable the crossref for
 * the XCS pointcut {@link Package#sizes(double, double)}*/
  pointcut sizes(double width, double height): ... ;
  Object around(...): sizes(width, height) && if(advise){return null;}

  /** Generated by AspectJML to enable the crossref for
   * the XCS pointcut {@link Package#sizeChange(double, double)} */
  pointcut sizeChange(double width, double height): ...;
  Object around(...): sizeChange(width, height)
                        && if(advise){return null;}

  /** Generated by AspectJML to enable the crossref for
   * the XCS pointcut {@link Package#packageMeths()} */
  pointcut packageMeths(): (execution(* Package+.*(..))) && ...;
  Object around(): packageMeths() && if(advise){return null;}
}
```

Figure 4.6 shows another example where the use of the AspectJ/AJDT helps an AspectJML programmer to write a valid pointcut declaration. As depicted, the AspectJML programmer got an error from AJDT because he/she forgot to bind the formal parameters of the pointcut method declaration with the pointcut expression by using the argument-based pointcut **args**. The well-formed pointcut can be seen in Figure 4.5. All the AJDT IDE validation is inherited by AspectJML.

Note that the AJDT is just a helpful functionality to assist (beginners) AspectJML programmers to see where the specified pointcuts intercept. Once pointcut language and quantification mechanism are understood, this tool is not required to reason about



```
class Package {
    double width, height;
    //@ invariant this.width > 0 && this.height > 0;
    double weight;
    //@ invariant this.weight > 0;

    //@ requires width > 0 && height > 0;
    //@ requires width * height <= 400; // max dimension
    @Pointcut("execution(* *Size(double,double))")
    void sizes(double width, double height) {}

                                h, double height){

                                ;
```

```
Multiple markers at this line
  - formal unbound in
   pointcut
```

Figure 4.6: An example of a malformed pointcut declaration in AspectJML.

AspectJML in a modular way (as discussed in Section 4.4).

4.8 Chapter Summary

In this chapter, we presented the DbC language AspectJML. We demonstrated how AspectJML can be used to tackle the crosscutting contracts problem while keeping the benefits of a classical design by contract language such as JML or Eiffel. We showed how to deal with crosscutting pre- and postconditions using JML specifications. Also, exceptional postconditions are allowed to crosscutting contract specification. In addition to crosscutting modularity, we also discussed how the technique called client-aware checking can be used to provide runtime checking at client side while respecting privacy information in specifications which promotes information hiding. Since AspectJML is an extension to JML using some AspectJ features, a programmer can decide which features to use: (1) only JML annotated Java programs, (2) only AspectJ programs, (3) only Java programs, or (4) all of them mixed including the pointcut specification that is only available with AspectJML.

Chapter 5

Evaluation

This chapter reports our evaluation regarding the AspectJML language. Such an evaluation is important to provide some evidence that the modularity features available in AspectJML are useful for developers using design by contract in practice. Section 5.1 presents the selected case studies and Section 5.2 the main procedures as phases we follow to conduct our evaluation. In Section 5.3, we quantitatively evaluate the XCS feature of AspectJML through a set of metrics structured in the form of a Goal-Question-Metric (GQM) (see Table 5.1). All the results are available online at: `https://dl.dropboxusercontent.com/u/875595/phd.zip`.

5.1 Target Systems

The first major decision that we made in our investigation was the selection of the target systems. The three chosen systems are Health Watcher (HW) [SLB02, GBF+07], Java Card API 2.2.1 [Tea14a], and HealthCard [Tea14b]. The first one is a real web-based information system that allows citizens to register complaints regarding health issues. We selected this system because it is an open source project with a detailed requirements document available [GBF+07]. This requirements document describes 13 use cases and forms the basis for our JML specifications. In relation to the last two systems, the former was selected because it consists in a detailed specification of the Java Card (JC) API using JML (the specifications are available online[1]). The latter consists of a prototype of a Java Card application for smart cards called HealthCard (HC). It was selected because it is specified in JML (the specifications are also available online[2]) to fulfill the requirements of the HealthCard application in Java Card. The size of the selected systems can be observed by the lines of code metric in Table 5.2.

5.2 Study Phases

The study was divided into three major phases: (i) The specification of the Heath Watcher contracts using JML in conformance with its requirements document; (ii) the modularization of crosscutting contracts using AspectJML for the three target systems,

[1]`http://wwwhome.ewi.utwente.nl/~mostowskiwi/software.html`
[2] `http://sourceforge.net/projects/healthcard/`

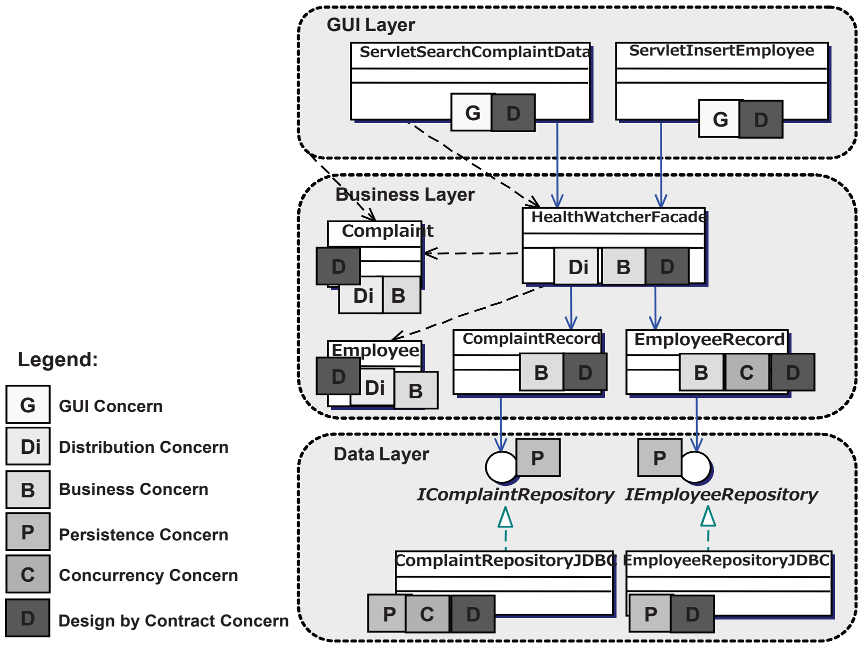


Figure 5.1: Health Watcher's Design including design by contract concern.

and (iii) the quantitative assessment of the JML and AspectJML versions of the three target systems.

Specification of the HW base release. In the first phase, we used JML to specify the contracts for the HW base release implemented in Java [GBF+07]. To create the JML specification, we analyzed the entire requirements document of the HW system. Such analysis was essential to understand the HW functionalities and involved actors. The inference of some contracts was not possible using only the requirements document. In these cases, we performed a deep analysis (inspection) of the HW source code. Figure 5.1 presents a partial class diagram of the HW base release implemented in Java and now specified with JML. Note that the concern DbC is now part of the system's diagram. As observed, the Layer architectural pattern [BMR+96] is used to structure the system classes in three main layers: GUI (Graphical User Interface), Business, and Data. The GUI layer implements a web user interface for the system. The Java Servlet API is used to codify the classes of this layer. The Business layer aggregates the classes that define the system business rules. Finally, the Data layer defines the functionality of database persistence using the JDBC API. Also, several design patterns [AB01, LASB01, GHJV95] are used in the design of the HW layers to achieve a reusable and maintainable implementation.

Crosscutting Contract Modularization using AspectJML. With the HW system specification in JML (in the previous phase), all the three target systems now present their own JML specifications for the design by contract concern. Hence, the second phase modularizes the crosscutting contracts contained in the selected systems. The modularization was performed by using AspectJML and its features presented in the last chapter.

Quantitative Assessment. The goal of the third phase was to compare in a quantitative way the design by contract modularity of JML and AspectJML versions of the chosen systems. The comparison was based on the size of the design by contract concern and compilation time.

5.3 Quantitative analysis

To drive the quantitative evaluation of our AspectJML language, we follow the Goal-Question-Metric (GQM) design [BCR94]. We structure it in Table 5.1. We answer Question 1 measuring the size of AspectJML features in terms of lines of code (LOC) [CK94], lines of DbC code (DbCLOC), number of preconditions (NOPre), number of postconditions (NOPo), number of operations (NOO) [CK94], and number of types (VS) [CK94]. To answer Question 2, we use instrumented source code (ISC) and instrumented bytecode (IBC) measures in megabytes to quantify the overhead of JML contract instrumentation. Also, we measure the compilation time (CT) overhead in both milliseconds and seconds.

We use the Google CodePro AnalytiX [3] to obtain the LOC, NOO, and VS metrics. In addition, we use sheets to help the computation of the other metrics, which were collected manually. In order to gather the compilation time information, we use the ajmlc compiler with the option −verbose enabled. This option prints in the end the total compilation time in both milliseconds and seconds. For each system, we used 15 samples and employed the T-Test. All the compilation time results, we present, have a 95% of confidence.

5.3.1 System size results

To answer Question 1 and investigate whether our AspectJML XCS feature increases the overall system size, we use the size metrics LOC, DbCLOC, NOPre, NOPo, NOO, and VS. We use this set of size metrics to evaluate the selected three systems. The results are illustrated in Table 5.2.

LOC and DbCLOC

As observed, the use of AspectJML XCS feature increases the lines of code (LOC) for all analyzed systems. This is expected since to write crosscutting contract specifications, we need to write pointcut methods with specifications. These pointcut methods are counted as extra lines of code. Since JML specifications are written in the form of Java comments, the LOC metric exclude them. Because of that, we use a specific metric to count all DbC related code (including JML specifications). In this case, we have reduced the overall system DbCLOC after crosscutting contract modularization with AspectJML (see Table 5.2). This happens since we locate all the recurrent specifications in one place with a pointcut-specification mechanism of AspectJML.

NOPre and NOPo

In Table 5.2 we can observe that by using AspectJML we can significantly reduce the overall number of system pre- and postconditions. The more scattered a recurring

[3] `https://developers.google.com/java-dev-tools/download-codepro`.

Table 5.1: GQM

Goal	
Purpose	Evaluate AspectJML regarding
Issue	size and compilation size and time of its crosscutting contract specification (XCS)
Object	from a
Viewpoint	software engineer viewpoint
Questions and Metrics	
Q1- Does AspectJML increase the system size regarding lines of code, lines of DbC code (specs), number of preconditions, number of postconditions, number of operations, and number of types?	
Lines of Code	LOC [CK94]
Lines of DbC code	DbCLOC
Number of Preconditions	NOPre
Number of Postconditions	NOPo
Number of Operations (methods or pointcuts) of each type	NOO [CK94]
Vocabulary Size (number of types)	VS [CK94]
Q2- Does AspectJML increase code instrumentation before and after compilation, and increase compilation time?	
Instrumented Source Code	ISC
Instrumented Bytecode	IBC
Compilation Time	CT

specification is, the better reuse and less lines we have when employing AspectJML. Note that the metric NOPo denotes both normal and exceptional postconditions expressed in JML.

NOO and VS

As expected, the use of AspectJML to specify and modularize crosscutting contracts increase the overall system number of operations (NOO) and number of types (VS). The number of operations is increased due to methods added in each studied system. These methods are pointcut methods which contains JML specifications for crosscutting contracts. For instance, in the HC system, we had an increasing of number of operations in 9.47% (see Table 5.2). So the 36 extra methods denote pointcut methods written in AspectJML for HC system.

In relation to VS metric, it shows us that the HC system had an increasing of 25% in terms of number of types. This happens in AspectJML when one modularize crosscutting contracts behind an interface and applies the crosscutting functionality by implementing this interface. Hence, each interface should be counted as a new type in the system. Moreover, due to some limitations discussed in previous chapter, each interface, which implements a crosscutting contract, needs an inner class to encapsulate its crosscutting contract. In this context, each inner class is computed as additional types as well. Therefore, each crosscutting contract interface adds two new types in the system.

5.3.2 Compilation System results

This section answers Question 2 by investigating the extent of compilation overhead when using AspectJML. Thus, we measure the compilation overhead in terms of code instrumentation and compilation time. For code instrumentation we employ the metrics ISC and IBC (see Table 5.1). The former denotes the source code generated to check

Table 5.2: LOC, DbCLOC, NOPre, NOPo, NOO, and VS metric results for all systems.

Metric		**LOC**	**LOCDbC**	**NOPre**	**NOPo**	**NOO**	**VS**
Before (HW) Modularization	JML	5916	2129	363	778	456	89
	AspectJML	-	-	-	-	-	-
	Total	**5916**	**2129**	**363**	**778**	**456**	**89**
After (HW) Modularization	JML	5916	1459	222	419	456	89
	AspectJML	267	236	24	32	45	18
	Total	**6183**	**1695**	**246**	**451**	**501**	**107**
	Diff.	**+4.32%**	**-20.39%**	**-32.24%**	**-42.03%**	**+8.98%**	**+16.82%**
Before (JC) Modularization	JML	3224	2148	427	347	392	89
	AspectJML	-	-	-	-	-	-
	Total	**3224**	**2148**	**427**	**347**	**392**	**89**
After (JC) Modularization	JML	3224	1782	246	264	392	89
	AspectJML	93	126	15	20	12	10
	Total	**3317**	**1908**	**261**	**284**	**404**	**99**
	Diff.	**+2.80%**	**-11.17%**	**-38.87%**	**-18.15%**	**+2.97%**	**+10.10%**
Before (HC) Modularization	JML	1752	2424	257	333	344	36
	AspectJML	-	-	-	-	-	-
	Total	**1752**	**2424**	**257**	**333**	**344**	**36**
After (HC) Modularization	JML	1752	2249	250	191	344	36
	AspectJML	99	69	1	14	36	12
	Total	**1851**	**2318**	**251**	**205**	**380**	**48**
	Diff.	**+5.34%**	**-4.37%**	**-2.33%**	**-38.43%**	**+9.47%**	**+25%**

JML contracts but before compilation. The latter denotes the bytecode instrumentation after compilation. Both are measured in megabytes. In relation to compilation time (CT), we present the measures in both milliseconds and seconds. The results are illustrated in Table 5.3.

Code instrumentation

For the systems studied, we can observe that by using AspectJML to modularize crosscutting contracts creates a slight impact on the size of code instrumentation both before and after compilation (see Table 5.3). The only exception, in terms of source code instrumentation, was the HC system. Our conjecture to this happen is when one modularize crosscutting contracts and there is no remaining specifications in the methods whose recurrent specifications were removed to pointcut methods. In this case the number of AspectJ advice generated to check contracts are reduced.

On the other hand, the number of generated AspectJ advice could be greater than usual because after modularization, we sill have specifications in the methods whose recurrent specifications were removed to pointcut methods. This situation corroborates to other two analyzed systems (HW and JC).

Compilation time

Table 5.3 also presents the results for compilation time. As observed, a system using AspectJML to handle crosscutting contracts, presents a small overhead in terms of the time needed to complete the system's compilation. For the three studied systems, we had an overall overhead of 4 seconds to compile a system. This is expected since in the end we have more AspectJ advice to compile. Also, this extra advice contains quantification that intercepts and instruments several join points. So, this also contributes to an increasing in the overall system's compilation time.

More compilation time

As discussed, we observed a small overhead in a system in relation to compilation time when using AspectJML XCS feature. But one question that can arise is whether this AspectJML's feature (when enabled) can increase the compilation time of a system without any JML specifications? To answer this question, we selected three more open-source systems available online[4]:

- Database schema visualizer called **dbviz** implemented in Java (version 0.5) with 6.714 of lines of code (LOC);
- JavaScript compiler/interpreter called **Rhino** implemented in Java (version 1.5) with 37.960 of LOC;
- Object-relational mapping tool called **iBATIS** implemented in Java (version 2.3) with 15.855 of LOC.

We evaluated these systems as before. That is, in terms of compilation time in both milliseconds and seconds, and we use the ajmlc's compiler −verbose option enabled to collect the compilation time in both milliseconds and seconds. We ran eight times each system: four with XCS disabled and another four with XCS enabled. Then, we took the mean of the compilation time results for each version (e.g., XCS enabled). The results is presented in Table 5.4.

Comparing the compilation time results of Table 5.4 with Table 5.3, we can conclude that the AspectJML/ajmlc compiler, where we do not have JML specifications at all, behaves in the same manner when XCS is enabled or not. We have minor differences to argue that one is better than other and vice versa. For example, in the dbviz system, we had better results for XCS (only 4% of reduction in seconds). On the other hand, considering the iBATIS system, we had better results for non-XCS (but only 1.66% of reduction in seconds). The Rhino system showed a somewhat equivalence between non-XCS and XCS compilation modes. In terms of seconds, they presented the same compilation time. Only when we consider milliseconds we can observe that non-XCS is better, but only 0.75% of reduction.

5.3.3 What about Scattering and Tangling metrics?

Since this empirical study is related to modularity, one natural question is why the separation of concern metrics [GSF+05, EZS+08] which mainly includes scattering and tangling, were not used? These metrics have been successfully applied to similar studies [GSF+05, KSG+06, FCF+06, GBF+07, CRG+08, EZS+08, ARG+11]. The answer to this question is based on how AspectJML deals with contracts in general. For example, if we consider the degree of the scattering metric over classes (DOSC) [EZS+08], even if after modularization, we can see worse results than before. The reason is that to

[4] http://www.cs.columbia.edu/~eaddy/concerntagger/

Table 5.3: ISC, IBC, and CT metric results for all systems.

Metric		**ISC (MB)**	**IBC (MB)**	**CT (ms)**	**CT (s)**
HealthWatcher	JML	3.30	6.32	25231	25
	AspectJML	3.37	6.83	30259	30
	Diff.	**+0.07**	**+0.51**	**+5028**	**+5**
	Diff. (%)	**+2.07%**	**+7.46%**	**+16.61%**	**+16.66%**
Java Card API	JML	2.52	4.96	25856	25
	AspectJML	2.54	5.37	28939	28
	Diff.	**+0.02**	**+0.41**	**+3083**	**+3**
	Diff. (%)	**+0.78%**	**+7.63%**	**+10.65%**	**+10.71%**
HealthCard	JML	2.14	9.08	25968	25
	AspectJML	2.02	9.16	29682	29
	Diff.	**-0.12**	**+0.08**	**+3714**	**+4**
	Diff. (%)	**-5.60%**	**+0.87%**	**+12.51%**	**+13.79%**

Table 5.4: CT metric results for dbviz, Rhino, and iBATIS systems.

Metric		**CT (ms)**	**CT (s)**
dbviz	XCS-disabled	25053	25
	XCS-enabled	24111	24
	Diff.	**-942**	**-1**
	Diff. (%)	**-3.76%**	**-4%**
Rhino	XCS-disabled	61525	61
	XCS-enabled	61863	61
	Diff.	**+338**	**0**
	Diff. (%)	**+0.54%**	**0%**
iBATIS	XCS-disabled	59667	59
	XCS-enabled	60123	60
	Diff.	**+456**	**+1**
	Diff. (%)	**+0.75%**	**+1.66%**

keep the documentation benefits, JML does not extract all the contracts to one place (e.g., type or method). Hence, the additional operations and crosscutting interfaces for crosscutting contract specifications can present more scattering than before the modularization. The same principle applies to metrics considering tangling. AspectJML does not eliminate tangling in its standard use.

Fortunately, AspectJML can help one eliminate the amount of specifications that are placed throughout the source code. As discussed in Section 2.4.10, JML specifications can be written in a separate corresponding file. Thus, instead of writing JML specifications in a file say Package.java, we write them in a corresponding file named, for example, Package.jml. As mentioned, this feature is also provided in AspectJML, and once used, it removes the tangling and scattering we have before in original source files. Therefore, the results of such separation of concern metrics are directly dependent of the use of separate files in AspectJML. This is somewhat like aspects (modular units to separate crosscutting functionality). If we use separate files, we have good modularity indicators through the set of separation of concern metrics, whereas we do not if we do not use separate files for specifications. This justifies why we are not considering these metrics (although useful) in our study. Considering our previous target applications, the JML specifications written for the JavaCard API 2.2.1 were written in separate .jml files

5.4 Representative Crosscutting Scenarios

This section presents some examples of scenarios that were identified during the process of refactoring the systems described in Section 5.1 with AspectJML. These scenarios represent recurring situations in which a developer would have to deal if faced with the task of modularizing design by contract code using AspectJML/XCS. We mainly discuss in our experience in detail using the HW system. We give more attention to this system because it was studied by using both requirements document and code analysis. Hence, we analyzed the crosscutting contract structure of the HW system, comparing its specification in JML and AspectJML. Even though the discussion is HW driven, we also provide some scenarios relative to the other two studied systems (Section 5.1).

5.4.1 Understanding the Crosscutting Contract Structure

One of the most important steps in the evaluation is to recognize how the contract structure crosscuts the modules of the HW system. We now show some of the crosscutting contracts present in HW using the standard JML specifications.

Crosscutting preconditions

Crosscutting preconditions occur in the HW system's IFacade interface. This facade makes available all 13 use cases as methods. Consider the following code snippet from this interface:

```
//@ requires code >= 0;
IteratorDsk searchSpecialitiesByHealthUnit(int code);

//@ requires code >= 0;
Complaint searchComplaint(int code);

//@ requires code >= 0;
DiseaseType searchDiseaseType(int code);

//@ requires code >= 0;
IteratorDsk searchHealthUnitsBySpeciality(int code);

//@ requires healthUnitCode >= 0;
HealthUnit searchHealthUnit(int healthUnitCode);
```

These methods comprise all the search-based operations that HW makes available. The preconditions of these methods are identical, as each requires that the input parameter, the code to be searched, is at least zero. However, in plain JML one cannot write a single precondition for all 5 search-based methods.

Crosscutting postconditions

Still considering the HW's facade interface IFacade, we focus now on crosscutting postconditions. First, we analyze the crosscutting contract structure for normal postconditions:

```
//@ ensures \result != null;
IteratorDsk searchSpecialitiesByHealthUnit(int code);
```

```
//@ ensures \result != null;
IteratorDsk searchHealthUnitsBySpeciality(int code);

//@ ensures \result != null;
IteratorDsk getSpecialityList()

//@ ensures \result != null;
IteratorDsk getDiseaseTypeList()

//@ ensures \result != null;
IteratorDsk getHealthUnitList()

//@ ensures \result != null;
IteratorDsk getPartialHealthUnitList()

//@ ensures \result != null;
IteratorDsk getComplaintList()
```

As observed, all the methods in IFacade that return an object of type IteratorDsk should return a non-null object reference. In standard JML there are two more ways to express this constraint [CJ07]. The first one, as discussed in Section 2.4.4, uses the non-null semantics for object references. In this case we do not need to write out such normal postconditions to handle non-null. However, we can deactivate this option in JML if most reference types in the system are possibly null or just by the programmer's decision. In this scenario, whenever we find a method that should return non-null, we still need to write these normal postconditions. So, by assuming that we are not using the non-null semantics of JML as default, these postconditions become redundant. The second option is to use the JML type modifier **non_null**; however, even this would lead to several repeated annotations.

With respect to exceptional postconditions of IFacade interface, we found an interesting crosscutting structure scenario. Consider the following code:

```
//@ signals_only java.rmi.RemoteException;
void updateComplaint(Complaint q) throws
  java.rmi.RemoteException, ...;

//@ signals_only java.rmi.RemoteException;
IteratorDsk getDiseaseTypeList() throws
  java.rmi.RemoteException, ...;

//@ signals_only java.rmi.RemoteException;
IteratorDsk getHealthUnitList() throws
  java.rmi.RemoteException, ...;

//@ signals_only java.rmi.RemoteException;
int insertComplaint(Complaint complaint) throws
  java.rmi.RemoteException, ...;

... // the other 12 facade methods contain this constraint
```

As can be seen, these IFacade methods can throw the Java RMI exception RemoteException (see the methods throws clause). This exception is used as a part of the Java RMI API used by the HW system. Even though we list only four methods, all the methods contained in the IFacade interface contain this exception in their throws clause. Because of that, the **signals_only** clause shown needs to be repeated for all methods in the IFacade interface. However, in JML one cannot write a single **signals_only** clause to constrain all such methods in this way.

Another example of exceptional postconditions occurs with the search-based methods discussed previously. All these search-based methods should have a **signals_only** clause that allows the ObjectNotFoundException to be thrown. As with the RemoteException, one cannot write this specification once and apply it to all search-based methods.

5.4.2 Modularizing Crosscutting Contracts in HW

To restructure/modularize the crosscutting contracts of the HW system, we use the XCS mechanisms of AspectJML. By doing this, we avoid repeated specifications, which is an improvement over standard DbC mechanisms. In the following we show the details of how AspectJML achieves a better separation of the contract concern for this example.

Specifying crosscutting preconditions

We can properly modularize the crosscutting preconditions of HW with the following JML annotated pointcut in AspectJML:

```
//@ requires code >= 0;
@Pointcut("execution(* IFacade.search*(int))"+
 "&& args(code)")
void searchMeths(int code) {}
```

With this pointcut specification, we are able to locate the preconditions for all the search-based methods in a single place. To select the search-based methods, we use a property-based pointcut [KHH+01] that matches join points by using wildcarding. Our pointcut matches any method starting with search and taking an **int** parameter. Before the executions of such intercepted methods, the precondition that constrains the code argument to be at least zero is enforced during runtime; if it does not hold, then one gets a precondition violation error.

Specifying crosscutting postconditions

Consider the modularization of the two kinds of crosscutting postconditions we discussed previously. For normal postconditions, we add the following code in AspectJML:

```
//@ ensures \result != null;
@Pointcut("execution(IteratorDsk IFacade.*(..))")
void nonNullReturnMeths() {}
```

With this pointcut specification, we are able to explicitly modularize the non-null constraint. The pointcut expression we use matches any method with any list of parameters returning IteratorDsk.

The AspectJML code responsible for modularizing the exceptional postconditions is similar:

```
//@ signals_only java.rmi.RemoteException;
@Pointcut("execution(* IFacade.*(..))")
void remoteExceptionalMeths() {}
```

```
//@ signals_only ObjectNotFoundException;
@Pointcut("execution(* IFacade.search*(..))")
void objectNotFoundExceptionalMeths() {}
```

These two specified pointcuts in AspectJML are responsible for modularizing the exceptional postconditions for methods that can throw RemoteException and methods that can throw ObjectNotFoundException, respectively. The first pointcut applies the specification for all methods in IFacade, whereas the second one intercepts just the search-based methods.

5.4.3 Reasoning About Change

The main benefit of AspectJML is to allow the modular specification of crosscutting contracts in an explicit and expressive way. The key mechanism is the quantification property inherited from AspectJ [KHH+01]. In addition to the documentation and modularization of crosscutting contracts achieved by using AspectJML, another immediate benefit of using our approach is easier software maintenance.

For example, if we add a new exception that can be thrown by all IFacade methods, instead of (re)writing a **signals_only** clause, we can add this exception to the **signals_only** list of the remoteExceptionalMeths pointcut. This pointcut can be reused whenever we want to apply constraints to methods already intercepted by the pointcut.

Another maintenance benefit occurs during system evolution. On one hand, we may add more methods in the IFacade interface to handle system's new use cases. On the other hand, we do not need to explicitly apply existing constraints to the newly added methods. The modularized contracts that apply to all methods also automatically apply to the newly added ones, with no cost. Finally, even if the crosscutting contracts are well documented by using JML specifications, the AJDT tool helps programmers to visualize the overall crosscutting contract structure. Just after a method is declared, we can see which crosscutting contracts apply to it through the cross-references feature of AJDT [KM05].

5.4.4 More Crosscutting Scenarios and their Modularization

In the following we present some code snippets that illustrate the crosscutting structure present in the JavaCard API 2.2.1 and in the HealthCard system. We also show how we modularized them with AspectJML. We give no much details about the code. We just looked for specification clones for modularization.

Crosscutting contracts in JavaCard

Consider the interface Service part of JavaCard API:

```
public interface Service {
  /*@ public normal_behavior
    @  requires apdu != null;
    @  ensures true;
    @  assignable apdu._buffer[*];
    @*/
  public boolean processDataIn(APDU apdu);
  /*@ public normal_behavior
    @  requires apdu != null;
    @  ensures true;
    @  assignable apdu._buffer[*];
    @*/
  public boolean processCommand(APDU apdu);

 /*@ public normal_behavior
   @  requires apdu != null;
   @  ensures true;
   @  assignable apdu._buffer[*];
   @*/
  public boolean processDataOut(APDU apdu);
}
```

As observed, the three interface methods have exactly the same JML specification case. Consider now the same interface specifications using AspectJML:

```
public interface Service {
  /*@ public normal_behavior
    @  requires apdu != null;
    @  ensures true;
    @  assignable apdu._buffer[*];
    @*/
  @Pointcut("execution(public boolean process*(javacard.framework.APDU))" +
            "&& args(apdu)")
  public void processXCS(APDU apdu);

  public boolean processDataIn(APDU apdu);

  public boolean processCommand(APDU apdu);

  public boolean processDataOut(APDU apdu);
}
```

The pointcut method processXCS is responsible for modularizing the crosscutting contracts. Now, instead of 15 lines of DbC code, we just have five. The pointcut mainly selects the three interface methods by using process∗, which denotes any method starting with process.

As discussed in Chapter 4, this crosscutting contract specification for the interface Service does not need an inner class since it is supposed to be declared within a separate .jml file. For a public method and within a .jml file, we do not need to create an inner class to modularize the interface's crosscutting contracts.

Crosscutting contracts in HealthCard

To exemplify crosscutting contracts for HealthCard system, consider the following interface methods and their specifications:

```
//@ ...
//@ signals_only UserException;
//@ signals (UserException e)
//@   medicines_model.equals(\old(medicines_model));
public byte addMedicine (...) throws UserException;
```

```
//@ ...
//@ signals_only UserException;
//@ signals (UserException e)
//@  medicines_model.equals(\old(medicines_model));
public void removeMedicine (...) throws UserException;

//@ ...
public short countAllMedicines () throws UserException;

... /* more 12 interface methods with the same
     * crosscutting specifications presented
     * by both addMedicine and removeMedicine */
```

This crosscutting scenario is presented in the interface Medicines. As illustrated, the interface consists in 15 methods in which 13 presents the same crosscutting structure involving JML exceptional postconditions. Only the method countAllMedicines present different specifications and is not a part of our crosscutting scenario.

An expected solution is an AspectJML pointcut method to modularize both **signals_only** and **signals** crosscutting clauses. However, we break down our solution in two parts since the **signals_only** clauses are crosscutting in other unrelated types. Hence, we have a separate crosscutting interface useful for all types that present this recurrent crosscutting contract. See crosscutting interface UserExceptionAllowedSignallingConstraint below:

```
public interface UserExceptionAllowedSignallingConstraint {
 @InterfaceXCS
 class UserExceptionAllowedSignallingConstraintXCS{
  //@ signals_only javacard.framework.UserException;
  @Pointcut("execution(* UserExceptionAllowedSignallingConstraint+.*(..)) && "
   + "!@annotation(cc.UserExceptionAllowedSignallingConstraint.ExcludeMarker)")
  public void userExceptionAllowedXCS(){}
  }
  public @interface ExcludeMarker {}
}
```

We use the standard way to add crosscutting contracts for an interface. Note the inner class declaration. The pointcut userExceptionAllowedXCS is responsible to add the above **signals_only** clause to all methods in a type that implements the interface UserExceptionAllowedSignallingConstraint. But the pointcut expression also says to exclude any member annotated with ExcludeMarker (see the use of **@annotation** pointcut). This is useful when not all methods should present the crosscutting functionality. We just need to implement the crosscutting interface in addition to mark all methods that should be excluded from the pointcut userExceptionAllowedXCS. So assuming that the interface medicines implements the crosscutting one UserExceptionAllowedSignallingConstraint, we have the following change:

```
//@ ...
//@ signals (UserException e)
//@  medicines_model.equals(\old(medicines_model));
public byte addMedicine (...) throws UserException;

//@ ...
//@ signals (UserException e)
//@  medicines_model.equals(\old(medicines_model));
```

```
public void removeMedicine (...) throws UserException;

//@ ...
@UserExceptionAllowedSignallingConstraint.ExcludeMarker
public short countAllMedicines () throws UserException;

... /* more 12 interface methods with the same
     * crosscutting specifications presented
     * by both addMedicine and removeMedicine */
```

In our first step, we removed all **signals_only** clauses modularized by implementing the above crosscutting interface. Moreover, note that we exclude the method countAllMedicines by adding the annotation ExcludeMarker on it. Let us now modularize the remaining crosscutting contract denoted by the JML **signals** clauses. See our second (and final) step below:

```
@InterfaceXCS
class MedicinesXCS{
 //@ signals (UserException e)
 //@  medicines_model.equals(\old(medicines_model));
 @Pointcut("execution(public * *Medicine*(..)) &&" +
     "!execution(public short countAllMedicines())")
 public void medicinesSignalsXCS(){}
}

//@ ...
public byte addMedicine (...) throws UserException;

//@ ...
public void removeMedicine (...) throws UserException;

//@ ...
@UserExceptionAllowedSignallingConstraint.ExcludeMarker
public short countAllMedicines () throws UserException;

... /* more 12 interface methods with the same
     * crosscutting specifications presented
     * by both addMedicine and removeMedicine */
```

Since the type Medicines is an interface, we add an inner class to modularize its crosscutting contracts. So the pointcut medicinesSignalsXCS intercepts all methods in interface Medicines but excludes the execution of countAllMedicines (which does not present crosscutting contracts).

5.4.5 Study Constraints

In what follows, we present the threats to validity of our study.

External validity. The scope of our experience is limited to Java, JML, and AspectJ languages. With respect to design by contract features, our experience about crosscutting contract modularization only considered the implementation of pre- and postconditions. Our results may potentially generalize to other OO and AO languages and design by contract features, though that requires further analyses.

Internal validity. The authors implemented the contracts in the HW system, which is a threat to internal validity. However, we minimize this threat since we followed the

HW requirements documentation, which contains information with respect to pre- and postconditions. We complemented our specification with code analysis. Hence, we implemented all contracts according to the documentation [GBF+07] and current code. As explained, the existence of this requirement document was one of the main reasons to use HealthWatcher in our analysis. Another reason to minimize this internal threat is that we used other applications which already came with JML specifications.

Conclusion validity. The applicability, usefulness, and representativeness of the set of metrics used in this study can be questioned. For example, we did not use the specific set of metrics for separation of concerns. This set of metrics have already been proved to be effective quality indicators in several case studies [GSF+05, KSG+06, FCF+06, GBF+07, CRG+08]. As discussed in Section 5.3.3, we explained that the use of this set of separation of concern metrics can diverge depending on how we use AspectJML. As such, we just leave those metrics that can really reveal some indications of the usefulness and drawbacks of the modularization of crosscutting contracts with AspectJML.

5.5 Chapter Summary

In this chapter, we conducted an evaluation to assess the expressiveness of AspectJML to handle crosscutting contracts in practice. To this end we quantitatively evaluated three real systems with metrics to gather information about the impact of crosscutting contract modularization. We concluded that by using AspectJML a programmer can significantly reduce the overall DbC lines of code of crosscutting contracts. However, we also observed a small overhead of XCS feature in terms of code instrumentation and compilation time. Also, we demonstrated several scenarios in which pointcuts-specifications written in AspectJML could be beneficial to the day-by-day development. We also discussed that when maintenance tasks are performed, the effort to specify new methods could be minimized due to the use of AspectJML pointcuts-specifications that could automatically apply to such new methods.

Chapter 6

Related Work

In this chapter, we present the main works related to this thesis.

6.1 Empirical Evidence About Crosscutting Contracts

Chalin *et al.* [CR05a, CR05b, CJ07, CJR08] conducted the most well-known study involving the non-null annotation crosscutting concern. They argue that specifying moderately large code bases, the use of non-null annotations is more intensive than it should be. Hence, a programmer needs to scatter non-null annotations repeatedly in several types. As a result, programmers may forget to add some non-null annotation to some reference type, thus leading clients to call methods with null arguments resulting in NullPointerException. In order to confirm this hypothesis, they conducted a study with 5 open source projects totaling 700 KLOC which confirms that on average, 75% of reference declarations are meant to be non-null, by design. With their findings, Chalin *et al.* proposed to change the JML semantics by allowing reference type declarations to be **non_null** by default. Since then, JML adopted the **non_null** semantics by default.

Another study conducted to assess the crosscutting structure of DbC was performed by Lippert and Lopes [LL00]. They used AO techniques to modularize design by contract features such as pre- and postconditions in a large OO framework, called JWAN. Also, they attempted to identify situations where it was easy to aspectize design by contract code. These situations were only two: (1) for preconditions, they wanted to make sure those objects references, used as parameters, are non-null, and (2) ensure that the return object reference for any method is non-null. This scenario is similar to non-null annotations studied by Chalin *et al.* [CR05a, CR05b, CJ07, CJR08]. The only problem with this study was that they only investigated scenarios where the use of AOP is good for. But, the study lack results to show where is not advantageous to aspectize DbC.

In this context, Rebêlo *et al.* [RLK$^+$13] complements the Lippert and Lopes' work since they also considered heterogenous contracts. Moreover, Lippert and Lopes' work only considered pre- and postconditions, whereas Rebêlo *et al.* also include invariants in the analysis. Finally, Rebêlo *et al.*'s work was the first to include a quantitative and qualitative analysis of the aspectization of design by contract concern and also with an analysis of the implementations regarding modularity and change propagation. Their main findings indicate that aspectual decompositions are superior especially when considering the Open-Closed principle. In certain circumstances aspectual decompositions tended to propagate to unrelated components due to ripple-effects caused by OO refactorings. In addition, even with higher reuse, AO implementations tended to present no significant gains regarding system and design by contract size in relation to OO decompositions as usually advertised by the literature [LL00, LLH02, KHH+01].

6.2 Crosscutting Contract Modularization

As discussed throughout the thesis, there are several works in the literature that argue in favor of implementing DbC with AOP [KHH+01, LLH02, F+06, RLL11]. Kiczales opened this research avenue by showing a simple precondition constraint implementation in one of his first papers on AOP [KHH+01]. In addition, in an interview, Kiczales explicitly cites DbC as an example of a crosscutting concern:

> *"[...] there are many other concerns that, in a specific system, have crosscutting structure. Aspects can be used to maintain internal consistency among several methods of a class. They are well suitable to enforcing a Design by Contract style of programming" [Kic03].*

After that, Lopes *et al.* also patented this idea [LLH02].

Briand [BDL05] discusses how to implement contracts with Aspect-Oriented Programming (AOP) using AspectJ. He defines how to efficiently instrument contracts and invariants in Java. The two main objectives of this work are: (1) to work at bytecode level avoiding polluting the source code; (2) to apply the Liskov Substitution Principle (LSP) [LW94] in order to check inheritance hierarchies.

Diotalevi [Dio04] proposes an approach for adopting design by contract [Mey92a] in the development of Java application using Aspect-Oriented Programming (AOP) with AspectJ. The work states that inserting pre- and postconditions assertions directly into the application code has serious drawbacks — in terms of code modularity, reusability, and cohesion — that lead to a common OO limitation called tangled code. These assertions are *crosscutting concerns* and mix business logic code with the nonfunctional code that assertions require; they are inflexible because we cannot change or remove assertions without updating the application code. Because of that, the work provides a solution based on the following four requirements:

- transparency — the pre- and postconditions code is not mixed with business logic;
- reusability — most of the solution is reusable;
- flexibility — the assertion modules can be added to, removed, and modified;
- simplicity — assertions can be specified using a simple syntax.

This solution has a "bridge" that is an AspectJ aspect. This aspect specifies the exact point where the contract is to be applied. The AspectJ implementation of contracts covers pre-, postconditions and invariant checks. As a result, this solution provides a clean and flexible solution, because it eliminates the drawbacks previously mentioned. The solution lets one code the contracts of the application separately (untangled) from one's business logic. Different from our approach, this work concentrates only on pre-, (normal) postcondition, and (instance) invariants.

"Assertion with Aspect" [IKKI04, IKIK05] proposes aspects for implementing assertions. The work aims to use AspectJ in order to inject `assert` statements into classes — the `assert` statement is a Java function of the standard library (available since JDK 1.4). Finally, this work enhances reusability of Java programs by eliminating tangled code with assert statements. As a result, programmers can add and remove assertions (contracts) of a class, thus enabling the programmers to separate (untangled) a class from its assertions. The proposed work differs from ours in that it covers only two types of assertions, precondition and (normal) postcondition assertions, and does not consider the inheritance of assertions.

In relation to the works discussed above, Balzer, Eugster, and Meyer argued that DbC aspectization is more harmful than good [BEM05], since one loses all the key properties of a DbC language: documentation, specification inheritance, and modular reasoning. Indeed, they argue that aspect interaction can make even worse the understanding of how contracts are checked, and in what order they are checked. They conclude that contracts are not crosscutting and that AOP cannot emulate contracts properly. Rebêlo, Lima, and Leavens' work contradict some of the finding by Balzer, Eugster, and Meyer. First of all, they show that AspectJ-like languages are suitable to enforce design by contract. They also show that with AspectJ we can implement and check properly contract inheritance. Finally, Rebêlo, Lima, and Leavens disagree with Balzer, Eugster, and Meyer's work in relation to the crosscutting structure of contracts. They believe and share the opinion of others [KHH$^+$01, LL00, LLH02, SL04, MMvD05, BDL05, LKR05, F$^+$06, RLL11, RLK$^+$13] that contracts are crosscutting. What is questionable here is if AspectJ-like languages are suitable to provide all benefits of a DbC language besides crosscutting modularity.

In this thesis, we go beyond these works by showing how to combine the best design features of a design by contract language like JML and the quantification benefits of AOP [FF00, VCFS10] such as AspectJ. As a result we conceive the AspectJML specification language that is suitable for specifying crosscutting contracts and preserving information hiding. In AspectJML, one can specify crosscutting contracts in a modular way while preserving key DbC principles such as documentation and modular reasoning [BEM05].

Moxa [YW06] is a behavioral interface specification language (BISL), which is an extension to JML, with a new modularization mechanism called assertion aspect. It captures the crosscutting properties among assertions. Like AspectJML, Moxa uses JML specifications to modularize crosscutting contracts. The problem is that they separate such contracts in a new module called assertion aspect. This brings back the obliviousness issue and compromises modular reasoning and documentation. Although the contracts are written in JML, a programmer cannot reason about a method's correctness with such a documentation in a modular way. This is even worse because Moxa does not use the AspectJ/AJDT IDE validation as we do with AspectJML. Therefore, all the crosscutting contract structure information is not available. A programmer does need to look at all assertion aspect modules to understand the use/correctness about a method. As such, this approach tends to help much less than the traditional AspectJ syntax to write contracts. Another major difference is that Moxa does not support pointcut specification as we do in AspectJML. For example, the wildcarding to apply to several join points is not supported. Moreover, named pointcuts are useful to reuse to other crosscutting contract specification. In Moxa, we need to list all methods with their corresponding syntactic interface all the time. Moxa does only support crosscutting contracts for preconditions and normal postconditions using JML. In AspectJML we support both in addition to exceptional postconditions and specification inheritance. Finally, similar to AspectJML, Moxa translates such crosscutting contract specifications written in JML to AspectJ aspects responsible to check them during runtime.

As with our work with AspectJML, Lam, Kuncak, and Rinard [LKR05] advocates that previous research in the field of aspect-oriented programming has focused on the use of aspect-oriented concepts in design and implementation of crosscutting concerns. Their experience indicates that aspect-oriented concepts can also be extremely useful for specification, analysis, and verification. In this sense, among other things, Lam, Kuncak, and Rinard designed constructs called *scopes* and *defaults* that can be used to improve the locality and clarity of specifications, and, at the same time, reducing the sizes of these specifications. These constructs cut across the preconditions and postconditions of procedures in a system. Like our work, their language provides a pointcut language to select where to apply constraints. The main difference is that their pointcut is not expressive as ours (since we reused the standard AspectJ pointcut language). For instance, in their language we cannot use wildcarding to select join points. Also, their pointcut language can apply to several modules, but this can break modular reasoning as we preserve using AspectJML. Finally, their work is intended to specification and verification, whereas we are concerned to specification and runtime checking.

6.3 Crosscutting Modularity and Modular Reasoning with Design Rule Languages

AOP is a popular technique for modularizing crosscutting concerns. However, constructs aimed at supporting crosscutting modularity may break class modularity. For example, to understand a method call may require a whole-program analysis to determine what advice applies and what that advice does. Moreover, in AspectJ, advice is coupled to the parts of the program advised, the base code, so the meaning of advice may change when the base code changes. Such coupling also hinders parallel development between base code and aspects.

In this context, several works based on design rules and/or specifications have been proposed to solve mainly the modular reasoning problem of aspect-oriented languages [GSS+06, SGR+10, BRLM11, BRD13, CNBR+13, RLL+13b]. Therefore, now it is possible to modularly reason about a modularized crosscutting concern that uses AspectJ-like features such as pointcuts and advice. However, the main problem in terms of DbC modularization is that such languages cannot provide crosscutting contract modularization while keeping the main DbC benefits such as documentation. Also, to use such languages for DbC modularization, we need more AOP features than those we need with AspectJML. Hence, AspectJML is simpler compared to those languages for DbC modularization and documentation.

6.4 Design by Contract Languages and Generative Programming

Feldman [F$^+$06] presents a design by contract (DbC) [Mey92a] tool for Java, know as *Jose*. This tool provides a DSL for a DbC language for expressing contracts. Similar to our AspectJML, Jose adopts AspectJ for implementing and checking contracts during runtime. The semantics of postconditions and invariants in Jose are distinct from JML. Jose states that postconditions are simply conjoined without taking into account the corresponding preconditions. Moreover, it establishes that private methods can modify invariant assertions. In the JML semantics, if a private method violates an invariant, an exception must be thrown. Only helper methods can violate invariants. Concerning recursive assertion checking, Jose, like Eiffel [Mey92b], only allows one level of assertion checking. To enforce this policy, Jose uses the control-flow based pointcut (using the AspectJ designator `cflowbelow`). We can do the same with AspectJML, but by default recursive assertion checking is allowed.

Contract4J [Wam06] is another AspectJ implementation of contracts. It is an open-source tool that supports design by contract [Mey92a] for Java. Like JML, Contract4J allows programmers to specify contracts as annotations. As with our crosscutting contract specification in AspectJML, their annotations are based on Java 5 annotations. The @Contract annotation signals that a class has a contract specification defined. Furthermore, Contract4J employs the @Pre, @Post, and @Invar annotations that indicate, respectively, a precondition, postcondition, and invariant test. The Contract4J tool also employs the use of @old annotation, which represents the evaluation of old expressions in pre-state (before method call). It is applied to state variables (attributes) that are used in post-state (after method call) within the postcondition test. The annotation-based approach (annotations) is interpreted and converted into AspectJ aspects (responsible for instrumenting and verifying the contracts during runtime).

Besides the annotation-based approach, the author of the Contract4J provides a second experimental syntax that uses a JavaBeans-style naming convention (method-based approach), which he called "ContractBeans". According to this style, the precondition and postcondition tests for a method named add, for example, are respectively written as preAdd and postAdd. (Compare with the JavaBeans convention for defining a getResult method for the result field present in a class.) This implementation approach is also based on AspectJ and has a significant runtime overhead, because it uses runtime reflection to discover and invoke the tests (when present). In addition, the work mentions another two drawbacks when using the method-based approach: (1) if the tests are not declared with public visibility, they are not visible for clients; (2) if the tests are not written in a proper JavaBeans-like convention, the tests are ignored. This happens because there is no mechanism in the Contract4J tool to warn the user. With relation to contract support, the method-based approach does not yet support **old** expressions when compared with the annotation-based approach.

Concerning the supported kinds of assertions and their implementation in AspectJ, the work covers only pre-, postcondition (normal), and invariant (instance) when compared with ours. Moreover, in contrast to our work, there is an important issue not covered by Contract4J — the current version does not provide support for inherited contracts — contravariance (used in precondition inheritance mechanism) and covariance (used in postcondition and invariant inheritance) behavior. Nevertheless, the Contract4J tool imposes at least the same contract conditions on derived classes. According to the author, he is planning to work on inheritance of contracts and release a new version at some time.

Pipa [ZR03] is a behavioral interface specification language (BISL) tailored to AspectJ. It uses the same approach (based on annotations) of JML language to specify AspectJ classes and interfaces, and extends JML with a few new constructs in order to specify AspectJ programs. The Pipa language also supports aspect specification inheritance and crosscutting. Pipa specifies AspectJ programs with pre-, postconditions, and invariants. Moreover, Pipa also can specify aspect invariants and the "decision" whether or not to call the proceed method within the around advice (using the **proceed** extended annotation). The aim in designing Pipa based on JML is to reuse the existing JML-based tools. In order to make this possible the authors developed a tool (compiler) to automatically transform an AspectJ program with Pipa specifications into a standard Java program with JML specifications. To this end, the authors modified the AspectJ compiler (ajc) to retain the comments during the weaving process. After the weaving process, all JML-based tools can be applied to AspectJ programs. Therefore, the main goal of Pipa is to facilitate the use of JML language to verify AspectJ programs. On the other hand, our aim is to use AspectJ to specify, implement, and check JML contracts for Java programs.

In summary, unlike AspectJML, none of these works provides all the benefits of a classical DbC language, such as Eiffel or JML, combined with constructs to specify crosscutting contracts in a modular and convenient way.

6.5 Design by Contract Languages and Information Hiding

A crucial feature of modules is that they support *information hiding*. Essentially, this means that the module's declarations are sliced into a public module interface, which is visible to all clients of the module, and a private module body, which contains the module implementation and is hidden from clients. The public module interface's behavior can be specified using a formal interface specification language. In this context, the most related work is by Leavens and Müller [LM07]. They present rules for information hiding in specifications for Java-like languages. Although some of these rules are enforced during compile time in JML, during the runtime assertion checking they are not taken into account as described in Section 3.4.

We complement their work with a precise runtime checking for JML information hiding features. We call this feature as client-aware checking and it is implemented as a part of our AspectJML compiler/ajmlc. AspectJML is the first DbC language to support information hiding rules for specifications that are checked both statically and dynamically.

6.6 Design by Contract Languages and Client-Side Checking

Code Contracts [FBL10] has an interesting relationship to our work: it also supports runtime checking at call sites. However, in Code Contracts only preconditions can be checked at call sites. With AspectJML/CAC, we can also check postconditions and invariants at call sites. In addition, Code Contracts do not support information hiding features. As a consequence, Code Contracts would not be able to properly check specifications with different privacy modifiers.

The work by Findler and Felleisen [FF01, FLF01a, FLF01b] is closest in spirit to our client-aware checking (CAC for short) in AspectJML. Their work describes contract soundness for a language called Contract Java. In Contract Java a programmer can specify pre- and postconditions of methods defined in classes and interfaces. Their work is closely related to ours in the sense that their translation rules for Contract Java also inject runtime checking code at call sites; that is, they perform client-side checking for pre- and postconditions. Hence their work is a precedent for runtime checking at call site, and for our work on AspectJML/CAC.

However, Findler and Felleisen's work is primarily concerned with enforcing behavioral subtyping and presenting the novel idea of soundness of contract checking. Hence, unlike our work, their work neither considers separate specifications for different privacy levels nor enforces information hiding of specifications, as we do in AspectJML.

Chapter 7

Conclusions

This work introduced AspectJML, a seamless aspect-oriented extension to JML. Programming with design by contract with AspectJML feels like a small extension of programming with Java and JML specifications. AspectJML annotated programs are largely ordinary Java programs in which we use ordinary Java for class-like modularity, and use AspectJ pointcuts (as Java 5 features) to specify crosscutting contracts (XCS). With XCS, AspectJML supports both specification and runtime checking for crosscutting contracts in a modular way.

Using AspectJML, allows programmers to enable modular reasoning in the presence of crosscutting contracts, and to recover the main DbC benefits such as documentation. Also, AspectJML gives programmers limited control over modularity for specifications. An AspectJML programmer cannot implicitly add contracts to unrelated modules. Therefore, using AspectJML, programmers get modular reasoning benefits at any time.

Since AspectJML is AspectJ-based, the AJDT tools can help the programmer navigate and understand the crosscutting contract structure. The main difference is that this feature is an add since we do not require it to perform modular reasoning. A programmer does need to know only the AspectJ's pointcut language to do so.

In relation to the client-aware checking technique (CAC), AspectJML enables enables consistent runtime assertion checking and modular reasoning in the presence of information hiding in specifications. CAC represents a change in the way the runtime checks are injected into code, since checks are placed in client code as opposed to being only done in supplier classes. Since runtime checks are injected at the point of method calls, the client's perspective on the called method can be taken into account, which allows runtime checking to be consistent with privacy modifiers used for information hiding.

We also conducted an evaluation to assess the expressiveness of AspectJML while handling crosscutting contracts. Hence, we quantitatively evaluated three real systems using a set of metrics to gather information about the impact of the crosscutting contract modularization. The evaluation provides evidence that the use of AspectJML can significantly reduce the overall DbC lines of code related to crosscutting contracts. For each analyzed system, we showed representative scenarios where AspectJML is beneficial in the day-by-day development. Despite the promising results, we did not evaluate many other interesting scenarios. For example, we did not consider the scalability of AspectJML when a system is evolving.

Finally, the benefits of AspectJML are currently being used in practice for teaching purposes. It has been used successfully in classes (for both undergraduate and graduate level) taught by professor Marcel Oliveira, at Federal University of Natal (UFRN), Brazil, and by professor Ana Cavalcanti, at University of York, UK. Prof. Marcel Oliveria is teaching JML using AspectJML/ajmlc for the second time, whereas Prof. Ana Cavalcanti just started using AspectJ/ajmlc this semester. As a feedback, we get a lot of suggestions for improving the AspectJML/ajmlc compiler and usage (which several issues were already implemented). In addition we fixed several bugs raised by them; other fixes and improvements are on the way.

7.1 Review of the Contributions

This work makes the following contributions:

- the concept of crosscutting contract specification (XCS) [RLK+10, RCL+11, RLL11, RLK+13, RLB+14] that enables developers to specify crosscutting contracts in a modular way, keeping the benefits of classical DbC languages such as documentation and modular reasoning;
- the concept of client-aware checking (CAC) [RLL12, RLL13a], that keeps the Parna's concept of information hiding when performing runtime assertion checking and error reporting, avoiding undesired surprises to clients;
- a compiler named ajmlc [RSL+08, RLL+13c, RLB+14] that is a part of the AspectJML DbC language to support both concepts of XCS and CAC;
- the compatibility with AspectJ/AJDT to support the crosscutting contract structure view, allowing better navigation between contracts and their application points in code;
- an empirical study to evaluate our proposal;
- empirical evidence of the expressiveness of the concepts/features available in AspectJML;
- AspectJML/ajmlc is being used in practice to teach JML language and formal methods at UFRN and University of York-UK.

7.2 Future Work

In particular, we intend to complement this work with the following future work:

- AspectJML covers only a subset of the JML language regarding crosscutting contract specification. It supports only pre- and postconditions in a crosscutting contract specification. Hence, we intend to support other JML features like assignable clauses.

- Currently, AspectJML can only be used to specify Java types. But as the name already suggests, we intend to allow programmers to also specify AspectJ programs. Hence, the crosscutting contract specification feature can be used for both Java and AspectJ programs.

- Enhancing AspectJML to fix the problem of modular reasoning in aspect-oriented programming. An early effort and discussions is provided with XPIDRs [RLL+13b].

- Our evaluation was only concerned to modularize the crosscutting contracts with AspectJML and compare the XCS version with a non-XCS version through a set of metrics. We also intend to consider other kinds of studies. For instance, studies considering maintenance tasks to analyze the gains of a system (that already uses AspectJML) while it evolves.

- Another future work we intend to perform is to use the SafeRefactor tool [SGM13] to improve the safeness of the AspectJML/XCS features to modularize crosscutting concerns. With SafeRefactor, for example, we can ensure that the refactored systems (with AspectJML) described in Chapter 5 are equivalent to the ones using JML. Its important to note that AspectJ already passed to a very similar assessment over the last 15 years.

- One important open issue is related to formal semantics. We intend to give a formal semantics to AspectJML regarding JML features and crosscutting features(XCS).

- We intend to investigate the use of emergent interfaces approach [RPTB10] to enhance and reduce the modular reasoning steps necessary when using AspectJML.

References

[AB01] Vander Alves and Paulo Borba. Distributed adapters pattern: A design pattern for object-oriented distributed applications. In *Proceedings of the 1st Latin American Conference on Pattern Languages of Programming*, SugarLoafPLoP '01, 2001.

[ABKS13] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[ACH$^+$05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[AJD14] AJDT AspectJ Development Tools., 2014. http://www.eclipse.org/ajdt/.

[ARG$^+$11] Rodrigo Andrade, Marcio Ribeiro, Vaidas Gasiunas, Lucas Satabin, Henrique Rebelo, and Paulo Borba. Assessing idioms for implementing features with flexible binding times. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 231–240, Washington, DC, USA, 2011. IEEE Computer Society.

[BCC$^+$05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[BCR94] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In *Encyc. of Software Engineering*, pages 528–532. John Wiley and Sons, New York, 1994.

[BDL05] Lionel C. Briand, W. J. Dzidek, and Yvan Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.

[BEM05] Stephanie Balzer, Patrick Th. Eugster, and Bertrand Meyer. Can Aspects Implement Contracts. In *In: Proceedings of RISE 2005 (Rapid Implementation of Engineering Techniques*, pages 13–15, September 2005.

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31:82–87, September 2005.

[BLS03] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw. Pract. Exper.*, 33:637–672, June 2003.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*. Springer-Verlag, 2005.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[Bon] Jonas Boner. Aspectwerks. http://aspectwerkz.codehaus.org/.

[BRD13] Mehdi Bagherzadeh, Hridesh Rajan, and Ali Darvish. On exceptions, events and observer chains. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 185–196, New York, NY, USA, 2013. ACM.

[BRLM11] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 141–152, New York, NY, USA, March 2011. ACM.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Cha07] Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 23–33, Washington, DC, USA, 2007. IEEE Computer Society.

[Che03] Yoonsik Cheon. *A runtime assertion checker for the Java Modeling Language*. Technical report 03-09, Iowa State University, Department of Computer Science, Ames, IA, April 2003. The author's Ph.D. dissertation.

[CJ07] Patrice Chalin and Perry R. James. Non-null references by default in java: alleviating the nullity annotation burden. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 227–247, Berlin, Heidelberg, 2007. Springer-Verlag.

[CJR08] P. Chalin, P.R. James, and F. Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *Software, IET*, 2(6):515–531, 2008.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

[CL94] Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, October 2000.

[CLSE05] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, May 2005.

[CMLC06] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, May 2006.

[CNBR$^+$13] Alberto Costa Neto, Rodrigo Bonifácio, Márcio Ribeiro, Carlos Eduardo Pontual, Paulo Borba, and Fernando Castor. A Design Rule Language for Aspect-oriented Programming. *J. Syst. Softw.*, 86(9):2333–2356, September 2013.

[CR05a] Patrice Chalin and Frédéric Rioux. Non-null references by default in the java modeling language. In *Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.

[CR05b] Patrice Chalin and Frédéric Rioux. Non-null references by default in the java modeling language. *SIGSOFT Softw. Eng. Notes*, 31(2), September 2005.

[CR06] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31:25–37, May 2006.

[CRG$^+$08] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt Staa, and Carlos Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 207–234, Berlin, Heidelberg, 2008. Springer-Verlag.

[Dio04] Filippo Diotalevi. Contract enforcement with AOP: Apply Design by Contract to Java software development with AspectJ. July 2004. Avaliable at http://www.ibm.com/developerworks/library/j-ceaop.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is ISU CS TR #95-20c, http://tinyurl.com/s2krg.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[EZS$^+$08] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, July 2008.

[F$^+$06] Yishai A. Feldman et al. Jose: Aspects for Design by Contract80-89. *IEEE SEFM*, 0:80–89, 2006.

[FBL10] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM.

[FCF$^+$06] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranh ao, Alessandro Garcia, and Cecília Mary F. Rubira. Exceptions and aspects: the devil is in the details. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–162, New York, NY, USA, 2006. ACM.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.

[FF01] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM.

[FLF01a] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 229–236, New York, NY, USA, 2001. ACM.

[FLF01b] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Object-oriented programming languages need well-founded contracts. Technical report, Department of Computer Science, Rice University, 2001.

[FLL^{+}02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[GBF^{+}07] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of the 21st European conference on Object-Oriented Programming*, LNCS, pages 176–200. Springer-Verlag, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GSF^{+}05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 3–14, New York, NY, USA, 2005. ACM.

[GSS^{+}06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Softw.*, 23(1):51–60, January 2006.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.

[Hoa69] Charles Antony R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa72] Charles Antony R. Hoare. Proof of Correctness of Data Representations. *Acta Inf.*, 1:271–281, 1972.

[HU03] Stefan Hanenberg and Rainer Unland. AspectJ idioms for aspect-oriented software construction. In *EuroPlop'03*, 2003.

[IKIK05] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue, and Toshihiro Kamiya. Aspect-oriented modularization of assertion crosscutting objects. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, pages 744–751, Washington, DC, USA, 2005. IEEE Computer Society.

[IKKI04] Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Assertion with Aspect. In *International Workshop on Software Engineering Properties for Aspect Technologies (SPLAT2004)*, March 2004.

[ITB11] Milton Inostroza, Éric Tanter, and Eric Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 508–511, New York, NY, USA, 2011. ACM.

[Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting tarted with AspectJ. *Commun. ACM*, 44:59–65, October 2001.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Softw.*, 13(1):8–11, January 1996.

[Kic03] Gregor Kiczales. TheServerSide.COM: Interview with Gregor Kiczales, topic: Aspect-oriented programming (AOP)., July 2003. http://www.theserverside.com/tt/talks/videos/GregorKiczalesText/interview.tss.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[KM05] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.

[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 *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

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

[Lad09] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[LASB01] Tiago Lima, Vander Alves, Sérgio Soares, and Paulo Borba. Pdc: Persistent data collections pattern. In *Proceedings of the 1st Latin American Conference on Pattern Languages of Programming*, SugarLoafPLoP '01, 2001.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 2006.

[LCC$^+$05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[Lea06] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[LKR05] Patrick Lam, Viktor Kuncak, and Martin Rinard. Crosscutting Techniques in Program Specification and Analysis. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 169–180, New York, NY, USA, 2005. ACM.

[LL00] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 418–427, New York, NY, USA, 2000. ACM.

[LLH02] Cristina V. Lopes, Martin Lippert, and Eric A. Hilsdale. Design By Contract with Aspect-Oriented Programming. In *U.S. Patent No. 06,442,750*, issued August 27, 2002.

[LM07] Gary T. Leavens and Peter Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, May 2007.

[LN13] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report CS-TR-13-03a, Computer Science, University of Central Florida, Orlando, FL, 32816, July 2013.

[LPC$^+$08] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from http://www.jmlspecs.org, May 2008.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java Programs via Guarded Commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, 1999. Springer-Verlag.

[LTBJ06] Yves Le Traon, Benoit Baudry, and Jean-Marc Jezequel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, August 2006.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16:1811–1841, November 1994.

[Mey92a] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, 1992.

[Mey92b] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, PTR, 2nd edition, 2000.

[MMM02] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[MMU04] C. Marche, Paulin C. Mohring, and X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[MMvD05] Marius Marin, Leon Moonen, and Arie van Deursen. A Classification of Crosscutting Concerns. In *ICSM ’05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA, 2005. IEEE Computer Society.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par11] David Lorge Parnas. Precise Documentation: The Key to Better Software. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer Berlin Heidelberg, 2011.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag Inc., New York, NY, USA, 1994.

[RCL$^+$11] Henrique Rebêlo, Roberta Coelho, Ricardo Lima, Gary T. Leavens, Alexandre Mota, and Fernando Castor. On the interplay of exception handling and design by contract: An aspect-oriented recovery approach. Technical Report CS-TR-11-02, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, April 2011.

[Reb08] Henrique Emanuel Mostaert Rebêlo. Implementing jml contracts with aspectj. Master's thesis, Departament of Computing and Systems, State University of Pernambuco, May 2008.

[RLB$^+$14] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel Zimmerman, Márcio Cornélio, and Thomas Thüm. AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the Thirteenth International Conference on Modularity*, Modularity '14, New York, NY, USA, 2014. ACM.

[RLK$^+$10] Henrique Rebêlo, Ricardo Lima, Uirá Kulesza, Roberta Coelho, Alexandre Mota, Ma´rio Ribeiro, and José Elias Araujo. The contract enforcement aspect pattern. In *Proc. of the 2010 SugarLoafPLoP*, pages 99–114, 2010.

[RLK$^+$13] Henrique Rebêlo, Ricardo Lima, Uirá Kulesza, Márcio Ribeiro, Yuanfang Cai, Roberta Coelho, Cáudio Sant'Anna, and Alexandre Mota. Quantifying the Effects of Aspectual Decompositions on Design by Contract Modularization: A Maintenance Study. *International Journal of Software Engineering and Knowledge Engineering*, 23(07):913–941, 2013.

[RLL11] Henrique Rebêlo, Ricardo Lima, and Gary T. Leavens. Modular Contracts with Procedures, Annotations, Pointcuts and Advice. In *SBLP '11: Proceedings of the 2011 Brazilian Symposium on Programming Languages*. Brazilian Computer Society, 2011.

[RLL12] Henrique Rebêlo, Gary T. Leavens, and Ricardo Lima. Modular enforcement of supertype abstraction and information hiding with client-side checking. Technical Report CS-TR-12-03, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, January 2012.

[RLL13a] Henrique Rebêlo, Gary T. Leavens, and Ricardo Massa Lima. Client-aware checking and information hiding in interface specifications with jml/ajmlc. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, &#38; Applications: Software for Humanity*, SPLASH '13, pages 11–12, New York, NY, USA, 2013. ACM.

[RLL$^+$13b] Henrique Rebelo, Gary T. Leavens, Ricardo Massa Ferreira Lima, Paulo Borba, and Márcio Ribeiro. Modular aspect-oriented design rule enforcement with XPIDRs. In *Proceedings of the 12th workshop on Foundations of aspect-oriented languages*, FOAL '13, pages 13–18, New York, NY, USA, 2013. ACM.

[RLL$^+$13c] Henrique Rebêlo, Ricardo Lima, Gary T. Leavens, Márcio Cornélio, Alexandre Mota, and César Oliveira. Optimizing generated aspect-oriented assertion checking code for JML using program transformations: An empirical study. *Science of Computer Programming*, 78(8):1137 – 1156, 2013.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, January 1995.

[RPTB10] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent feature modularization. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '10, pages 11–18, New York, NY, USA, 2010. ACM.

[RSL$^+$08] Henrique Rebêlo, Sérgio Soares, Ricardo Lima, Leopoldo Ferreira, and Márcio Cornélio. Implementing Java modeling language contracts with AspectJ. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 228–233, New York, NY, USA, 2008. ACM.

[SGM13] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *Software Engineering, IEEE Transactions on*, 39(2):147–162, Feb 2013.

[SGR$^+$10] Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with XPIs. 20(2):5:1–5:42, September 2010.

[SL04] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 196–197, New York, NY, USA, 2004. ACM.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 174–190, New York, NY, USA, 2002. ACM.

[Som01] Ian Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[SPAK10] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, 20(1):1:1–1:43, July 2010.

[Spi89] M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[Ste06] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *OOPSLA 2006: Proceedings of the 21st International Conference on Object-oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 481–497, October 2006.

[Tea08] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 8.1, May 2008. http://pauillac.inria.fr/coq/doc/main.html.

[Tea14a] HealthCard Specification Team. HealthCard: JavaCard + JML specs, January 2014. http://wwwhome.ewi.utwente.nl/ mostowskiwi/software.html.

[Tea14b] Java Card Specification Team. The Java Card API Specification in JML, Version 2.2.1, January 2014. http://healthcard.sourceforge.net/.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[TSK$^+$12] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying design by contract to feature-oriented programming. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 255–269, Berlin, Heidelberg, 2012. Springer-Verlag.

[VCFS10] Marco Tulio Valente, Cesar Couto, Jaqueline Faria, and Sérgio Soares. On the benefits of quantification in AspectJ systems. *Journal of the Brazilian Computer Society*, 16(2):133–146, 2010.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In *TACAS 2001: Proceedings of the 7thInternationalConference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, London, UK, 2001. Springer-Verlag.

[Ver03] Joe Verzulli. Getting started with JML: Improve your Java programs with JML annotation., march 2003. http://www-128.ibm.com/developerworks/java/library/j-jml.html.

[Wam06] Dean Wampler. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *ACP4IS Workshop at AOSD 2006*, pages 27–30, March 2006.

[WK99] Jos Warmer and Anneke Kleppe. *The object constraint language: precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[YW06] Kiyoshi Yamada and Takuo Watanabe. An aspect-oriented approach to modular behavioral specification. *Electronic Notes in Theoretical Computer Science*, 163(1):45 – 56, 2006.

[ZR03] Jianjun Zhao and Martin Rinard. Pipa: a behavioral interface specification language for AspectJ. In *Proceedings of the 6th international conference on Fundamental approaches to software engineering*, FASE'03, pages 150–165, Berlin, Heidelberg, 2003. Springer-Verlag.

Appendix A

AspectJML Grammar Summary

The following is a summary of the grammar for AspectJML language that ajmlc compiler handles.

A.1 JML-based Reserved Words

```
\requires        \pre              \same
\ensures         \post             \result
\signals         \exsures          \old
\assignable      \modifies         pure
\not_specfied    also              invariant
behavior         normal_behavior   exceptional_behavior
spec_public      spec_protected    instance
static           \everything       \nothing
\only_assigned
```

A.2 @AspectJ-based Reserved Words

```
@Pointcut  @InterfaceXCS  thisJoinPoint
call       execution      within
args       @annotation
```

A.3 Method Specfication

$Method-Specification$ ::= $Specification$
$Specification$ ::= $Spec-Case$ [**also** $Spec-Case$] **. . .**
$Spec-Case$ ::= $Lightweight-Spec-Case$ | $Heavyweight-Spec-Case$
$Lightweight-Spec-Case$ ::= $Generic-Spec-Clause$
$Heavyweight-Spec-Case$ ::= $Behavior-Spec-Case$
    | $Normal-Behavior-Spec-Case$
    | $Exceptional-Behavior-Spec-Case$
$Behavior-Spec-Case$ ::= [privacy] $Behavior-Keyword$

```
                    Generic − Spec − Clause
Behavior − Keyword ::= behavior | behaviour
Normal − Behavior − Spec − Case ::= [ privacy ] Normal − Behavior − Keyword
                                    Generic − Spec − Clause
Normal − Behavior − Keyword ::= normal_behavior | normal_behaviour
Exceptional − Behavior − Spec − Case ::= [ privacy ] Exceptional − Behavior − Keyword
                                    Generic − Spec − Clause
Exceptional − Behavior − Keyword ::= exceptional_behavior | exceptional_behaviour
Generic − Spec − Case ::= Requires − Clause [Requires − Clause] ...
      | Ensures − Clause [Ensures − Clause] ...
      | Signals − Clause [Signals − Clause] ...
      | Signals − Only − Clause [Signals − Only − Clause] ...
```

A.4 Type Specfication

```
Invariant ::= [Privacy] [static] Invariant − Keyword Predicate ;
Invariant − Keyword ::= invariant
JML − Field − Decl ::= Model − Field − Decl | Represents − Clause | Field − Decl
Model − Field − Decl ::= [Privacy] [Modifiers] Model − Keyword FieldType Field − Ident ;
Model − Keyword ::= model
Represents − Clause ::= [Privacy] [static] Represents − Keyword Field − Ident = Exp ;
Represents − Keyword ::= represents
Field − Decl ::= [Privacy] [Modifiers] [JML − Modifier] FieldType Field − Ident [= Exp] ;
Exp ::= FieldAccessExp | MethodCallExpr | ConstantExp
...
```

A.5 Predicates and Specification Expressions

```
predicate ::= spec − expression
spec − expression ::= expression

I ∈ identifier
E ∈ expression
E ::= E1 <==> E2
    | E1 <=!=> E2
    | E ==> E
    | E <== E2
    | E1 || E2
    | E1 && E2
    | E2 == E2
    | E1 != E2
    | E.I
    | !E
    | ...
    | jml−expressions

jml − expressions ::= result − expression
    | old−expression

result − expression ::= \result
old − expression ::= \old ( spec − expression)
    | \pre ( spec − expression )
```

A.6 Pointcut Expressions

```
Method − Specification ::= ...
@Pointcut("Pointcut − Expression")
[Privacy] void Pointcut − Ident ([Args]) {}
Pointcut − Expression ::= Call − Pointcut
     | Execution − Pointcut
     | Args − Pointcut
     | Within − Pointcut
     | Annotation − Pointcut
     | Pointcut − Expression && Pointcut − Expression
     | Pointcut − Expression || Pointcut − Expression
     | !Pointcut − Expression
Call − Pointcut ::= call(Method − Pattern)
Execution − Pointcut ::= execution(Method − Pattern)
Args − Pointcut ::= args(Type | Id [, Type | Id] ...)
Within − Pointcut ::= within(Type − Pattern)
Annotation − Pointcut ::= @annotation(AnnotationType)

Method − Pattern =
   [Modifiers − Pattern] Type − Pattern
           [Type − Pattern . ] Id − Pattern (Type − Pattern | ".." , ... )
           [ throws Throws − Pattern ]
Constructor − Pattern =
   [Modifiers − Pattern ]
           [Type − Pattern . ] new (Type − Pattern | ".." , ...)
           [ throws Throws − Pattern ]
Throws − Pattern =
   [ ! ] Type − Pattern , ...
Type − Pattern =
      Id − Pattern [ + ] [ [] ... ]
      | ! Type − Pattern
      | Type − Pattern && Type − Pattern
      | Type − Pattern || Type − Pattern
      | ( Type − Pattern )
Id − Pattern =
  Sequence of characters, possibly with special * and .. wildcards
Modifiers − Pattern =
  [ ! ] JavaModifier ...
...
```