FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Informática

An Aspect-Oriented Infrastructure for Design by Contract in Java

Sérgio Miguel Fortunato Agostinho

"Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do grau de Mestre em Engenharia Informática"

Orientador científico: Dr. Pedro Guerreiro

Co-Orientadora científica: Drª. Ana Moreira

Monte de Caparica

2008

Agradecimentos

Antes de mais, quero agradecer à minha família, amigos e colegas que me continuam a aturar.

Em seguida, quero agradecer às pessoas que directamente tornaram esta tese possível. Ao Ricardo Raminhos e ao Ricardo Ferreira por toda a coordenação e apoio que me deram desde que nos conhecemos, sem pedir nada em troca. Aos Ricardos e ao André Marques, pelos comentários aos esboços da dissertação. Aos meus orientadores Ana Moreira e Pedro Guerreiro, por me terem ajudado a encontrar uma tese com que me identifiquei; tal como seu rigor e profissionalismo. Aos professores Miguel Pessoa Monteiro e Artur Miguel Dias, pelas diversas conversas e trocas de email interessantes que tivemos. Ao Hugo Taborda, por ter "desbravado" caminho para mim neste trabalho.

Quero também agradecer a amizade e profissionalismo de todas as pessoas com as quais estive a trabalhar durante este período. No CITI: André Marques, Isabel Brito e João Araújo. No projecto ISIS/CA^3: José Fonseca, André Mora, Inês Guerra, Fernando Moutinho, Catarina Gomes, Sofia Guerra e Carlos Figueira.

Não me esqueci dos meus colegas (ainda) não licenciados pela companhia e amizade dos últimos anos: João Morgado e Paulo Mestre. Vítor Pires fez a gentileza de me continuar a fornecer uma conta de *Subversion*.

A nível mais pessoal, quero agradecer ao "grupo", por estes dois últimos anos de partilha: Maria João, Eva, Francisco, Pedro, Ricardo, Sandra, Sara, Tânia, e Vanessa. Quero também agradecer à minha mais recente "professora", Cláudia Inocêncio, por me abrir a mente e o corpo a novas experiências.

Em geral, quero agradecer às pessoas por detrás da Wikipedia e do Google Scholar, por fornecerem estas excelentes ferramentas de pesquisa. As comunidades *online* presentes nos fóruns e *mailing lists*, em particular a "aosd-discuss" e "aspectj-users", também ajudaram bastante.

Finalmente, quero também agradecer à Fundação da Faculdade para a Ciência e Tecnologia por financiar o projecto SOFTAS (POSI/EIA/60189/2004), no qual trabalhei sob bolsa durante parte da escrita desta dissertação.

'[Holding the Holy Hand Grenade of Antioch]

King Arthur: How does it... um... how does it work?

Sir Lancelot: I know not, my liege.

King Arthur: Consult the Book of Armaments.

Brother Maynard: Armaments, chapter two, verses nine through twenty-one.

Cleric: [reading] And Saint Attila raised the hand grenade up on high, saying, "O Lord, bless this thy hand grenade, that with it thou mayst blow thine enemies to tiny bits, in thy mercy." And the Lord did grin. And the people did feast upon the lambs and sloths, and carp and anchovies, and orangutans and breakfast cereals, and fruit-bats and large chu...

Brother Maynard: Skip a bit, Brother...

Cleric: And the Lord spake, saying, "First shalt thou take out the Holy Pin. Then shalt thou count to three, no more, no less. Three shall be the number thou shalt count, and the number of the counting shall be three. Four shalt thou not count, neither count thou two, excepting that thou then proceed to three. Five is right out. Once the number three, being the third number, be reached, then lobbest thou thy Holy Hand Grenade of Antioch towards thy foe, who, being naughty in my sight, shall snuff it.

Brother Maynard: Amen.

All: Amen.

King Arthur: Right. One... two... five.

Galahad: Three, sir.

King Arthur: Three. '

– Monty Python and the Holy Grail (1975)

Sumário

O "desenho por contrato" foi introduzido na linguagem de programação Eiffel, com o objectivo de melhorar a fiabilidade associada ao desenvolvimento de software orientado a objectos, nomeadamente a interacção entre módulos. Por outro lado, o desenvolvimento de software orientado a aspectos tem por objectivo principal criar meios para a identificação, modularização e composição de assuntos transversais (do inglês *crosscutting concerns*) que as abordagens clássicas de desenvolvimento de software não conseguem modularizar.

O trabalho desenvolvido para esta dissertação pretende determinar a viabilidade de suportar a verificação de contratos numa linguagem de programação sem suporte nativo para a mesma, utilizando aspectos reutilizáveis. Assim, esta dissertação propõe a implementação de uma infraestrutura de verificação de desenho por contrato em Java. Esta infraestrutura utiliza o AspectJ, uma extensão orientada a aspectos para Java. No âmbito deste trabalho, desenvolvemos um protótipo de uma biblioteca AspectJ/Java, que foi validada com um conjunto de casos de estudo. Fizemos ainda um levantamento e uma análise comparativa de soluções alternativas de desenho por contrato existentes para Java. Além disso, também apresentamos soluções alternativas ao AspectJ. Finalmente, discutimos ainda as questões levantadas ao transpor desenho por contrato do desenho orientado por objectos para o desenho orientado por aspectos.

Abstract

Design by Contract was first introduced in the Eiffel programming language to address the reliability concern in Object-Oriented Software Development, namely on module interaction. On the other hand, Aspect-Oriented Software Development aims at providing a means to identify, modularise and compose cross-cutting concerns, which classical approaches for software development can not modularise.

The work developed for this dissertation intents to determine the feasibility of implementing Design by Contract on a language with no native support for it, using reusable aspects. Thus, this dissertation proposes the use of Aspect-Oriented Programming to implement an infrastructure for Design by Contract verification in Java. This infrastructure is based on AspectJ, an aspect-oriented extension to Java. In the course of this work, we developed a prototype AspectJ/Java library which was validated through a set of case studies. We perform a comparative analysis on existing alternative Design by Contract solutions for Java. Also, we present alternative solutions to AspectJ. Finally, the issues of transposing Design by Contract from Object-Oriented Design to Aspect-Oriented Design are discussed.

Index

Chapter 1
Introduction....1
1.1 Context and Motivation....1
1.2 Thesis Goals....1
1.3 Document Structure....2
1.4 Notations and Styles....2
Chapter 2
Design by Contract....3
2.1 Introducing Design by Contract....3
2.1.1 Basic Concepts....3
2.1.2 Implementing Liskov's Substitution Principle....4
2.1.3 Contract Levels....5
2.1.4 Development and Production....5
2.1.5 Design by Contract vs Defensive Programming....5
2.1.6 Design by Contract vs Unit Testing....6
2.1.7 Specification languages....6
2.2 Programming languages with native support....6
2.2.1 Eiffel....7
2.2.2 Blue....8
2.2.3 Chrome....9
2.2.4 D....9
2.2.5 Lisaac....11
2.2.6 Nemerle....11
2.2.7 Sather....12
2.2.8 Other programming languages....13
2.3 Language extensions for Java....13
2.3.1 Oak....14
2.3.2 ContractJava....14
2.3.3 Handshake....15
2.3.4 KJC....16
2.4 Traditional ad-hoc solutions for Java....16
2.4.1 C4J....16
2.4.2 iContract2....18
2.4.3 Jass....19
2.4.4 Jcontract....20
2.4.5 jContractor....21
2.4.6 JML Tools....23
2.4.7 Modern Jass....24
2.4.8 STClass....25
2.5 Aspect-Oriented ad-hoc solutions for Java....26
2.5.1 Barter....26
2.5.2 ConFA....27
2.5.3 Contract4J....28
2.5.4 OVal....29
2.5.5 SpringContracts....30
2.6 Static verification: a step further....31
2.6.1 JML static verification tools....31
2.6.2 Spec#....32
2.7 Summary....34
Chapter 3
Aspect-Oriented Programming....35
3.1 Introducing Aspect-Oriented Programming....35
3.2 AspectJ....35
3.2.1 Join Points and Pointcuts....35
3.2.2 Advices: dynamic cross-cutting....36
3.2.3 Static cross-cutting....36
3.2.4 The Aspect module....37
3.2.5 Annotation based development style....38

3.2.6 Weaving....38
3.2.7 Tool support....39
3.3 Other Solutions....40
3.3.1 JBoss AOP....40
3.3.2 Spring AOP....42
3.3.3 AspectWerkz....42
3.3.4 FuseJ....42
3.3.5 Hyper/J....43
3.3.6 CaesarJ....44
3.4 Native aspect-oriented support....46
3.5 Summary....46
Chapter 4
Library Design and Implementation....49
4.1 Requirements....49
4.2 Design decisions....49
4.2.1 Executable code....49
4.2.2 Aspect-Oriented Programming....50
4.2.3 Programming language....50
4.2.4 "result" and "old" constructs....51
4.2.5 Boolean methods....51
4.2.6 Exceptions....51
4.3 Features....52
4.3.1 Preconditions, Postconditions and Invariants....52
4.3.2 The "Old" mechanism....55
4.3.3 Inheritance....55
4.3.4 Contract rules....57
4.3.5 Other features....58
4.4 Prototype Limitations....59
4.5 Summary....60
Chapter 5
Case Studies....63
5.1 FPL Tower Interface....63
5.1.1 Brief introduction to the system....63
5.1.2 Objectives and scope of the study....63
5.1.3 Analysis....63
5.2 Design by Contract, by Example....67
5.2.1 Objectives and scope of the study....67
5.2.2 Analysis....67
5.3 Syntactic contracts vs Behavioural contracts....67
5.4 Summary....67
Chapter 6
Evaluation....69
6.1 Criteria....69
6.2 Comparison with existent solutions....75
6.2.1 Native support....75
6.2.2 Language extensions for Java....78
6.2.3 "Traditional" ad-hoc solutions for Java....80
6.2.4 Aspect-Oriented ad-hoc solutions for Java....82
6.2.5 Overall comparison with DbC4J....84
6.2.6 Third-party evaluation....84
6.3 Performance....85
6.3.1 Experiment subject....85
6.3.2 Measurement method....85
6.3.3 Quantitative results....85
6.3.4 Qualitative analysis....86
6.4 Evaluation of the proposed requirements....86
6.5 Prototype technical details....87
6.6 Further ideas....88
6.6.1 Library features....88
6.6.2 Auxiliary tools....90
Chapter 7
Contracts for Aspect-Oriented Design....91
7.1 Aspect-aware Contracts....91

7.1.1 Classic View....91
7.1.2 Issues....92
7.1.3 A Solution....92
7.2 Contracts for Aspects....93
7.2.1 Dynamic cross-cutting....93
7.2.2 An Example....93
7.2.3 Behavioural Subtyping....94
7.2.4 Mixins and multiple inheritance....94
7.3 Language Specific Considerations....95
7.3.1 AspectJ....95
7.3.2 CaesarJ....95
7.3.3 FuseJ....95
7.4 Related Work and Discussion....96
7.4.1 Open Modules / Cross-cutting Interfaces....96
7.4.2 Pipa....96
7.4.3 ConFA....97
7.4.4 COW....97
7.5 Prototype Integration....97
7.6 Summary....98
Chapter 8
Conclusions....99

Index of Figures

Figure 2.1: Precondition evaluation, original (left) and Findler's proposal (right)......4
Figure 2.2: Postcondition evaluation, original (left) and Ecma's revision (right)......4
Figure 4.1: Exception class diagram......52
Figure 4.2: DesignByContract aspect diagram......54
Figure 4.3: ContractMemory class......55
Figure 4.4: ContractMemory aspect diagram......56
Figure 4.5: Total preconditions and postconditions......56
Figure 4.6: Rules class diagram......58
Figure 4.7: ContractRules aspect diagram......58
Figure 6.1: Comparison of languages with native support......77
Figure 6.2: Comparison of language extensions for Java......79
Figure 6.3: Comparison of "traditional" ad-hoc solutions for Java......81
Figure 6.4: Comparison of aspect-oriented ad-hoc solutions......83
Figure 6.5: A possible improvement over the ContractLogic class......90

Index of Tables

Table 2.1: Obligations and benefits in a contract....3
Table 2.2: iContract quantifier syntax....19
Table 2.3: jContractor's first-order logic quantifiers and operators....22
Table 2.4: Java Modelling Language first-order logic constructs....24
Table 2.5: Modern Jass basic annotations' attributes....25
Table 2.6: First-order logic support in STClass....26
Table 2.7: First-order logic support in SpringContracts....30
Table 3.1: Static join points in AspectJ....36
Table 3.2: Dynamic join points in AspectJ....36
Table 3.3: JBoss invocation classes....41
Table 3.4: Hyper/J composition functions....44
Table 6.1: Plösch evaluation summary on DbC4J....85
Table 6.2: Performance experiment quantitative results....86
Table 6.3: Program size in bytes....87
Table 6.4: Program size in lines of code....88
Table 7.1: Classical responsibilities in Design by Contract....91
Table 7.2: Responsibilities in aspect-aware Design by Contract....92

Index of Listings

Listing 2.1: Eiffel basic assertion syntax (left) and loop syntax (right)........7
Listing 2.2: Blue basic assertion syntax........8
Listing 2.3: Chrome basic assertion syntax........9
Listing 2.4: D basic assertion syntax........10
Listing 2.5: Lisaac basic assertion syntax (left) and assertion types (right)........11
Listing 2.6: Nemerle basic assertion syntax........12
Listing 2.7: Sather basic assertion syntax........13
Listing 2.8: Oak assertion syntax........14
Listing 2.9: ContractJava basic assertion syntax........15
Listing 2.10: Handshake basic assertion syntax........15
Listing 2.11: KJC basic assertion syntax........16
Listing 2.12: C4J basic assertion syntax........17
Listing 2.13: C4J postcondition extended features........18
Listing 2.14: iContract basic assertion syntax (left) and optional exception clause (right)........18
Listing 2.15: Jass basic assertion syntax (left) and frame rules example (right)........19
Listing 2.16: Jass trace assertions example........20
Listing 2.17: Jcontract basic assertion syntax........21
Listing 2.18: jContractor basic assertion syntax (left) and exception handler (right)........22
Listing 2.19: Java Modelling Language basic assertion syntax........23
Listing 2.20: Modern Jass basic assertion syntax........25
Listing 2.21: STClass basic assertion syntax........26
Listing 2.22: Barter basic assertion syntax........27
Listing 2.23: ConFA basic assertion syntax........28
Listing 2.24: Contract4J basic assertion syntax........28
Listing 2.25: OVal basic assertion syntax........29
Listing 2.26: SpringContracts basic assertion syntax using annotations........30
Listing 2.27: SpringContracts basic assertion syntax using external XML files........31
Listing 2.28: Spec# basic assertion syntax........33
Listing 3.1: AspectJ pointcut examples........37
Listing 3.2: AspectJ advice examples........38
Listing 3.3: AspectJ aspect example........39
Listing 3.4: AspectJ code in annotation-based style........40
Listing 3.5: JBoss aspect convention........41
Listing 3.6: JBoss XML example........41
Listing 3.7: JBoss introduction example........41
Listing 3.8: FuseJ service specification example........43
Listing 3.9: General structure of a FuseJ configuration entity........43
Listing 3.10: CaesarJ virtual class example (left) and mixin example (right)........45
Listing 3.11: CaesarJ local deployment example........45
Listing 3.12: CaesarJ thread-based deployment example (left) and static deployment example (right)........46
Listing 4.1: DbC4J alternative assertion implementation example........51
Listing 4.2: Precondition and postcondition example under DbC4J........53
Listing 4.3: Class invariant example under DbC4J........53
Listing 4.4: Ambiguous contracts under DbC4J........54
Listing 4.5: DbC4J configuration file example........57
Listing 4.6: Example on using the runtime API to change the contract rules........58
Listing 4.7: DbC4J's expose feature example........59
Listing 5.1: Arrivals.setEtb (defensive implementation)........64
Listing 5.2: Arrivals.setEtb (contracted overloaded implementation)........64
Listing 5.3: Arrivals.setEtb (typed implementation)........64
Listing 5.4: Time class........65
Listing 5.5: Arrivals class invariant........66
Listing 5.6: Arrivals.setAtaAndRwy (exposed)........66
Listing 5.7: An example written in Eiffel (left) and Java/DbC4J (right)........68
Listing 6.1: A possible solution for concurrent programming with contracts........88
Listing 6.2: Current implementation (left) and possible solution for the frame problem (right)........89
Listing 7.1: Web store example class (left) and aspect (right) with contracts........93

Chapter 1

Introduction

In this chapter the thesis is introduced, namely in terms of context and goals. Also, the dissertation structure and conventions are presented.

1.1 Context and Motivation

Design by Contract deals with reliability in software development. It was first introduced in the Eiffel programming language in the 80's. It is currently absent from most modern programming languages, namely Java and C#. The original Java language specification [FirstPerson94] (called Oak at the time) did have support for Design by Contract, but due to time restrictions it was not part of the final specification. Nevertheless, there is an interest on the Java developer community for Design by Contract support; it has been a Request For Enhancements for Java on the Sun Developer Network since 2001, and it is currently (December 2006) the top request. Sun Microsystems is ambiguous on whether it will or will not include such a mechanism in the Java language, but if it does, it will not be in a near future[1].

Aspect-Oriented Software Development (AOSD) aims at providing means for the systematic identification, modularization and composition of *crosscutting concerns*. Traditional software development methods are not able to modularize such concerns and their implementation ends up scattered along several other modules. This results in tangling implementations that are difficult to maintain, reuse and evolve. For the last few years, the AOSD concepts travelled up the software development life cycle, currently encompassing programming, design, analysis, software architecture and middleware. Today, there is a considerable offer on programming language extensions to Aspect-Oriented Programming. Some of these languages have reached a very mature state, with AspectJ being the most remarkable example.

Having these two topics in mind, there is a research space in using Aspect-Oriented Programming to implement Design by Contract in Java. This dissertation attempts to contribute in this area, by providing a practical, yet correct solution. Existing solutions tend to be either excessively academic, by focusing on a single facet of the problem and ignoring further facets of the solution; or excessively practical, underestimating the existing theoretical knowledge of the problem. This work provides a bridge between these two fields. At another level, an analysis of this thesis can be viewed as a validation test to whether the claims on improved separation of concerns with Aspect-Oriented Programming do hold in practice.

1.2 Thesis Goals

Many authors have proposed techniques to use Aspect-Oriented Programming to implement specific contracts [AspectJTeam03] [Laddad03] [Diotalevi04]. However, the notion of contracts as cross-cutting concerns has been strongly criticised [Balzer06], as it separates contracts for the base modules, when they should be part of the modules' interfaces. Thus, our aim is to provide a generic solution of reusable aspects, that is, an *infrastructure* for contract validation, on which assertions can be written inside the actual classes, as actual executable code. We acknowledge the significant research work on Design by Contract *ad-hoc* implementations for Java (discussed later in this dissertation), but our aim is to provide a *seamless, non-invasive* solution, that is, a solution which is based on standard Java, with contracts written in Java executable code, without resorting to pre/post-processing or virtual machine

1 Sun's statement can be found in Annex B.

modification.

Therefore, the main goal of this dissertation is to implement a library to extend the Java programming language. This library implements mechanisms that support the Design by Contract approach, through the use of an Aspect-Oriented Programming language. Part of the work necessary to reach that goal is research the state of the art on Design by Contract and on Aspect-Oriented Programming solutions for implementing it. Once the library is implemented, we need to carry out some case-studies, to evaluate its applicability and to compare the results with other existing solutions.

1.3 Document Structure

This document is composed of eight chapters and two annexes. Chapter 1 provides is this introduction. Chapters 2 and 3 recall the state of the art in Design by Contract and Aspect-Oriented Programming, respectively. Chapter 4 presents the design and implementation of our library for Design by Contract in Java using Aspect Oriented Programming. Chapter 5 discusses the experience by means of two case studies. Chapter 6 performs a critical evaluation of the work developed. Chapter 7 discusses the issues using contracts for aspect-oriented design. Finally, Chapter 8 is the conclusion. Annex A contains complementary information related to one of the case studies, and Annex B presents Sun Microsystems' official statement on Design by Contract support for Java.

Additionally to this document, a CD-ROM is also provided with a snapshot of the supporting website of this thesis (http://aosd.di.fct.unl.pt/sergioag). The CD includes: this document, the user manual, the developed library binaries, source code, and documentation.

1.4 Notations and Styles

In this document several notations are used. Acronyms are mostly avoided, unless in some cases where the acronym is better well-known than the actual name (e.g. JAR). Text body is written in Times New Roman 11 pt, or 9 pt inside tables. Headings are written in the Verdana font, 21 pt for level 1, and 14 pt for levels 2 and 3. Terms in *italics* refer to new terms introduced, non English words, or simply to emphasise a term; while terms in the text body written in the Verdana 9 pt refer to language and technology related keywords, data types, functions, or library modules. Words or expressions in green are not meant to be understood literally, but interpreted, and are used in source code listings.

Pictures, diagrams and formulas are indexed as “Figures”, and source code examples are indexed as “Listings”. Figures and Listings have captions above, while Tables have captions below. In all of them, the numbering is restarted in every chapter, by prefixing the chapter number, separated by a dot (e.g. “Figure 3.11”).

References are written in brackets, with the first author surname or company name, and the final two digits of the year (e.g. “[Meyer97]”). In case of collisions, a letter is appended to the year, in subsequent references.

The English spelling used in this document is the United Kingdom one, except when referring to names that originated from other countries, such as libraries and articles.

This document was written in OpenOffice.org 2.0, XML-hacked in GNU Emacs and gedit, and reviewed in Microsoft Word 2003. The UML diagrams were built in ArgoUML, StarUML, and MVCASE tools, with some post-processing in The GIMP 2.2.

Chapter 2

Design by Contract

In this chapter, Design by Contract is discussed. First, an introduction to Design by Contract is provided. Afterwards, a survey on available solutions that support some level of Design by Contract is presented. The existent solutions are varied, such as: programming languages with native support, language extensions, and libraries. Finally, we take a step further and discuss static verification of contracts.

2.1 Introducing Design by Contract

The Design by Contract approach was introduced by Bertrand Meyer in the Eiffel programming language [Meyer91] [Meyer97] [EiffelSoftware07]. Design by Contract aims to address the *reliability*[1] concern in object-oriented software development.

2.1.1 Basic Concepts

In order to achieve software reliability, Design by Contract advocates that methods and classes can have *assertions* [Hoare69], namely:

- method *preconditions*: assertions that must be true prior to a method execution;
- method *postconditions*: assertions that must be true as a result of a method execution;
- class *invariants*: assertions that must be always held during the lifetime of the class objects.

A classic analogy to explain Design by Contract is expressed through the responsibilities expressed by a business contract. In a business, there are (at least) two interested parties: the *client* and the *supplier*[2]. In order to be able to provide a service properly, the supplier expects that the client fulfils some stated obligations: the *preconditions* (an obligation to the client, but a benefit to the supplier). On the other hand, at the end of the service, the client expects to benefit from some stated results, achieved by the service provided by the supplier: the *postconditions* (an obligation to the supplier, but a benefit to the client). Additionally, the contract can have some clauses that specify certain conditions that both parties must guarantee throughout the duration of the service: the *invariants*. The pre/postcondition dichotomy is summarized in Table 2.1.

Table 2.1: Obligations and benefits in a contract

	Client	**Supplier**
Obligation	Precondition	Postcondition
Benefit	Postcondition	Precondition

Furthermore, according to Liskov's Substitution Principle [Liskov93], if a class B is a subtype of A, then in any situation that an instance of A can be used, it can replaced by an instance of B. To achieve this in a contract, the concept of *contract inheritance* must be introduced. As such, preconditions are *contravariant* and postconditions are *covariant* [Liskov99] [Castagna95]. This means that when extending a class, it is only possible to maintain or *weaken* preconditions (with less restrictive preconditions) and maintain or *strengthen* postconditions (with more restrictive postconditions).

1 Meyer [Meyer97] defines reliability as correctness and robustness.
2 In object-oriented terminology, these are also called “client” and “server”.

Invariants must be held for all classes in the hierarchy.

Using a “real-life” example, imagine that a person is using a mail service to send a package. In order to do so, a contract must exist between the person (*client*) and the mail service (*supplier*). If the person wants to send a package, it must have a correct stamp, a correct receiver address and is accepted only through 8h to 17h (*precondition*); on the other hand the person is expecting that the package will be delivered in two days (*postcondition*). Both accept that the service will only be provided in working days, not weekends (*invariant*).

The previous example can be expanded to explain contract inheritance. In this example, inheritance is represented through work delegation. In a mail service, the work is divided through several departments. There are two departments, one that accepts packages and another that delivers them. While delegating the package from the first department to the second, it is not a problem to *weaken* the precondition (e.g. the package can be sent until 20h), but it is not acceptable to strengthen it (e.g. demanding that the address is written in black ink). On the other hand, it is acceptable to *strengthen* the postcondition (e.g. delivering the package in the same day), but not weaken it (e.g. delivering the package in five days). Further invariants can apply (e.g. the package must have valid transport papers).

2.1.2 Implementing Liskov's Substitution Principle

In practice, the substitution principle can implement the contravariance/covariance/invariance behaviour by applying a logical disjunction to the partial preconditions, and a logical conjunction to the partial postconditions and invariants. Indeed this was the Eiffel language original implementation.

There has been some recent development in that area, namely by Findler et al. [Findler01] [Findler01a], claiming that such solution allows the formation of incoherent contracts. For example, lets consider two classes, class A with a method f() and a precondition pre_A, and class B which extends A and redefines method f() with a precondition pre_B. According to the original algorithm, the precondition for a method execution B.f() would be a logical disjunction (Figure 2.1, right side).

This however, assumes that pre_B is actually weaker than pre_A, leaving the responsibility of ensuring that to the programmer. As such, Findler proposed that the algorithm must ensure that the weakening/strengthening dependency must be correct (at least in the current state). This would mean that the previous situation would be evaluated as a logical disjunction, as long as the original partial precondition implies the subcontracted partial precondition (Figure 2.1, left side). If the implication does not hold then the precondition fails, this is due to a failure in the contract refinement, not to a client failure.

$pre_A \vee pre_B$	$(pre_A \vee pre_B) \wedge (pre_A \Rightarrow pre_B)$

Figure 2.1: Precondition evaluation, original (left) and Findler's proposal (right)

Oblivious to this issue, the Ecma standard for Eiffel [ECMA06a] has since revised the original algorithm, relaxing the postcondition evaluation rule. Namely, when refining postconditions, not all of the partial postconditions must be held, only the ones whose partial precondition is true. This is a logical corollary of the rule that states that postconditions must only be held when preconditions were met. Following the example of this section, postcondition evaluation in the original Eiffel algorithm would be the following the simple conjunction of the partial postconditions (Figure 2.2, left side). The relaxed version would be the conjunction of postconditions whose corresponding preconditions where valid (Figure 2.2, right side).

$post_A \wedge post_B$	$(pre_A \Rightarrow post_A) \wedge (pre_B \Rightarrow post_B)$

Figure 2.2: Postcondition evaluation, original (left) and Ecma's revision (right)

2.1.3 Contract Levels

There are several types of contracts in software development [Beugnard99]: *syntactic contracts*, typically defined by interface definition languages (e.g. Corba [OMG04]) and type mechanisms of programming languages; *behavioural contracts*, typically defined by specification languages (e.g. OCL [OMG05]) or by assertions in programming languages (e.g. Eiffel); *synchronization contracts*, defined by mechanisms that ensure atomicity of services and operations (e.g. Java's synchronized keyword for methods), and *quantitative contracts*, defined by quality-of-service insurance mechanisms. The scope of contracts studied in this dissertation is on behavioural contracts, although in some situations syntactic and synchronization contracts can be partly addressed by these contracts.

2.1.4 Development and Production

When a software developer meets Design by Contract for the first time, a question immediately follows: how is performance affected by the assertions? Since Design by Contract focuses on a clean and documented design, in its simplest form the contract can be expressed within comments. However, the contract becomes much more useful if it can be executed in run-time. In such case, performance is a real concern, since validating conditions adds a considerable overhead to execution. Should Design by Contract be validated only during development or also in production software? To answer this question there are two positions.

The “conservative” position claims that since Design by Contract addresses reliability, it makes no sense to have a reliable system in development, but unreliable in production (where software faults present a real harm). In an interesting analogy, Hoare [Hoare89] compares this situation with someone that uses a life jacket while learning to ride a boat on a lake, but leaves it at home when going to the sea.

The “liberal” position acknowledges that in some situations, performance *is* part of the reliability, that is, the system is only useful if it executes within a certain time frame (e.g. a real-time application). As such, these authors propose that in production, contract verification should disabled, at least partially. Usually, the partial verification is for the preconditions (it is useless to validate postconditions if preconditions are ignored – if the client does not meet the conditions requested, the supplier is not obliged to fulfil the contract). Some authors go a little further [Mitchell02] and suggest that this should be considered while specifying the contract, and as such, the preconditions should be very fast to execute (even if it makes the postconditions slower).

2.1.5 Design by Contract vs Defensive Programming

A naïve approach for software reliability assumes that software developers can never be sure that the caller of a certain routine in a module will provide the correct parameters, and as such, the routine should always assume the worst and check in its body the validity of the parameters[3]. An approach that accepts such premise is usually called *defensive programming*[4]. How does Design by Contract cope with that?

Design by Contract proponents [Mitchell02] [Meyer92] claim that such premise is wrong, for a number of reasons:

- the client may already be performing the verification, and in that case it is redundant;
- validating conditions in the body of the routine introduces more complexity and more code, which in the long run will result in more bugs;
- this verification may “obscure” some client bugs (by handling them in supplier code), therefore adding difficulty to the debug process;

3 Possibly an application of Murphy's Law.

4 Note that this is an informal term that can be used to designate a wide array of techniques, including Design by Contract. We employ it here in the scope of module communication through function/routine calls.

- documenting these conditions is harder, for it is common to change a routine and forget to change its documentation (especially with deadlines).

Nevertheless, the scope of Design by Contract is in calls between two software modules. As such, there are some situations where it cannot be applied, such as user input (e.g. console or form inputs) or heterogeneous distributed applications (e.g. web applications). In these situations defensive programming is the viable solution. According to Meyer, this does not invalidate Design by Contract – he proposes that in the points where the client cannot be "trusted", a layer of defensive modules should be placed.

2.1.6 Design by Contract vs Unit Testing

Test-Driven Development [Beck99] [Beck03] is an agile software design method that involves repeatedly writing test cases and then implementing the necessary code to pass the tests, in order to provide rapid feedback. To apply Test-Driven Development cycle, an automated testing tool is required. From this need, the concept of Unit Testing has emerged: a procedure that is used to validate that individual modules or units of source code are working properly. Today, there is a growing popularity of Unit Testing, with various frameworks available, most of them collectively know as xUnit, since they are based on a design originally implemented for Smalltalk as SUnit [Beck94].

There is some confusion between Design by Contract and Unit Testing, since both of them address software correctness through assertion mechanisms. However, although they intersect each other, these are two distinct approaches, with different goals and properties. While Design by Contract is concerned with the general correctness of client/supplier module responsibilities, Unit Testing is concerned with the correctness of a single module under specific situations. Design by Contract provides good documentation and clean design; Unit Testing provides a set of repeatable test cases that ensure the consistency of the software throughout modifications.

In summary, these two approaches are independent, but can be used together, since they complement each other. In fact, as it will be shown, some solutions make these mechanisms synergetic.

2.1.7 Specification languages

Specification languages provide a way to formalize contracts in a much concrete way than a programming language. However, specification languages are usually relatively complex. This makes them difficult to implement, and worse, difficult to adopt by software programmers. As such, and since this dissertation aims to present a simple solution in terms of programmer adoption, we will not go into detail over this subject. Nevertheless, specification languages such as Z [Abrial80], VDM [Bjørner78], Larch [Guttag93], their object-oriented extensions, and more recently UML's Object Constraint Language (OCL) [OMG05], had a considerable influence in most Design by Contract solutions, either implicitly or explicitly (syntactically). Furthermore, there is some work on bridging analysis and design with OCL contract specification in Java [Dzidek05].

The Java Modeling Language [Leavens99] is a behavioural interface specification language for Java modules. While not a programming language *per se*, there are various tools that use this specification, generating Java code from it). As it will be discussed in subsequent sections, these tools can be used to implement simple Design by Contract, and even go further.

2.2 Programming languages with native support

In this section, some programming languages with native support for Design by Contract will be presented. These languages are representative in terms of objectives (some are industrial while other are academic) and architecture (some generate native executable code, while other generate virtual machine *bytecode* or even C source code). Finally, mainstream programming languages with no native support for Design by Contract but with some simple assertion mechanism, will also be briefly discussed.

2.2.1 Eiffel

Eiffel [Meyer91] [ECMA06a] is a programming language designed by Bertrand Meyer and developed by the Interactive Software Engineering, among others. Since Eiffel introduced Design by Contract, it was the first language to implement it. Also, in terms of "mainstream" (that is, non-research) programming languages, it is still one of the few to implement it.

As shown in the left side of Listing 2.1, preconditions are specified in the require clause and postconditions are specified in the ensure clause. Multiple conditions can be specified in the clauses, being evaluated from top to bottom. Invariants are specified in the invariant clause, and are not attached to a routine, as they apply to the whole class.

A common reliability condition is that the execution of a method only affects a specific set of features: the *frame problem*. To solve this, postconditions have an optional only clause, which is used to specify which features' values can be modified as a consequence of the routine execution.

Contract assertions are checked before and after the execution of public routines and constructors (with the exception of invariants, which are not checked before constructors).

Since contract assertions are Eiffel regular code, it is possible to use all the expressiveness of the language, including calling other routines. Routine calls in assertions are not verified for assertions.

```
class ClassName feature
    ...
    routineName(parameter list) is
        header comment
    require [ else ]
        precondition
    do
        body
    ensure [ then ]
        postcondition
        [ only feature list ]
    end
    ...
    invariant
        invariant
end
```

```
from
    initialization
unit
    exit condition
invariant
    invariant
variant
    variant
loop
    body
end
```

Listing 2.1: Eiffel basic assertion syntax (left) and loop syntax (right)

Eiffel supports contract inheritance as previously defined, that is, obeying to the Liskov Substitution Principle. *Contract extensions* preconditions must be written as require else, and postconditions as ensure then, in order to force the programmer to be syntactically aware of the subcontracting, as well as simplifying the underlying implementation of contract checking. When routines are overridden without specifying assertions, the super class assertions are inherited. Missing contract extensions are equivalent to true for preconditions and false for postconditions.

In postconditions, there is a Result variable and the old expression. The Result variable refers to the return value of the routine, and can be used to make assertions on the expected result (e.g. in a routine that returns an integer, it is possible to assert that the value is greater than zero). The old expression allows referring to the "old" state of the object, that is, the state before the execution of the routine (e.g. it is possible to assert that a certain attribute was modified by the routine in the expected way). Although the old expression is pre-computed before the actual method execution, if its evaluation yields an exception, the exception "throwing" is deferred until the postcondition.

Eiffel also supports the check instruction and loop variants/invariants. The check instruction validates an assertion at any point of the routine body. The loop variants/invariants, as shown in the right side of Listing 2.1, specify further assertions in loops: the *loop invariant* is a condition that must always be true in every iteration; the *loop variant* is an integer expression whose value is non-negative after initialization, and must be decreased in every iteration, consequently assuring that the loop is finite.

In Eiffel, it is possible to select the level of contract verification at compilation time, class by class. The level of contract can be:

- no: no checking;
- require: just preconditions are checked;
- ensure: preconditions, and postconditions are checked;
- invariant: preconditions, postconditions, and invariants are checked;
- loop: preconditions, postconditions, invariants, and loop variants/invariants are checked;
- check: all (preconditions, postconditions, invariants, loop variants/invariants and check statements are checked).

In case of contract violation, a run-time exception is raised by the routine in cause. Eiffel offers an optional rescue clause, that specifies the behaviour of the routine when a certain exception type is raised. If none is written, the default behaviour is to pass the exception to the calling routine. If the exception reaches the main routine, the execution halts.

Due to its simplicity and power in specifying contracts, Eiffel offers an elegance that makes it a "benchmark" language for evaluating Design by Contract implementations.

2.2.2 Blue

Blue [Kölling98] is a programming language essentially designed for Object-Oriented teaching.

As shown in Listing 2.2, invariants are written in the invariant clause of the class. Pre/postconditions are written in the pre and post clauses of the routine, respectively before and after variable declarations and routine body.

Class invariants are checked before and after executing public routines, but only when called from outside the class. Blue follows the substitution principle, as in Eiffel.

Assertions can use the "implies" (⇒) conditional operator. Postconditions can use the old keyword to access expression values of the object's state before routine execution.

In summary, Blue's Design by Contract is very simple, essentially a subset of Eiffel, reflecting the

```
class ClassName is
    internal
        ...
    interface
        routineName(parameter list) -> (returnType) is
            pre
                precondition
            var
                ...
            do
                body
            post
                postcondition
        end routineName
    routines
        ...
    invariant
        invariant
end class
```

Listing 2.2: Blue basic assertion syntax

nature of the language, the teaching of "good" Object-Oriented Programming techniques.

2.2.3 Chrome

Chrome is a programming language developed by RemObjects Software [RemObjects06], which produces *bytecode* for the Common Language Infrastructure (the Microsoft .NET Framework specification [ISO06]). Chrome is based in Object Pascal [Inprise99] but extends it for some new features, such has a concept of "class contracts".

As shown in Listing 2.3, preconditions are specified in the require clause and postconditions are specified in the ensure clause. Multiple conditions can be specified in the clauses, and are evaluated from top to bottom. Invariants are specified in the invariants clause, and are not attached to a routine, as they apply to the whole class. This is very similar to Eiffel.

Contract conditions are checked only after the execution of public methods. As for invariants, Chrome supports two levels of invariants: public and private. Public invariants are only checked after public method executions, but private invariants are checked after every method execution, both public and private. Since conditions are only checked after executions, Chrome applications may be subject to the

```
type

   ClassName = public class
   public
      method MethodName(parameter list): returnType;
      require
         Precondition
      begin
         Body
      ensure
         Postcondition

      other method declarations...

   public invariants
      Invariant

 end;
```

Listing 2.3: Chrome basic assertion syntax

Indirect Invariant Effect. Meyer [Meyer97] defines this problem as a situation in which "(...) an operation may modify an object even without involving any entity attached to it.". This situation occurs when an object holds a reference to another object, which in its turn holds a reference to the original object. The consequence of this effect is that verifying class invariants after method executions is not enough, it is necessary to verify them also before.

As usual in a language with native Design by Contract support, the conditions are written in the same language, taking advantage of the expressiveness of the language, including calls to other methods (and consequent contracts evaluations). Contract inheritance is not supported.

The definition of the level of contract verification is very limited: when compiling in release mode (typically for production), the contract code is not generated (this is achieved through a compiler flag). Similarly to most modern programming languages, an assertion failure will raise an exception.

In spite of its simplicity, Chrome is the first programming language to introduce Design by Contract in *Rapid Application Development* tools [Martin91].

2.2.4 D

D is a recent[5] programming language developed by Digital Mars [DigitalMars06]. Its goal is to be a powerful programming language (as C++) but offering modern programming mechanisms (as Java and C#). In order to be a powerful language it offers "low-level" features, such a memory management and in-line *assembly* code; and "modern" features, such as *garbage collection*, *unit testing*, and *contract programming*[6].

As shown in Listing 2.4, preconditions are specified in the in clause and postconditions are specified in the out clause. Invariants are specified in the invariant clause, and are not attached to a routine, as they apply to the whole class. As in Eiffel, multiple assertions can be composed together, being evaluated from top to bottom. Each assertion is enclosed in an assert() instruction.

Contract conditions are checked in the same situations as in Eiffel: before and after the execution of public methods and constructors (with the exception of invariants, which are not checked before constructors). Additionally, since the D language supports *destructors*, the contract is also checked before its execution.

Just like in Eiffel, the conditions are written in the language. allowing to use most of the expressiveness of the language. A new rule is that the code in the invariant may not call any public non-static members

```
class class_name {

    attribute declarations...

return_type function_name(argument declarations)
    in
    { Precondition }
    out (result)
    { Postcondition }
    body
    { Routine body }

    other method declarations...

    invariant
    { Invariant }
}
```

Listing 2.4: D basic assertion syntax

of the class, either directly or indirectly (to avoid infinite recursion). The absence of side-effects must be ensured by the programmer.

Condition inheritance is supported in D, following the Liskov Substitution Principle. However, there is a difference regarding its implementation. While in Eiffel an absent precondition in the hierarchy tree is considered having a false value (except if it is the only one), in D it is considered a true value. This means that if any function in an inheritance hierarchy has an absent precondition, then preconditions of functions overriding it will have no useful effect.

In postconditions, there is a result variable, which must be declared in the *out* argument section explicitly (but without the type). There is no old state mechanism.

The definition of the level of contract verification is more limited than in Eiffel: when compiling for release, the contract code is not generated. Unlike in C, the assert failure does not halt execution, it raises an error in a similar fashion as in Java (see next section for an explanation of C and Java

5 The final specification of the language (1.0) was released during the writing of this dissertation, in January 2, 2007.

6 Some implementations use the term "contract programming" or "programming by contract" instead of "design by contract". This is mostly to trademark issues.

assertion mechanisms).

Although more simple than Eiffel's implementation, the D programming language introduces Design by Contract coherently with its philosophy: to produce a flexible, efficient, and pragmatic language.

2.2.5 Lisaac

The Isaac project at INRIA developed *Isaac* – an object prototype based operating system and *Lisaac* – an object prototype based language. This object-oriented, prototype-based language [Benoit04] produces ANSI C [Kernighan88] code instead of machine code, in order to provide cross-platform portability.

As shown in the left side of Listing 2.5, preconditions are specified before the *slot*[7] body and postconditions are specified after the slot body. Multiple conditions can be specified, being evaluated from top to bottom. Invariants are specified after the slot definitions, as they apply to the whole class.

Contract conditions are checked before and after the execution of public slots and constructors. If the slots are called from within the class without the self[8] object, the invariants are not verified.

Since contract assertions are regular code, it is possible to use all the expressiveness of the language. It is also possible to declare local variables.

```
Section Header
   class definition...

Section Public
   slot definitions...

   - slotName <-
   [ precondition ]
   ( body )
   [ postcondition ];

[
   invariant
]
```

```
// unary message assertion:
? { foo > 616 }

// binary message assertion:
6 ? { bar > 616 }
```

Listing 2.5: Lisaac basic assertion syntax (left) and assertion types (right)

Lisaac supports simple contract inheritance. Unlike other languages though, inheritance is explicit, which means that it is possible to completely override the contract in a class. As such, there is no (implicit) preservation of the Liskov Substitution Principle.

Lisaac support a Result variable and an Old mechanism, with some differences. The old mechanism can be used both in postconditions as well as in invariants. Since the language supports results in the form of *ordered n-tuples*, the result variable can be used to address a specific tuple element by its index.

Explicit assertions exist, similar to C's assert function. Any assertion failure, including failures in preconditions, postconditions, and invariants, provokes program abortion.

At compilation time, there is a debug_level variable, which indicates what level of verification is to be performed. Assertion with priority level greater that the specified are not validated (e.g. if debug_level is set to 4, level 1 and 3 assertions are verified, but not level 5). Assertions can be specified without priority though, as shown in the right side of Listing 2.5.

In conclusion, Lisaac's Design by Contract mechanisms reflect the language's academic nature, with its somewhat unusual syntax, and its operating system perspective, with its low-level assertion handling mechanism.

7 In Lisaac naming, a slot is a class method, function or attribute.
8 The *self* object is the analogous to the *this* object in C++/Java.

2.2.6 Nemerle

Nemerle [UniversityOfWroclaw06] is a high-level statically-typed programming language for the .NET platform, developed at Computer Science Institute of the University of Wroclaw. It is a multi-paradigm language, offering functional, object-oriented and imperative features.

As shown in Listing 2.6, preconditions are specified in the requires clause, and postconditions in the ensures clause. Invariants are specified in the invariant clause, just after class declaration.

Contract preconditions and postconditions are checked as usual, but invariants are only checked after method executions, and after expose blocks. The expose block redefines the granularity of invariant checking. While usually method executions are not atomic, code in the expose block is considered atomic and is not checked.

Contracts are written using the regular language syntax, including method calls. The assertions also have an optional otherwise clause, on which is possible to state specific behaviour, in case of check failure.

In postconditions, there is a value variable for evaluating the result of the method execution. Some common features are not yet supported: the old mechanism, contract inheritance, and contract disabling.

In conclusion, current support for Design by Contract in Nemerle is very basic. Future development will dictate the usability of its Design by Contract support.

```
class ClassName
invariant invariant
{
    public methodName (parameter list): returnType
    requires precondition
    ensures postcondition
    {
        body
    }
}
```

Listing 2.6: Nemerle basic assertion syntax

2.2.7 Sather

Sather is an object-oriented language that was designed in 1990, at the International Computer Science Institute at Berkeley University of California [UniversityOfCalifornia96]. Instead of native binary code, the compiler produces C code. Sather's development has been discontinued since 1996.

As shown in Listing 2.7, preconditions and postconditions are specified in the pre and post clauses, respectively. Invariants are specified through an invariant method (a regular method that by convention has no arguments and returns a Boolean value).

Contract conditions are checked before and after the execution of (public or private) methods and constructors. Class invariants are checked after public method calls.

Contracts are regular code for invariants, so it is possible to use all the expressiveness of the language, except for declaring local variables, which is only available for invariants.

Sather does not support contract inheritance.

In postconditions, it is possible to use the result variable. It is also possible to use the old mechanism, in the form of the initial() function. This function takes an expression as argument (which cannot contain initial calls, nor the result variable).

Sather also supports an assert statement, that like the remaining contract mechanisms, provokes the program to exit with a fatal error if the asserted expression evaluates to false..

```
class CLASS_NAME is
   attributes

   methodName(parameter list): returnType
   pre precondition
   post postcondition
   is
      body
   end;

   invariant: BOOL is
      invariant
   end;
end;
```

Listing 2.7: Sather basic assertion syntax

At compile time, it is possible to disable assertion checking.

In conclusion, the Sather programming language offers a poor set of Design by Contract functionalities.

2.2.8 Other programming languages

Although Design by Contract is acknowledged as an interesting approach, few programming languages have native support for it. Nevertheless, there are assertion mechanisms in mainstream programming languages that are worth mentioning.

The C programming language [Kernighan88], and consequently C++ [Stroustrup00], have an assert() macro which takes a Boolean value as argument. If evaluated false, a message is printed and execution is aborted. Assert macros can be disabled through a compiler flag.

In response to public demand for Design by Contract, Java 1.5 [Gosling05] introduced the assert statement, which takes a Boolean expression as argument. If the expression evaluates false, an error is raised (therefore halting execution, unless the error is handled[9]). Assertions can be disabled/enabled in a per-class and/or per-package manner, through JVM switches.

The PHP Hyper Processor script language introduced the assert() function in version 4 [PHPDocumentationGroup06]. Depending on the assert options provided earlier, the assertions can be disabled, terminate execution, issue warnings and/or handled by a user-defined function. The options are provided explicitly in the code in a per-module manner (class, function or simply in-line code).

The .NET Framework [Microsoft06] since version 1.1 provides an assert method in the Debug class. If the assertion fails, a dialogue box with a stack trace is presented (if the application is running in user-interface mode), allowing the user to take debug actions (“abort”, “retry” and “ignore”). It also possible to provide a user-defined function (called a *trace listener*).

Finally, many languages (e.g. the Python [Rossum06] script language and the OCaml [Leroy05] functional language, etc.) have an assertion statement very similar to Java, but without the possibility of disabling it per-class.

2.3 Language extensions for Java

In this section, language extensions for Java that provide support for Design by Contract are presented. The idea is to extend the language syntax, in order to mimic Eiffel's mechanisms. For historical reasons, the syntax for contracts in the original Java specification (Oak) is also presented.

9 In the Java language, errors are similar to exceptions (both are subclasses of Throwable).

2.3.1 Oak

The original Java language specification, called “Oak” [FirstPerson94], provided support for assertions. This support was never part of the final specification, and as such not implemented, but is described here for historical reasons.

Oak supported pre/postconditions, as the first and last statements of a method, respectively. Also, it supported constraints on instance variables and methods. Both these features are shown in Listing 2.8. Failures to such assertions resulted in an `AssertionFailedException`.

Pre/postconditions in Oak are inherited in subclasses, but cannot be changed. While this makes contract inheritance respect the Liskov substitution principle, it is limiting, since preconditions cannot be weakened, and postconditions cannot be strengthened.

Variable constraints work somewhat like class invariants, since they are enforced at the entry and exit of every `public` or `protected` method.

In summary, the Oak language featured a simple, but interesting support for Design by Contract. Ironically, one of the features it did not support was a “check” statement, the only feature currently implemented in Java.

2.3.2 ContractJava

```
class ClassName {
   datatype variable assert(constraint);

   returnType methodName(parameter list)
   {
      precondition: precondition;
      method body
      postcondition: postcondition;
   }
}
```

Listing 2.8: Oak assertion syntax

ContractJava [Findler01] is a language extension based on the study of the implementation of existing Design by Contract solutions, as well as stating and proving a *contract soundness theorem* for the behavioural subtyping principle contract verification. The ContractJava language extension is supported by a “contract compiler”, that is, a specialized compiler for producing Java regular bytecode, but including wrapper methods for method calls under contract verification.

The authors argue that the algorithm used by Design by Contract solutions (including Eiffel) is flawed, in the sense that the conditions evaluated by disjunction/conjunction are necessary, but not sufficient [Findler01a]. They claim that the flaw is in the fact that the algorithm assumes that the programmer writes overridden preconditions that are never strengthened and postconditions that are never weakened, but the algorithm does not actually check this. This can result in situations where contract blame is incorrectly assigned or undetected. As such, they propose new conditions: partial method preconditions imply the preconditions of subclasses overridden methods, and partial method postconditions imply the postconditions of super classes overridden methods. Failure of these conditions results in an exception, in which blame is assigned to the class that is incorrectly declared as a subtype, since it does not preserve the subtyping principle. Actually, the Eiffel standard [ECMA06a] does state this condition (section 8.9.4), but does not specify it in the implementation algorithm.

As shown in Listing 2.9, pre/postconditions are written after methods as Boolean expressions, using the `@pre` and `@post` clauses, respectively. Class invariants are not supported. Postconditions have no access to any “result” or “old” special variables.

In summary, the ContractJava language extension is focused on formalizing the base for correct subtyping with contracts (including interfaces and multiple inheritance), but does not feature much more than that. The absence of class invariants, “result” and “old” in postconditions, or any advanced feature, makes it insufficient for development. Also, they do not address important issues, such as contract recursion, or absence of side-effects in assertions.

```
class ClassName {

   returnType methodName(parameter list) {
      method body
   }
   @pre { precondition }
   @post { postcondition }
}
```

Listing 2.9: ContractJava basic assertion syntax

2.3.3 Handshake

Handshake [Duncan98] is a Java extension that adds contracts, through a new module, the contract. Handshake's architecture is based on a compile which produces the binary contracts, and a *dynamically linked library* that intercepts the JVM file operations calls in C, and the contract code is in-lined in the class code.

As shown in Listing 2.10, contracts are written in separate files. The contract name is the same as the interface or class it is contracting. Invariants are written in the invariant statement. Pre/postconditions are written after the method in the pre and post statements, respectively. Each assertion has an optional message, which is displayed in case the assertion fails.

Contract invariants are checked before and after any non-private method call. Contract inheritance follows the subtyping principle. It is possible to use the $result variable.

In summary, Handshake presents an alternative architecture and a new module. The architecture is apparently very efficient, since it works on low-level. However, this means it is not operating system portable, and possibly incompatible to new JVM releases or third-party implementations. Also, as acknowledged by the authors, it does not work with JAR files, which is very crippling, since most Java programs are distributed that way. The new module, while simplifying to the implementation (and not requiring access to the source code), breaks the premise that contract are part of a module. Finally, the fact that method signatures must be repeated in the contract makes it a little cumbersome; although this could be mitigated through IDE tool support.

```
interface InterfaceName {
   returnType methodName(parameter list);
}

contract InterfaceName {
   invariant invariant [ message ];

   returnType methodName(parameter list);
      pre precondition [ message ];
      post precondition [ message ];
}
```

Listing 2.10: Handshake basic assertion syntax

2.3.4 KJC

Lackner *et al.* [Lackner02] propose an extension to the Java language with support for contracts, through the KJC compiler.

As shown in Listing 2.11, pre/postconditions are specified in the @require and @ensure clauses, before and after the method body, similarly to Eiffel. Invariants are specified in the @invariant clause. These clauses can be used with classes and with interfaces. Assertions are written using the Java 1.4 assert statement, which means the assertions are constrained by Java assertion enabling/disabling mechanisms.

Invariant checking is performed before and after non-private methods and constructors (unless constructors terminate through exception). Unlike most implementations, invariant failures absorb postcondition failures, although they are evaluated after. Contract inheritance follows Eiffel closely, although acknowledging Findler's proposal [Findler01].

Postconditions can access the result value of the method body, through the @@() construct. The old value of expressions can be accessed using the some construct, in the form: @@(expression). Since these expressions are evaluated before the method execution, side-effects in them will modify the

```
class ClassName {
    @invariant {
        assert invariant;
        ...
    }

    returnType methodName(parameter list)
        @require {
            assert precondition;
            ...
        }
        {
            body
        }
        @ensure {
            assert postcondition;
            ...
        }
}
```

Listing 2.11: KJC basic assertion syntax

method behaviour.

In summary, this solution is the most complete of all those base of an extension to the language and the one that most closely follows Eiffel, both in terms of syntax and semantics.

2.4 Traditional ad-hoc solutions for Java

As shown in the previous section, not many programming languages have native support for Design by Contract, and this is especially true for mainstream languages. However, most mainstream languages have one or more third-party libraries that implement these features. In this section, such libraries and tools for the Java programming language are presented.

2.4.1 C4J

C4J [Bergström06] is a Design by Contract solution that uses class loading instrumentation. Contracts are written in plain Java code, in separate classes. These contracts can apply to classes or interfaces.

As shown in Listing 2.12, contracts are written in a separate class (by convention adding the "Contract" suffix, but not mandatory), which is specified in the ContractReference annotation. Preconditions are specified in the "pre_" method, postconditions in the "post_" method, and class invariants in the classInvariant method. Assertion methods are void, since they use the Java assert statement. Contracts for interfaces follow the same rules.

Contract inheritance respects the Liskov Substitution Principle, for classes and interfaces. Contracts defined on the same inheritance level are merged[10] (e.g. a class that implements two interfaces with contract on the same method). It is also possible to define contracts for *abstract methods*.

In postconditions, it is possible to use the old and result mechanisms, with some changes in the syntax (as shown in Listing 2.13). The contract class must extend the ContractBase parametrized class, in order to use the getPreconditionValue() and getReturnValue(). The first one must be previously set in the precondition, using the setPreconditionValue(), using a tag string to identify the value. The

```
@ContractReference(contractClassName = "classNameContract")
public class ClassName
{
   // ...
   public returnType methodName(parameter list) {
      body
   }
}

public class ClassNameContract
{
   Dummy target;

   public ClassNameContract(Dummy target) {
      this.target = target;
   }

   public void classInvariant() {
      assert invariant;
   }

   public void pre_methodName(parameter list) {
      assert precondition;
   }

   public void post_methodName(parameter list) {
      assert postcondition;
   }
}
```

Listing 2.12: C4J basic assertion syntax

ContractBase class also allows access to private members, using the getTargetField() method.

C4J has the concept of *pure methods*, that is, methods that do not change the state of the object. Simple *getters*[11] are automatically detected as pure, and the remaining must be explicitly marked as such using the Pure annotation.

In summary, despite its lack of advanced features, C4J provides a simple way of writing contracts using just Java. The use of executable code allows the programmer to use regular Java tools (such a IDE support), and simplifies the procedure of unplugging the library. It also interesting to note that C4J takes advantage of Java "recent" features [Gosling05], such as *generic types*, *annotations*, assert

10 In the studied version, this merging only happens in preconditions.
11 In Java naming, the methods that return and modify a private member variable are called "getters" and "setters", respectively.

```
public class ClassNameContract extends ContractBase<Dummy>
{
   // ...

   public void pre_methodName(parameter list)
   {
      super.setPreconditionValue(tag, value);
   }

   public void post_methodName(parameter list)
   {
      valueType foo = super.getPreconditionValue(tag);
      returnType bar = super.getReturnValue();
      assert postcondition;
   }
}
```

Listing 2.13: C4J postcondition extended features

statements, and Java Agent instrumentation.

2.4.2 iContract2

iContract2 is a preprocessor/code-generator for Java, based on iContract [Kramer98] [Enseling01], the first Design by Contract tool for Java, which was developed by Reliable Systems, but has been discontinued since. In iContract2, the contracts are specified in comments, similar to Javadoc[12].

As shown in the left side of Listing 2.14, preconditions are specified in the pre comment and postconditions are specified in the post comment. Multiple conditions can be specified in the annotations, being evaluated from top to bottom. Invariants are specified in the inv comment. Contracts can be specified in either classes or interfaces.

Contract conditions are performed for public, protected, and package methods. Contract preconditions are checked as expected, but postconditions are checked upon exit and exception of methods. In case of static methods, only public ones are checked.

To avoid infinite recursion, iContract provides a mechanism to provide safe contract recursion.

As shown in the right side of Listing 2.14, iContract also can specify which class is to be used, in case

```
/**
 * @inv invariant
 */
class ClassName
{

   /**
    * @pre precondition
    * @post postcondition
    */
   returnType methodName(parameter list)
   {
      body
   }
}
```

```
class ClassName
{
   /**
    * @pre item != null #NullPointerException
    */
   void setItem(Item item)
      throws NullPointerException {
      // ...
   }
}
```

Listing 2.14: iContract basic assertion syntax (left) and optional exception clause (right)

12 Javadoc is the industry standard for documenting Java classes, used by tools to generate API documentation.

of a verification failure (the default is RuntimeException, the root class of unchecked exceptions).

Contract conditions are Java Boolean expressions, and therefore they can use most of the Java's expressiveness, including method calls. Additionally, it is possible to use first-order logic quantifiers and operators, namely: forall (universal quantifier), exists (existential quantifier), and implies (conditional operator). Quantifiers can be used in objects of the following classes: Enumeration, Array, and Collection. Their syntax is shown in Table 2.2.

Table 2.2: iContract quantifier syntax

forall Class var in Enum | ExprVar
exists Class var in Enum | ExprVar

Contract inheritance follows the "type substitution principle" (Liskov Substitution Principle), for all Java type extension mechanisms: class extensions, inner classes, interface implementation, and interface extension.

In postconditions, there are the result and @pre variables that access the return value of the method, and the state of the object before the method execution, respectively. This follows OCL naming. The classes on which @pre is used must implement Cloneable, or else only the object reference is stored.

iContract is multi-thread safe, as long as no quantifier, result, and @pre mechanism is used.

The types of conditions to validate are given at compile time, through an argument.

2.4.3 Jass

The Jass (or **J**ava with **ass**ertions [Bartetzko01]) tool, is a pre-compiler tool that checks contracts provided by comments.

As shown in the left side of Listing 2.15, preconditions are specified inside a require comment, at the start of the method body; postconditions are specified inside an ensure comment, at the end of the method body. Class invariants are specified in the invariant comment, at the end of the class. Although contracts are specified as comments, they can only be used in classes.

Contract invariants are checked before and after the execution of methods. Contract assertions are Boolean expressions, therefore being able to use most of the Java's expressiveness, including method calls.

Contract inheritance follows Liskov Substitution Principle rules as defined by Findler. Unlike in other solutions, classes that wish to follow the substitution principle by inheriting a contract, must explicitly do so by implementing the jass.runtime.Refinement interface. As defined by Findler, Jass distinguishes between client faults, server faults, and *design faults*.

In postconditions, it is possible to use the special Result and Old variables, which have the expected

```
class className implements Cloneable
{
    returnType methodName(parameter list)
    {
    /** require precondition **/
        body
    /** ensure postcondition **/
    }

    /** invariant invariant **/
}
```

```
class Point3D
{
    int x, y, z;
    ...

    void moveX(int n)
    {
        ...
        /** ensure changeonly{x}; **/
    }
}
```

Listing 2.15: Jass basic assertion syntax (left) and frame rules example (right)

meaning. Additionally, it is possible to use the changeonly construct. This special construct is used to list the attributes that can be changed. If specified, only these declared attributes are allowed to change. This feature is called *frame rule*, a solution to the frame problem. The right side of Listing 2.15 provides an example.

Additionally, Jass supports the check, rescue and retry statements. The check statement has the same meaning as the assert statement, which at the time Jass was designed was not part of the Java language. The rescue statement can be placed at the end of the method body, in order to specify which assertion exceptions should be caught, and what code blocks are to be executed (this is implemented through try/catch blocks). The retry statement can be used in rescue blocks to re-initiate the method, possibly with different parameter values.

A feature that is innovative in Jass, when compared to other solutions, is the *trace assertions* feature. This feature is used to specify the intended dynamical behaviour, using a Communicating Sequential Processes [Hoare85] like notation for describing allowed traces of events. Listing 2.16 exemplifies this feature, for the factorial function.

Some interesting features, which have a partial support is the *interference check* and Javadoc support. Interference check is a simple facility to detect situations when assertions in one thread may become invalid through statements in another thread. So far, the support is limited, since all threads must be started by a main method, and the Java synchronized modifier is not taken into account. Javadoc is supported adding HTML code for the assertions, in the generated source code.

The Jass solution adds some interesting features, regarding previous solutions, namely frame rules and trace assertions. Additionally, it implements Eiffel's loop variants/invariants, supports some level of concurrency, and adds Javadoc support.

2.4.4 Jcontract

Jcontract [Parasoft05] is a commercial Design by Contract solution developed by Parasoft Corporation. Unlike previous tools, it is part of a larger solution, the Parasoft Automated Error Prevention, which

```
public class Factorial {
    public int factorial(int value) {
    /** require value > 0; **/
    if (value == 1)
        return 1;
    else
        /** ensure Result > 0; **/
    }
    /** invariant [variant] trace (
        MAIN() {
            int value;
            factorial(?value).b -> CALL Decrease(value)
        }
        Decrease(int variant) {
            int nextVariant;
            IF(variant < 0) {
                EXECUTE(throw new RuntimeException ("negative method variant!");)
                -> STOP
            } ELSE {
                factorial(?nextVariant).b WHERE(nextVariant < variant)
                -> CALL Decrease(nextVariant)
            }
        }
    ); **/
}
```

Listing 2.16: Jass trace assertions example

includes Jtest (unit test tool), a methodology, and a GUI.

As shown in Listing 2.17, contracts are written in javadoc-like comments. Preconditions and postconditions are written before the method code, while invariants are written before the class code. Additionally, Jcontract supports the assertion tag, which is used to write assertions at a specific point of code.

Contracts can be specified for public, package, protected, and private methods. In postconditions, it is possible to use the $result keyword, in order to access the return value of the method.

Configuration can be done through a GUI, in order to restrict the classes and/or packages to be checked, the types of checks to be performed, and the runtime handler to be used.

Finally, it is also possible to automatically generate test units for the Jtest tool. In summary, the Jcontract tool is an interesting tool in the sense that is integrated with a GUI and a unit testing tool. However, it is a proprietary tool that is very "closed", in the sense that there is not significant documentation available, nor a public release version[13].

2.4.5 jContractor

jContractor [Karaorman99] [Abercrombie02] [Karaorman03] is a tool for writing contracts, evaluated

```
/**
 * @invariant invariant
 */
class ClassName
{
   /**
    * @pre precondition
    * @post postcondition
    */
   returnType methodName(parameter list)
   {
      body
   }
}
```

Listing 2.17: Jcontract basic assertion syntax

through reflective *bytecode* instrumentation. This is achieved by adding the library during compilation, which replaces the standard class loader with a specialized one, through the Byte Code Engineering Library (BCEL) [Dahm98] [Apache06a].

As shown in the left side of Listing 2.18, preconditions are specified each in a protected Boolean method, with the same arguments as the original method, following a naming convention. The naming convention is that it must have the same name as the method, but with the "_Precondition" string appended. Postconditions are analogous, but must have an extra parameter for the result value. Invariants are specified in a private Boolean method, with no arguments. The naming convention for invariants is that they must be called "_Invariant".

Invariants are checked before and after every public method call, as expected. Preconditions and postconditions can be specified for protected, package or private methods.

Contract inheritance follows the Liskov Substitution Principle rules, both for class inheritance as well as for interface implementation. In jContractor it is also possible to separate the contract from the class or the interface (which is the only way to assign contracts to interfaces). This is performed simply be specifying a class with the same name of the original class, but with the "_CONTRACT" string appended in the name.

13 An evaluation version for academic purposes was requested, but received no answer.

```
class ClassName implements Cloneable
{
    protected boolean _Invariant()
    {
        invariant
    }

    returnType method(parameter list)
    {
        body
    }
    protected boolean
        method_Precondition(parameter list)
    {
        precondition
    }
    protected boolean
        method_Postcondition(parameter list + result)
    {
        postcondition
    }

    private ClassName OLD;
}
```

```
class className
{
    returnType method(parameter list)
    {
        body
    }
    protected Object
        method_OnException(Exception e)
        throws Exception
    {
        exception handler
    }
}
```

Listing 2.18: jContractor basic assertion syntax (left) and exception handler (right)

In postconditions, it is possible to use the "result" and "old" mechanisms. The RESULT is obtained through the last variable, and OLD is a class member. Note that later requires the class to implement interface Cloneable, and therefore implement the clone() method.

jContractor provides a support library for first-order logic quantifiers and operators (Table 2.3). The Forall quantifier ensures that all the elements of a set meet an assertion. The Exists quantifier ensures that at least one element of a set meet an assertion. The Elements operator, given a set of elements and an assertion, returns a sub-set of elements that meet the assertion. Finally the Implies operator, represents logical implication. Sets are represented through the Java Collection class, except the return type of Elements, which is of type Vector. The existence of an Assertion and Operator interface can be used to extend the model. The provided assertions are InstanceOf, Equal, InRange, and Not (their names are self-explaining).

Table 2.3: jContractor's first-order logic quantifiers and operators

ForAll.in(collection).ensure(assertion)
Exists.in(collection).suchThat(assertion)
Elements.in(collection).suchThat(assertion)
Logical.implies(A,B)

The tool is composed by two utilities: jContractor and jInstrument. The first provides instrumentation at run-time, while the later produces that instrumentation at compile-time. The difference is that, while the later option produces larger *bytecode* (the contract is embedded), it does not require distributing jContractor with the end-user application in order to run. These utilities are useful for specifying which assertions are to be checked, and where are they checked (classes and/or packages).

Additionally, the authors also propose a special type of postconditions – the *exception handlers*, for methods that finish by exception instead of regular return. For this, they propose a method appended with the "_OnException". This exception handler provides an attempt to restore the class invariant or simply reset the state (see the right side of Listing 2.18). However, this proposal has yet to be implemented [Karaorman03].

In summary, jContractor is an interesting solution since, unlike most other approaches, it uses the Java language for expressing the contracts, thus enabling all the advantages of executable code: it can be executed, compiled and debugged using standard Java tools.

2.4.6 JML Tools

As mentioned in Section 2.1.7, Java Modelling Language (JML) is a specification language [Leavens06a] that was closely developed with several tools to support it, in the Iowa State University. This development resulted in more researchers from around the world adopting it for subsequent tools [Leavens00] [Burdy05]. One of the original Iowa Java Modelling Language tools is the Java Modelling Language Compiler (jmlc). The jmlc tool is a runtime checker, based on source code preprocessing, that allows a simple Design by Contract implementation [Leavens06].

As shown in Listing 2.19, preconditions and postconditions are specified in the requires and ensures comment tags, respectively. The class invariants are specified in the invariant comment tag. Additionally, it also possible to write assertions in specific parts of the code, using the assert tag.

JML distinguishes two types of invariants: *type invariants*, that is, assertions that define acceptable states of the object that are client-visible; and *representation invariants*, assertions with access to the

```
class ClassName
{
   //@ invariant invariant

   //@ requires precondition
   //@ ensures postcondition
   returnType methodName(parameter list) {
      body
      ...
      //@ assert assertion
      ...
      body
   }
}
```

Listing 2.19: Java Modelling Language basic assertion syntax

(private) internal state of the object. In any case, invariants are checked after the constructor, and before and after any method execution. However, it is possible to declare a method as helper, meaning that invariants are not checked around these methods, allowing them to temporarily break the class invariants.

Java Modelling Language also distinguishes two types of postconditions: *normal postconditions*, assertions that specify what must true if the method returns; and *exceptional postconditions*, assertions that specify which exceptions may be thrown in the method (through the signals_only clause), and specify what must be true when an exception is actually thrown (through the signals clause).

Contract inheritance is supported in a similar way to Eiffel, respecting the Liskov Substitution Principle. Note that, just like in Eiffel, the also keyword must be used when in the presence of contract inheritance.

In postconditions, it is possible to use the \result and \old(expression) keywords, allowing access to the result value of the contracted method, and the value of a certain expression before the execution of the contracted method, respectively.

Java Modelling Language supports first-order logic through several operators and quantifiers (as shown in Table 2.4). *Generalized quantifiers* (\sum, \product, \min, and \max), as well as a *numeric quantifier* (\num_of), are omitted from the table.

Table 2.4: Java Modelling Language first-order logic constructs

name	syntax
implies operator	expression1 ==> expression2
follows operator	expression1 <== expression2
if and only if operator	expression1 <==> expression2
not if and only if operator	expression1 <=!=> expression2
universal quantifier	(\forall element; collection; expression)
existential quantifier	(\exists element; collection; expression)

Java Modelling Language assertions are guaranteed to be side-effect free, since assignment/modifying operators (=, +=, -=, --, ++, etc.), as well as side-effect methods, are not allowed. Java Modelling Language considers pure methods as side-effect free. The pure methods are the simple getters, and other methods explicitly declared as pure.

Java Modelling Language has many other features: *model variables*, which are local variables; the spec_public declaration, that allows to use private variables of the class as public in the contract; the non_null keyword for class variables, equivalent to var != null invariant; and *informal contracts*, that is informal comments that cannot be verified through a tool.

Beyond jmlc, there are other useful tools supplied: jml, a Java Modelling Language interpreter, in case you do not want to produce compiled code; jmlunit, a unit testing tool that generates JUnit tests [Rainsberger04] for the Java Modelling Language specification; jmldoc, a documentation generator that allows to introduce the Java Modelling Language specification in Javadocs; and finally jmlc-gui, a graphical user interface.

In spite of using a modified compiler, the produced bytecode can be run in a regular Java Virtual Machine, as long as Java Modelling Language runtime library classes are included in the *boot class path*.

In summary, the Java Modelling Language is a very powerful specification language, which is only partially explored in the Iowa Java Modelling Language Tools. There are other tools that take advantage of larger subsets of Java Modelling Language, as discussed later in this dissertation. The fact that many support tools are available is a good reason to use Java Modelling Language, especially because the learning curve can be adjusted with the subset of the Java Modelling Language/tools you are using.

2.4.7 Modern Jass

Modern Jass is a Design by Contract solution developed in the context of a master thesis [Rieken07]. This solution differs from the others, since it takes advantage of the features introduced in Java 6, namely the pluggable annotation processing API [Darcy06]. This feature makes tool integration seamless (compiler, IDE, etc.), as far as the tools support this API.

As shown in Listing 2.20, contracts can be placed in classes and in interfaces, left and right examples, respectively. Invariants can be written in the @Invariant annotation, whereas preconditions, postconditions and exceptional postconditions can be written in the @SpecCase annotation (pre/postconditions can also be written in the @Pre/@Post annotations, but this is only *syntactic sugar*). These annotations have optional attributes, show in Table 2.5.

Other basic annotations include: @Model and @Represents for defining model variables (variables whose scope in the contract assertions) and attributing values to them, respectively; @Pure, for defining pure methods (methods without side-effects that can be used in contracts); @Helper, for defining helper methods (methods that can violate class invariants). Besides the @Pre/@Post annotations, other syntactic sugar annotations exist for defining frequently used assertions: @NonNull, @Length, @Max, @Min, and @Range. Finally, some expression annotations represent special variables in the body of an

```
class ClassName
{
   @Invariant("invariant")
   object attribute;

   @Pre("precondition")
   returnType foo() { body }

   @Post("postcondition")
   returnType bar() { body }
}
```

```
@Invariant("invariant"),
interface InterfaceName
{
   @SpecCase(
      pre = "precondition",
      post = "postcondition",
      signalsPost = "exceptional postcondition")
   returnType foobar();
}
```

Listing 2.20: Modern Jass basic assertion syntax

assertion, namely: @Result, the result value of a method invocation for postconditions; @Signal, the exception value of method invocation which was captured by an exceptional postcondition; @Old, the value of an expression before the execution of the method; @ForAll and @Exists, the usual first-order logic quantifiers.

Table 2.5: Modern Jass basic annotations' attributes

@Invariant		**@SpecCase**	
attribute	**explanation**	**attribute**	**explanation**
value	The actual assertion code.	pre	The precondition assertion code.
visibility	The visibility of the assertion.	preMsg	A user-defined message for precondition failure.
context	Is the assertion to be checked in a static context.	post	The postcondition assertion code.
msg	A user-defined message for invariant failure.	postMsg	A user-defined message for postcondition failure.
-	-	signalsPost	The exceptional postcondition assertion code.
-	-	signals	The exception type that triggers the exceptional postcondition.
-	-	signalsMsg	A user-defined message for exceptional postcondition failure.
-	-	visibility	The visibility of the assertion.

Inheritance follows the Liskov substitution principle, as stated in the Eiffel ECMA standard. Unlike other implementations though, Modern Jass has the concept of contract visibility, which means that for example, private contracts are only checked within that class, and not in its subclasses.

In summary, this solution is quite interesting in the sense that it uses a novel approach to implement Design by Contract. Although it does not implement all the features found in other solutions (e.g. frame rules, disabling mechanisms, it has a promising future, especially because it is being aligned with a JML 5, an upcoming annotation-based release of JML.

2.4.8 STClass

STClass (Self-Testable Class) [Jézéquel01] [Deveaux02] is a contract based built-in testing framework for Java. This framework supports the *Design for Trustability* approach. The Design for Testability development process includes a step that is an extension to Design by Contract. The process is composed by the following steps: specification, class skeleton/methods prototypes, contracts definition, test scenario, code implementation, validation, and quality control. As such, the tool supports definition of contracts and unit tests, within the class.

As shown in Listing 2.21, preconditions, postconditions, and invariants, are written a pre, post and invariant javadoc-like comments. Contracts are available for classes and interfaces, following the Liskov Substitution Principle for inheritance.

Contracts may include first logic operators and quantifiers, namely implies, exists and forall, as shown in Table 2.6. Note that expression is a Boolean expression, and iteration is an expression that returns a java.util.Iterator object.

Table 2.6: First-order logic support in STClass

expression1 implies expression2
forall type object in iteration | expression
exists type object in iteration | expression

In postconditions, the @pre tag allows to refer to the old state of the object, before the method execution. The return variable refers to the return value of the method under contract.

Since, this solution is based in a preprocessor (javacst), and contracts are inside comments, disabling contract checking is accomplished by simply using the regular Java compiler.

```
/**
 * @invariant invariant // description
 */
class ClassName {

   /**
    * @pre precondition // description
    * @post postcondition // description
    */
   returnType methodName(parameter list) {
      body
   }

}
```

Listing 2.21: STClass basic assertion syntax

In summary, in spite of all the problems associated with a preprocessor solution, STClass is interesting in the sense that it provides contract/unit testing integrated solution, supporting an original design approach.

2.5 Aspect-Oriented ad-hoc solutions for Java

In the previous section we have seen several *ad-hoc* solutions for implementing Design by Contract in Java. In this section we show some other solutions that take advantage of Aspect-Oriented Programming.

2.5.1 Barter

Barter [Szathmary02] was one of the first Design by Contract implementations for Java that used AspectJ. It also used XDoclet [Walls03] a metadata processor that preceded the standardization of annotations in Java 1.5.

As shown in Listing 2.22, contracted classes/interfaces must start with a @barter tag. Class invariants are declared with the @barter.invariant tag. Method preconditions and postconditions, are declared in the @barter.pre and @barter.post tags. Besides these, there are still auxiliary tags at class level:

@barter.warning and @barter.error, which emit a compile-time warning or error, if a given *pointcut* exits in the class. At method level, the @barter.before and @barter.after tags allow to execute a determined Java expression before and after executing the method.

Besides tags, there are also some special variables available: $this refers the instance where the contract is being evaluated; $result refers to the result value of the method execution.

Contract inheritance is supported, but does not follow the Liskov Substitution Principle, namely preconditions are logical conjunctions instead of disjunctions.

The most advanced features allow a certain level of configurability. The barter.runtime.ViolationHandlerI interface allows the programmer to specify its own implementation of the handler, whose default behaviour is to throw BarterAssertionFailed error. In terms of assertion configuration, it is possible to define which classes and which assertions are evaluated, by supplying a custom implementation of the barter.runtime.AssertionConfigurationI interface. However, any of these configuration features require the setting of system properties, which implies a recompile on the library.

In summary, Barter is a simple and lightweight Design by Contract implementation, due to the fact that it does not implement the more complex features on basic Design by Contract: Liskov Substitution Principle inheritance and the "old" construct. The configurability is interesting, but implies writing Java code and rebuilding the tool.

```
/**
 * @barter
 * @barter.invariant invariant
 */
class ClassName {

   /**
    * @barter.pre precondition
    * @barter.post postcondition
    */
   ReturnType methodName(parameter list) {
      body
   }
}
```

Listing 2.22: Barter basic assertion syntax

2.5.2 ConFA

ConFA [Skotiniotis04] is a tool for implementing Design by Contract through Aspect-Oriented Programming, and extending Design by Contract for Aspect-Oriented Programming[14].

ConFA's architecture is based on a preprocessing tool developed with DemeterJ [Hulten98], that generates an AspectJ aspect for each class (or interface) contract, that is immediately weaved, producing Java bytecode.

Contracts are written using a language extension, as shown in Listing 2.23. Invariants are placed in the @invariant statement of the class, pre/postconditions are placed in the @pre and @post statements of the method. These statements are the first statements, placed before the method body. The invariant statement can be placed at the beginning or end of the class/interface.

The old keyword in postconditions, allows accessing the state of the object before the method execution. Using this keyword in the old(methodName) form, allows accessing the method's return value.

Contract inheritance follows the rules proposed by Findler regarding the substitution principle.

14 The actual name of the tool is not "ConFA", but it is omitted throughout this dissertation, since the actual name has an inappropriate meaning in Portuguese.

```
class ClassName {
   @invariant{ invariant }

   returnType methodName(parameter list) {
      @pre{ precondition }
      @post{ postcondition }
      body
   }
}
```

Listing 2.23: ConFA basic assertion syntax

In summary, ConFA is a fairly complex tool, since it is a language extension, a preprocessing tool, and an Aspect-Oriented Programming solution. This is due to the broad scope of the tool, which is intended as a Design by Contract solution both for Object-Oriented and Aspect-Oriented Programming.

2.5.3 Contract4J

Contract4J [Wampler06] [Wampler06b] is a library developed by Aspect Research Associates [AspectResearch07] that extends Java for Design by Contract by the use of annotations. Its implementation is non-intrusive through the use of AspectJ and Java's reflection mechanisms. This library is very flexible, in terms of the interpreter for the metadata (it is compatible with several open-source interpreters: JEXL [Apache06], Groovy [Laforge04] and JRuby [JRubyTeam07]). Note that this section refers to actual supported implementation – "Contract4J5", the author [Wampler06a] has experimented various approaches.

As shown in Listing 2.24, preconditions are specified in the `Pre` annotation and postconditions are specified in the `Post` annotation. Multiple conditions can be specified in the annotations, and they are evaluated from left to right. Invariants are specified in the `Invar` annotation. Contracts can be specified in classes and in interfaces. Each class/interface under contract must start with the `Contract` annotation. Finally, Contract4J allows the definition of *attribute invariants*, besides the regular *class invariants*.

Contract invariants are checked before and after the execution of public methods and constructors. Contract assertions are Java Boolean expressions, and therefore they are able to use most of the Java's expressiveness, including method calls.

Contract inheritance is not yet fully supported. Contract preconditions and postconditions can be inherited explicitly, by using an empty condition. Invariants however, are implicitly inherited, but cannot change.

In postconditions, there are the `$return` and `$old` keywords. The `$return` keyword refers to the return value of the method, enabling to assert on the expected result. The `$old` keyword refers to a primitive type or method return value before the execution of the method, but not to the old state of the object *per*

```
@Contract
@Invar("invariant")
class className
{

   @Pre("precondition")
   @Post("postcondition")
   returnType methodName(parameter list)
   {
      body
   }
}
```

Listing 2.24: Contract4J basic assertion syntax

se.

The type of conditions to validate can be configured at runtime using the API, or through configuration files (such as Java properties[15]). The configuration files can also be used to further define the library behaviour (e.g. information to be displayed in case of an exception, the interpreter in use, the classes that implement specific features, etc.).

In summary, Contract4J is the most powerful library available for Design by Contract in Java, and it is a very active open-source project.

2.5.4 OVal

OVal [Thomschke06] is a generic verification framework for Java, based in Java 1.5 annotations and AspectJ. Assertions can be written through annotations and/or XML configuration files.

OVal is based on three types of constraints: fields, parameters, and results. Parameter and result constraints work as preconditions and postconditions. However, unlike traditional implementations, field constraints are only evaluated when explicitly required by the method. As shown in Listing 2.25, field constraints are specified just before field declaration (line 4), parameter constraints are specified just before parameter declaration (line 7), and return type constraints are specified just before method declaration. Field constraints are evaluated before or after a certain method execution (lines 18 and 24).

Since it is based in annotations, it does not support contract inheritance.

In summary, OVal presents an alternative Aspect-Oriented Programming implementation for simple contracts. Nevertheless, it does not support a complete Design by Contract implementation, making it very difficult to express complex contracts, especially invariants.

```
1  @net.sf.oval.annotations.Guarded
2  class ClassName
3  {
4      field constraints
5      FieldType field;
6
7      ReturnType methodName(parameter constraints ParameterType parameterName) {
8          body
9      }
10
11     return type constraints
12     ReturnType methodName(ParameterType parameterName) {
13         body
14     }
15
16     @PreValidateThis
17     ReturnType methodName(ParameterType parameterName) {
18         body
19     }
20
21     @PostValidateThis
22     ReturnType methodName(ParameterType parameterName) {
23         body
24     }
25 }
```

Listing 2.25: OVal basic assertion syntax

15 A Java ".property" file is a semi-structured text file in which each line is in the form of "property = value".

2.5.5 SpringContracts

SpringContracts [Gleichmann06] [Gleichmann06a] is a Design by Contract solution with seamless integration in the Spring Framework [Interface06] [Walls05], a popular Java 2 Enterprise Edition framework. Just like in Contract4J, it is possible to configure the interpreter used by the library (such as Groovy [Laforge04] and OGNL [Davidson04]). However, there are several ways of using it.

First of all, contracts can be defined inside the classes/interfaces through annotations (Listing 2.26), or in external XML files (Listing 2.27). Second, contracts can be checked through two Aspect-Oriented Programming weaving options: Spring AOP (proxy based) or AspectJ. Either way, Aspect-Oriented Programming is not transparent; programmers must (at least) explicitly write an aspect in an external XML file.

Using the annotation form to write contracts, these are written in the Preconditon and Postcondition annotations before the method, while the Invariant annotation is before the class. These annotations can be written in either classes or interfaces.

Contract inheritance follows Liskov Substitution Principle. Contract invariant evaluation is dependent on the weaving strategy. For Spring AOP weaving, contract verification is only active when the call is performed by a client via proxy, method calls in the contracts will not be verified. For AspectJ weaving, all method calls are checked; infinite recursion is avoided through the writing of a specific pointcut to the effect.

There are several keywords available for contracts. arg<n> allows to access a keyword by index, instead of name, while args refers to the collection of arguments of a method. In postconditions, return

```
@Invariant(condition="invariant")
public interface InterfaceName
{
   @Precondition(condition="precondition")
   @Postcondition(condition="postcondition")
   public returnType methodName(parameter list);
}
```

Listing 2.26: SpringContracts basic assertion syntax using annotations

allows to access the return value of the method, while old allows access to the old state of an expression.

In terms of first order logic, the expected quantifiers and operators are provided. Table 2.7 summarizes this. Note that collection must be a class that implements the java.util.Collection interface, while expression is a Boolean expression that can use the element.

Configuration files for SpringContracts specify which classes and packages are to be contracted, which contract form is to be used (annotation or XML), which language is to be used (the default is Expression Language), and the assertion failure behaviour (the default is an Exception). This is similar to Contract4J.

Table 2.7: First-order logic support in SpringContracts

construct	syntax
universal quantifier	all (element : collection, expression)
existential quantifier	exist (element : collection, expression)
implication operator	expression1 => expression2

In summary, SpringContracts is an interesting tool for Spring Framework users due to its integration, but it is a complex tool, since it requires programmers to have a minimal knowledge in Java 2 Enterprise Edition and in AspectJ concepts.

```
<bean ... >
   ...
         <property name="invariants">
           ...
               <property name="condition"><value>invariant</value></property>
               <property name="message"><value>tag</value></property>
          ...
         </property>
         <property name="methodContracts">
           ...
               <constructor-arg><value>method signature</value></constructor-arg>
               <property name="preconditions">
                 ...
                      <property name="condition"><value>precondition</value></property>
                      <property name="message"><value>tag</value></property>
                 ...
               </property>
               ...
</bean>
```

Listing 2.27: SpringContracts basic assertion syntax using external XML files

2.6 Static verification: a step further

While most Design by Contract technologies existing today are based on runtime verification, there is some research work on static verification. Static checking aims to perform verification during compile-time. This allows covers a wider verification space since unlike, unit testing, it is not dependent on the coverage of the set of tests available. Of course, it is impossible to perform a complete verification[16], so the verification completeness obtained by static checking is dependent on the completeness of the specification, and the complexity of the tool used. A trade-off between completeness and usability must be defined by the tool authors.

In this section, some tools are presented, namely tools based on the JML specification language, and the Spec# language.

2.6.1 JML static verification tools

As mentioned earlier, although Java Modelling Language (JML) specifications represent a simple form of Design by Contract, wider subsets of Java Modelling Language can specify more elaborate conditions for verification. Next, tools for static checking of Java Modelling Language, as well as auxiliary tools, are presented.

ESC/Java2. ESC/Java was the first static checking tool for Java Modelling Language, developed at Compaq Research. After it was discontinued in 2002, a second version was developed, called ESC/Java2 [Chalin05]. Besides simple assertions, ESC/Java can perform other checks, such as dereferencing a null reference, indexing an array outside its bounds, and casting a reference to invalid type. Under the hood, ESC/Java2 uses the Simplify theorem prover [Detlefs03].

LOOP. The LOOP project at the University of Nijmegen started out as an exploration of the semantics of (sequential) Java. Currently, the tool translates Java Modelling Language-annotated Java code into proof obligations for the PVS interactive theorem prover [Owre96]. The difference between this and the other static checking tools is that Java and Java Modelling Language *formal semantics* are defined in PVS, instead of relying on *axiomatic semantics*.

JACK. The JACK tool [Burdy03] was initially developed at the Gemplus research lab (a *smartcard* manufacturer), but is currently at INRIA. This tool aims at providing a environment for Java and Java Card verification. Unlike the previous tools, the proof obligations used can generated from different

16 If it were possible, one could solve the *halting problem*, which was proven impossible to solve by Alan Turing.

theorem provers, and it is fully integrated in an Eclipse plug-in.

Other tools. In order to aid this static check tools, other auxiliary tools have been developed, in order to generate specifications. The Daikon and Houdini tools observe programs and detect emerging properties from these programs, which are inserted as Java Modelling Language annotations in the Java code. This alleviates the problem of writing Java Modelling Language specifications.

2.6.2 Spec#

Microsoft Research developed Spec# [Barnett04], an extension to the C# language [Microsoft05] [ECMA06] that enables Design by Contract support through runtime or static checking of C# programs. As such, Spec# adds some new features, which will be presented and exemplified in Listing 2.28.

Non-null types. Spec# authors acknowledge that many programming errors appear in the form of null references. In order to avoid this, Spec# features the possibility of declaring variables as non-null, using the exclamation mark ('!'). As such, class constructors need to be extended, in order to offer the default value facility available in some languages, such as C++. This is shown in line 3.

Exceptions. Unlike Java, C# does not have the distinction of *checked* and *unchecked exceptions*. Spec# adds this feature, through the throws keyword. Checked exceptions are exceptions that implement the ICheckedException interface. This is shown in line 8.

Preconditions. Preconditions are requires clauses, Boolean expressions, written before the method body. Multiple preconditions can be written. If checked at runtime, the behaviour for failed preconditions is to throw a RequiresViolationException, or a user defined exception, through the otherwise clause. In any case, the exception must be unchecked. This is shown in lines 9 and 10.

Postconditions. Postconditions are analogous to the preconditions, specified in ensures clauses. The default exception is EnsuresViolationException. An additional feature available is the old mechanism, which is used to evaluate the value of an expression before the execution of the method. This is shown in lines 11 and 12.

Exceptional postconditions. Exceptional postconditions are assertions that must be held in the result of a checked exception. These are written by combining an ensures statement with the throws statement. This is shown in line 8.

Frame conditions. Frame conditions (also know as frame rules), are constraints that indicate which attributes are changeable by a method. In Spec# this is obtained through the modifies clause. Since this clause cannot explicitly refer to private attributes, a wild card (^) is available to refer to all the attributes in the class. Frame conditions are not enforced at runtime. This is shown in line 13.

Inheritance. Contracts are inherited through class extension and interface implementation. Method overrides can add additional postconditions, and exceptional postconditions, as long as they are covered in the original throws clause. Frame conditions and preconditions cannot be changed. Due to the multiple inheritance for interfaces, some other restrictions apply, regarding the combination of contracts (that is, contracts for the same method in two interfaces): a class cannot implement two interfaces with requires clauses combined, and a class cannot implement two interface with combined different frame conditions. Nevertheless the slight differences with other solutions, Spec# guarantees behavioural subtyping (that is, the Liskov substitution principle rules).

Invariants. Spec# distinguishes two types of invariants: object invariants, and class invariants. *Class invariants* apply only to static fields, and are not inherited. *Object invariants* are what it is usually called class invariants in other solutions, that is, assertions about the class' state which are inherited into its subclasses. This is shown in line 5.

Expose statement. Spec# offers an expose statement to temporarily break the invariants of an object (or several objects). This is similar to the Nemerle language (see section 2.2.6). Invariants are verified at the end of the block. This is shown in lines 16 to 18.

Exceptions within contracts. Exceptions occurred within contracts are wrapped in a specific exception

```
1  class ClassName
2  {
3      FieldType regularField;
4      FieldType! nonNullField;
5      other fields declarations
6      invariant invariant;
7
8      ReturnType methodName(parameter list)
9      throws CheckedException ensures exceptional postcondition;
10     requires precondition;
11     requires precondition otherwise UserDefinedException;
12     ensures postcondition;
13     [ Conditional("DEBUG") ] ensures postcondition;
14     modifies frame conditions;
15     {
16         body
17         expose (object list)
18         {
19             modifications to the objects in the list that violate its invariant(s)
20         }
21         body
22     }
23 }
```

Listing 2.28: Spec# basic assertion syntax

for that use.

Conditional attribute. C# supports a metadata feature called *custom attributes*[17], which can be used in classes, methods and fields. The Spec# compiler uses the Conditional attribute to control which assertions are to be included in the release builds. This is shown in line 12.

Pure methods. Spec# only allows side-effect free contracts. As such, only pure methods can be called from the assertions. Pure methods are methods that do not change the state of the object and do not throw checked exceptions; or methods marked with the Pure attribute.

Assert call. Spec# allows putting assertions at any point inside a method, through the usual assert statement. Runtime check throws an exception if the assertion fails, while static checking will issue an error if it cannot prove the condition in that context.

First-order logic. Spec# has some logic statements, namely the exists (existential quantifier) and forall (universal quantifier).

Runtime checking. Runtime checking is obtained through the compiler, which produces standard CLR bytecode. The contracts are inlined in the code and tagged with attributes. Object invariants are inserted in a method, and called at the respective verification points.

Static checking. Static checking is performed by the Boogie checker. This transforms the code into an intermediate language, BoogiePL. This code is further transformed until being "fed" by Simplify, the theorem prover.

In conclusion, Spec# is an interesting solution, since it extends a mainstream language with Design by Contract and static checking, with integration in the Visual Studio platform. Some work is being performed in order to be able to write external contracts for code that cannot be changed, such as the .NET Base Class Library (BCL). Furthermore, Microsoft Research is working on the Singularity operating system [Fähndrich06], which is written in Sing#, an extension to Spec# with concurrency support, among other low-level mechanisms.

17 This is the equivalent to the annotation feature in Java.

2.7 Summary

The developer community interest on Design by Contract is undeniable. Nevertheless, the Design by Contract approach is rarely a priority for programming language developers, especially outside the academic world. Some recent programming languages aimed at the industry, such as D and Chrome, have come to address it, although only partially.

In the Java world however, the number of solutions is very large, and each solution offers a different subset of mechanisms, making it difficult to compare or choose one. Moreover, the semantics of the substitution principle are not unanimous. Solutions form three groups: language extensions, “traditional” approaches (source code preprocessing and bytecode instrumentation), and aspect-oriented solutions. Language extensions are not well accepted, since they break backwards-compatibility with standard compilers, and as such, “traditional” solutions are more popular. Aspect-oriented based solutions seem to be the next trend. Parallel to these types of solutions, there is also a dichotomy for assertion writing: some solutions follow Eiffel more closely, using executable code for writing assertions; other solutions use a specification language for expressing assertions.

Finally, it was shown that runtime verification is not the only path for Design by Contract. Static verifications solutions exist, both for Java and C#. Static verification can prove that a program is correct, instead of just checking that a program is not incorrect in certain test sets. However, static verification can only be used for a subset of the language, since some language mechanisms make programs not provable in general. Moreover, specifications for static verification are more extensive than runtime specifications, making it a significant investment in development time[18]. For general software development, runtime contracts are the most practical approach; for critical importance systems or components static contracts are a safer approach. Both can be used simultaneously, for maximum coverage.

Analysing all of these solutions provided an insight on the technologies, techniques, and problems facing when developing a Design by Contract solution. We will return to this solutions later in the document, when comparing them on a systematic way. In the next chapter we take a look at the state of the art of Aspect-Oriented Programming, so we can choose a technology to implement our solution.

18 It is not uncommon that the specification size is greater than the actual executable code.

Chapter 3

Aspect-Oriented Programming

In this chapter, Aspect-Oriented Programming is presented. We start by a brief introduction to the subject. Then we introduce the AspectJ language is introduced, as well as some alternative languages. Finally, we make some brief remarks on languages with native aspect-oriented support.

3.1 Introducing Aspect-Oriented Programming

A *concern* is any piece of interest or focus in a program [Dijkstra70]. Typically, concerns are synonymous to features or behaviours. *Separation of concerns* is the process of breaking software into distinct features that overlap in functionality as little as possible. This separation can be supported in various ways: by process, by notation, by organization, by language mechanism, among others. Usually this is achieved through modularity, encapsulation and information hiding. However, certain broadly scoped properties or features, such as persistence, security and logging, are difficult to modularize and their implementation is typically scattered along several different modules [Laddad03]. These concerns, known as *cross-cutting concerns*, hinder understandability, maintainability, and evolution.

Aspect-Oriented Software Development [Filman04] addresses cross-cutting concerns by providing means for their systematic identification, separation, representation and composition. Cross-utting concerns are encapsulated in separate modules, called *aspects*, and composition mechanisms are later used to weave them back with other core modules, at load time, compile time, or run-time. The best known Aspect-Oriented Programming implementation is the AspectJ language [Kiczales97] [Kiczales01], which is based on previous work, namely *composition filters* [Aksit92], *adaptive programming* [Gouda91] and *subject-oriented programming* [Harrison93].

3.2 AspectJ

AspectJ is an extension to Java that includes some new constructs, namely the aspect module. Initially developed at Xerox PARC, AspectJ is now an open-source project supported by the Eclipse Foundation [AspectJTeam03] [AspectJTeam05].

3.2.1 Join Points and Pointcuts

Join Points are points in the program where an aspect can insert behaviour. Join points can be *static*, if they refer to place in the code that can be determined at compile-time, or *dynamic* if they refer to a point in the code that can only be determined at run-time. Table 3.1 and Table 3.2 present the static and dynamic join points available in AspectJ[1], together with a brief explanation. *Type* is a data type (class or primitive), *Id* is a variable identifier, *Signature* is method/constructor signature (including annotations), *Type Patterns* are collections of types, *Expression* is a Java Boolean expression, and *Pointcut* is explained in the next paragraph. Type patterns can include wildcards ('*'), variable number of arguments ('..'), and subtypes ('+'). Table 3.1 provides some pointcut examples.

1 The AspectJ reference version is 1.5, based on Java 1.5.

Table 3.1: Static join points in AspectJ

join point	explanation
adviceexecution()	Execution on any advice.
call(Signature)	Call to any method or constructor at the call site matching the signature.
execution(Signature)	Execution to any method or constructor matching the signature.
get(Signature)	Reference to a field matching the signature.
handler(TypePattern)	Exception handler for Throwable type(s) matching the type pattern.
initialization(Signature)	Initialization of an object when the first constructor matches the signature, encompassing the return from the super constructor call to the return of the first-called constructor.
preinitialization(Signature)	Pre-initialization of an object when the first constructor matches the signature, encompassing the entry of the first-called constructor to the call to the super constructor
set(Signature)	Assignment to a field matching the signature.
staticinitialization(TypePattern)	Execution of a static initializer matching the type pattern.
within(TypePattern)	Join point from code defined in a type matching the type pattern.
withincode(Signature)	Join point from code defined in the method or constructor matching the signature.

Table 3.2: Dynamic join points in AspectJ

join point	explanation
args((Type | Id)+)	Join point when the arguments are instances of Types or the types of the Ids.
cflow(Pointcut)	Join point in the control flow of each join point matching the pointcut.
cflowbelow(Pointcut)	Join point below the control flow of each join point matching the pointcut.
if(Expression)	Join point when the expression is true.
target(Type | Id)	Join point when the target executing object is an instance of the type or id.
this(Type | Id)	Join point when the currently executing object is an instance of the type or id.

3.2.2 Advices: dynamic cross-cutting

Advices are the actual code that represents the behaviour to be inserted at the specified pointcut. The advice can be inserted before, after, or around the pointcut. The after advices can be namely specific, by after returns or after throws. The pointcut can be a previously declared *named pointcut*, or an *anonymous pointcut*. The advice body is the same as the body of a regular Java method, with the possibility of using three special variables that access the context of the advice: thisJoinPoint, thisJoinPointStaticPart, and thisEnclosingJoinPointStaticPart. Listing 3.2 gives some advice examples.

3.2.3 Static cross-cutting

The previous constructs refer to *dynamic cross-cutting*, that is, events that take place at runtime. However, AspectJ also allows *static cross-cutting*, the possibility to add behaviour at compile-time. Note that only static pointcuts are allowed for these features.

Inter-type declarations. Inter-type declarations are used to add code to an existing class without changing it. It is possible to add new attributes and methods. In a certain way this allows to implement an *ad-hoc* multiple inheritance mechanism.

Inheritance modification. It is possible to change the class/interface hierarchy. Using the declare parents construct, new extension/implementation relations can be added.

Compile-time warnings and errors. Using the declare warning and declare error constructs, it is

```
/**
 * Pointcut which captures method execution in J2SE system packages.
 */
pointcut systemPackageExecution():
    execution(java.* *(..)) || execution(javax.* *(..));

/**
 * Pointcut which captures all non-private, non-static method executions.
 */
pointcut nonStaticMethodExecution(Object object):
    target(object) && execution(!private !static * *.*(..));

/**
 * Pointcut which combines the previous two poincuts.
 */
pointcut nonStaticSystemPackageExecution(Object object):
    systemPackageExecution() && nonStaticMethodExecution(Object object);

/**
 * Pointcut which captures getters and setters executions.
 */
pointcut publicIntegerGetterCall():
    call(public int *.get*());

/**
 * Pointcut which captures all the situations within certain classes.
 */
pointcut insideClasses():
    withincode(org.acme.xpto.*) || within(org.acme.ypto.Foo+);

/**
 * Pointcut which captures the control flow of constructor executions.
 */
pointcut constructorFlowExecution():
    cflow(execution(public void *.new(..)));
```

Listing 3.1: AspectJ pointcut examples

possible to add new compile-time warnings and errors for certain pointcuts. This is useful for implementing code conventions within a development team. For example, one might want to restrain the programmers from using methods System.err.print() and System.out.print() and have an aspect to enforce that automatically.

Softened exceptions. This mechanism, through the declare soft construct, wraps checked exceptions at a certain pointcut in a org.aspectj.lang.SoftException, which is an unchecked exception. This way, the usual exception handling mechanism for Java is bypassed, allowing us not to declare in the throws clause those exceptions that were wrapped.

Advice precedence. Since several aspects can have overlapping advices, it is necessary to enforce the order of execution of the advices. The declare precedence construct allows to do just that, assigning an order for precedence among a set of aspects.

Annotation insertion. With the release of Java 1.5, annotations are having became very popular. The declare @annotation construct allows to add certain annotations to a determined pointcut.

3.2.4 The Aspect module

In AspectJ, an aspect is in terms of meta model [Han04], essentially an extension to a class, with some limitations.

```
before(Object object) : nonStaticMethodExecution(object)
{
    body
    Class objectClass = thisJoinPoint.getTarget().getClass();
    body
}

after() returning (Object result): staticMethodExecution()
{
    body
    Class staticClass = thisJoinPointStaticPart.getSourceLocation().getWithinType();
    body
}

int around(Object object): publicIntegerGetterCall()
{
    body
    int result = proceed(object);
    body
    return result + x;
}
```

Listing 3.2: AspectJ advice examples

The main limitation is that an aspect, unlike a class, cannot be explicitly instantiated. This is because AspectJ is a bi-dimensional aspect-oriented language, where classes are *oblivious* to aspects. An aspect may have a constructor to initialize any internal structures, but it cannot have any arguments. The instantiation takes place implicitly, at the start of the application, as default behaviour. Alternative instantiation types are: perthis, pertarget, percflow, percflowbelow and pertypewithin; in these cases, a pointcut must be specified.

An aspect can be declared as privileged, meaning it can break object encapsulation, in terms of member access.

It is worth noticing that an aspect can extend a class, but not otherwise, due to the class obliviousness. Aspects can also be abstract and extended by other aspects, but the semantics is a little different: both original and overloaded advices are inserted, bottom-up. Client/server relationships can exist among aspects, allowing them to share pointcuts. Listing 3.3 provides an example with aspects.

3.2.5 Annotation based development style

Various tools have taken advantage of the annotation metadata facility, since its introduction in Java 1.5. AspectJ 1.5 [AspectJTeam05a] allows an annotation-based style, in which the aspect part is written mostly in annotations. This allows us to write AspectJ code using standard Java code. The advantage of this style is to use "AspectJ unaware" tools, that is, tools that only known the Java BNF, in AspectJ code. The semantics of this style is the same as the "regular" style, meaning that the produced *bytecode* is the same in either style. However, the privileged keyword is not available in the annotation style. Listing 3.4 presents the Logging aspect example from Listing 3.3 written in this style.

3.2.6 Weaving

Weaving is the process of composing classes and aspects into end-user *bytecode*. With AspectJ, this process inputs *bytecode* and outputs (standard Java) *bytecode*. The weaving process can occur at one of three times: compile-time, post-compile time, and load-time.

The default process is *compile-time weaving*. Therefore, Java source code is weaved with AspectJ source or binary code, and the end-user *bytecode* is obtained. If the Java code requires the AspectJ code to compile (namely, if static cross-cutting is being used), then weaving must be performed this way.

```
// package visibility aspect
aspect Logging {

   // precedence declaration (static cross-cutting)
   declare precedence: Logging, Cache;

   // regular class atribute
   private Logger logger;

   // constructor (no arguments allowed!)
   Logging() { ... }

   // regular class setter and getter
   void setLogger(Logger logger) { ... }
   Logger getLogger() { ... }

   // pointcut declaration
   // (since it has package visibility, it can be shared with other aspects in the same package)
   pointcut publicDBLoadingExecution():
      execution(public * *.loadFromDB());

   // advice declaration
   before(): publicDBLoadingExecution() { log something... }
}

privileged aspect Cache {

   Hashtable <Integer, Object> cacheTable;
   ...

   // advice declaration (mixes a Logging pointcut with an anonymous pointcut)
   Object around(): Logging.publicDBLoadingExecution() && execution(private * *.loadFromDb())
   { use cache... }
}
```

Listing 3.3: AspectJ aspect example

Post-compile/binary weaving is used when the Java source code is not available. Thus, given Java binary code and AspectJ source or binary code, the end-user *bytecode* is obtained. This is useful for using third-party components.

Load-time weaving is used to perform binary weaving when the class loader is loading the Java code to be weaved. This requires that the Java Virtual Machine have a *weaving class loader* available. This is useful for inserting behaviour without stopping and recompiling an application.

3.2.7 Tool support

AspectJ runtime. Since weaved *bytecode* is standard Java *bytecode*, AspectJ applications can be run on a standard Java Virtual Machine, such as the `java` command provided by Sun Microsystems' Java Runtime Environment. The only requirement is that the runtime JAR [Sommerer98] is to be included in the *classpath*.

AspectJ tools. The AspectJ tools include: `ajc`, the compiler and weaver (analogous to the `javac` command); `ajdoc`, the documentation support (analogous to the `javadoc` command); `ajbrowser`, a GUI for cross-cutting code viewing; Ant [Apache07] support; and load-time weaving support.

AspectJ Development Tools (AJDT). Since AspectJ is being developed by the Eclipse Foundation, there is a great synergy between AspectJ and the Eclipse IDE development. The most remarkable result of this synergy are the AspectJ Development Tools [Colyer04], an Eclipse plug-in that adds a seamless

```
// package visibility aspect
@Aspect
@DeclarePrecedence("Logging,Cache")
class Logging {
    // regular class atribute
    private Logger logger;

    // constructor (no arguments allowed!)
    Logging() { ... }

    // regular class setter and getter
    void setLogger(Logger logger) { ... }
    Logger getLogger() { ... }

    // pointcut declaration (since it has package visibility, it can be shared with other aspects in the same
package)
    @Pointcut("execution(public * *.loadFromDB())")
    void publicDBLoadingExecution() {}

    // advice declaration
    @Before("publicDBLoadingExecution()")
    void beforePublicDBLoadingExecution() { log something... }
}
```

Listing 3.4: AspectJ code in annotation-based style

support for AspectJ development into Eclipse.

3.3 Other Solutions

Several implementations of Aspect-Oriented Programming exist. Typically, they are extensions for existing programming languages, but other solutions include libraries and middleware frameworks. This section presents a select set of such implementations for Java. More comprehensive surveys have already been made, such as in the context of the AOSD-Europe project [Brichau05] [Dinkelaker06]. IBM's AOP@Work [Kersten05] also provided a comparison of some popular solutions

3.3.1 JBoss AOP

JBoss [Fleury06] is a Java 2 Enterprise Edition-based application server, currently developed by Red Hat Inc. JBoss AOP was released in 2004.

JBoss AOP [RedHat07] [RedHat07a] aspects are plain Java objects with methods that take an Invocation object parameter, as shown in Listing 3.5. An invocation object is an instance of an invocation class from Table 3.3, which contains the necessary runtime context information. Methods in one of these classes work as *advices*. *Pointcuts* are written in external XML files (as shown in Listing 3.6), or using annotations. The *pointcut* expression language is similar to AspectJ. Regarding pointcut designators, most of them are equal to AspectJ: execution(), construction(), get(), set(), call(), within(), and withincode(); some are generalizations of the previous: field() and all(); and some are not present in AspectJ[2]: has() and hasField().

2 Actually, AspectJ 1.5 does support these pointcut designators (called "hasMethod()" and "hasField()"), but as experimental and undocumented features which are activated through a compiler flag.

```
class AspectName {

   Object methodName(Invocation object) throws Throwable {
      ...
   }
}
```

Listing 3.5: JBoss aspect convention

Table 3.3: JBoss invocation classes

MethodExecution
ConstructorExecution
FieldInvocation
FieldReadInvocation
FieldWriteInvocation
MethodCalledByMethod
MethodCalledByConstructor
ConstructorCalledByMethod
ConstructorCalledByConstructor

```
<aop>
   <aspect class=”MyAspect” />
   <bind pointcut=”execution(public void MyClass->myMethod(int parameter))”>
      <advice name=”myAdvice” aspect=”MyAspect” />
   </bind>
</aop>
```

Listing 3.6: JBoss XML example

In terms of static cross-cutting, JBoss supports *introductions*, which just like AspectJ inter-type declarations, allow the use of *ad-hoc mixins*[3]. Introductions are obtained by writing an aspect class with the introduced code, and an XML file with its binding. Listing 3.7 provides an example.

```
<introduction class="OriginalClass">
   <mixin>
      <interfaces>NewInterface</interfaces>
      <class>MixinClass</class>
      <construction>new MixinClass(this)</construction>
   </mixin>
</introduction>

public class MixinClass implements NewInterface {
   OriginalClass original;

   public MixinClass(OriginalClass original) {
      this.original = original;
   }

   // implementation of the methods defined in NewInterface...
}
```

Listing 3.7: JBoss introduction example

3 A mixin [Bracha90] is an inheritance relationship without the semantics of subtyping, but simply with the intent of code reuse.

Other features of JBoss AOP include: *hot deployment*, the ability to deploy/undeploy aspects at runtime; *per instance Aspect-Oriented Programming*, the ability to instantiate an aspect for every object in the pointcut. Also, a plug-in for Eclipse IDE is available.

3.3.2 Spring AOP

The Spring Framework [Walls05] [Johnson07] is a Java 2 Enterprise Edition framework, developed by Interface21. Spring AOP was released in 2004.

Spring 2.0 AOP allows aspect writing using a XML-based or annotation-based style. Using the annotation style, aspect writing and using is identical to the AspectJ annotation style syntax. However, only some of the AspectJ pointcut designators are available: execution, within, this, target, args, @target, @args, @within, and @annotation (that is, matching a specific annotation). Using the XML style, the same expressiveness is available, but aspects are written as regular classes, and the binding of pointcuts to advices is written in an XML file.

Also, Spring AOP provides two Aspect-Oriented Programming implementations: the Spring AOP (described in the previous paragraph), and the AspectJ AOP, which uses AspectJ for weaving. Using the Spring AOP, then one is using *proxy-based:* each client call to a method is intercepted by a proxy object. In practise this means that a method call inside another method will not be intercepted, and therefore not advised, since it is not in the same context of the original client call. Using the AspectJ AOP, the full AspectJ expressiveness is available, included the AspectJ language syntax.

3.3.3 AspectWerkz

AspectWerkz is an Aspect-Oriented Programming language, released in 2002 and developed by BEA Systems.

Aspect code is regular Java code with annotations, very similar to the described previously in section 3.2.5. AspectWerkz is an Aspect-Oriented Programming language, but also an *aspect container* [Bonér04], i.e., an architecture that allows the coexistence of other aspect technologies. In this architecture, the AspectWerkz container is responsible for handling *pointcut* matching, aspect life-cycle, deployment management; each *aspect model extension* is responsible for handling the aspect sources and metadata. Custom models can be added by implementing an AspectModel interface, but the distribution already includes support for AspectJ, Spring AOP, as well as AOP Alliance[4] interfaces. In 2005, the development of new releases for AspectWerkz was discontinued, although existing versions are maintained, as its authors decided to merge the project with the AspectJ project, which resulted in AspectJ 1.5's support for annotation-style.

3.3.4 FuseJ

Although there are several technologies for introducing Aspect-Oriented Programming in Component-Based Software Development, most of them require language extensions or frameworks for modularizing cross-cutting concerns, typically using an “aspect” module. FuseJ [Suvée05] [Suvée06] authors claim that there is no need for the aspect module. More, they claim that using an aspect compromises the evolution of the cross-cutting concern to a component, and vice-versa. As such, they propose that cross-cutting concerns are implemented as components, and that the composition itself can be performed afterwards, either as a regular component or as an aspect.

To do so, they introduce some new concepts, namely *service specification* and *configuration*. A service specification is a set of operations that a component will provide (the provides clause) and can expect to be available by the environment where it will be deployed (the expects clause). The operations are expressed in regular Java interfaces, as shown in Listing 3.8. The configuration construct enables the component composition, through the use of *linklets*. A configuration configures a set of components

4 AOP Alliance is a project that aims to ensure interoperability between Java/J2EE AOP implementations [Pawlak03].

```
interface IFoo {
    int operationA();
    char operationB();
}

interface IBar {
    float operation1();
    String operation2();
}

service MyService {
    provides IFoo;
    expects IBar;
}
```

Listing 3.8: FuseJ service specification example

```
configuration name configures (component | service)+ as service {
    (linklet linkName {
        execute | expose : (componentOperation | serviceOperation)+
        for | before | after | around | as : (componentOperation | serviceOperation)+
        (where : (parameterMapping)+)?
        (when : (componentOperation | serviceOperation)+)?
    })+
}
```

Listing 3.9: General structure of a FuseJ configuration entity

and/or services to a specific service. Each *linklet* is composed by: target role, the set of operations to execute; source role, the set of operations that act as trigger; property mapping, the set of property mappings among source, target, and external operations; and condition specification, the set of preconditions among source, target or external operations. This is shown in detail in Listing 3.9.

3.3.5 Hyper/J

Most aspect-oriented approaches are bi-dimensional, in the sense that they have a *dominant decomposition*, usually object-oriented, and an aspect-oriented decomposition. There is, however, an alternative approach: multidimensional separation of concerns [Tarr99]. According to this, software is divided in *hyperslices*, each of it can be a requirements specification, a design model, or source code. A hyperslice encompasses a set of *units* (e.g. modules) that represent a concern dimension. In order to the system to be build, hyperslices must be composed together, since they overlap. A set of hyperslices with a composition rule that enables the building of a new hyperslice, is called a *hypermodule*. Hypermodules can be further composed with other hypermodules or hyperslices, in order to produce a new hypermodule.

Hyper/J [Ossher99] [Tarr00] is a code implementation of Multi-Dimensional Separation of Concerns, that is, a multidimensional aspect-oriented implementation for Java. Units can be packages, classes, interfaces, and members. The composition is achieved through a specification file, which includes: hyperspace specification, concern mapping, and hypermodule specification, The *hyperspace specification* defines a set of hyperspaces, each with a name and a set of units which belong to it. The *concern mapping* defines a set of dimensions and concerns, and how the units address the dimensions and concerns. The *hypermodule specification* defines a set of hypermodules. Each hypermodule is a particular integration of units which belong to specific selection of concerns.

There are several composition operators (as shown in Table 3.4), and three general strategies. These strategies define the default composition to apply: `mergeByName`, whereby units with the same name in different hyperslices are connected by a `Merge` relationship (and therefore the units are integrated into one); nonCorrespondingMerge, whereby units with the same name in different hyperslices are not connected in any way; and overrideByName, whereby units with the same name in different hyperslices are to be connected by an Override relationship (the last unit overrides the preceding, the order is given by the hyperslice declaration order).

Table 3.4: Hyper/J composition functions

composition function	general syntax
Equate	equateRelationship ::= equate unitKind unitName [, unitName]* ; unitKind ::= class | interface | operation | action | field
Order	orderRelationship ::= order unitKind unitName [, unitName]* (before | after) unitKind unitName [, unitName]*; unitKind ::= hyperslice | class | interface | operation | action
Rename	renameRelationship ::= rename unitKind unitName to newUnitName; unitKind ::= class | interface | operation | action | field
Merge	mergeRelationship ::= merge unitKind unitName [, unitName]* ; unitKind ::= class | interface | operation | action | field
NoMerge	noMergeRelationship ::= noMerge unitKind unitName [, unitName]* ; unitKind ::= class | interface | operation | action | field
Override	overrideRelationship ::= override unitKind unitName [, unitName]* with unitKind unitName; unitKind ::= class | interface | operation | action
Match	*regular expression*
Bracket	bracketRelationship ::= bracket [classMatchPattern .] operationMatchPattern [from unitKind unitName [, unitName]*] [with] [before fullyQualifiedMethodName,] [after fullyQualifiedMethodName,] unitKind ::= hyperslice | class | operation | action
Summary	summaryFunctionRelationship ::= set summary function for unitType unitName to summaryFunction; summaryFunction ::= external unitName | unitType unitName unitType ::= action | operation

Hyper/J evolved into the Concern-Manipulation Environment project of the Eclipse Foundation, but has been discontinued since 2005.

3.3.6 CaesarJ

CaesarJ [Mezini03] [Aracic06] is an aspect-oriented language that attempts to address issues in aspect-oriented and component-based programming, by unifying the aspect and class modules.

CaesarJ introduces a new module, the *virtual class*, named cclass to distinguish from regular Java classes. The virtual class provides a larger scale unit of modularity than a regular class, in the sense that allows to group classes. Inner classes of a cclass are inherited through subtyping, allowing a *family polymorphism*, i.e., the ability to extend a particular class (or classes) within a family, while maintaining the remaining classes. Also, CaesarJ relaxes the Java rule that a class containing an abstract method must be declared as abstract: a method can be abstract as long as at least one of the enclosing

```
cclass FigureDisplay
{
    cclass Figure
    { ... }
    abstract cclass FigureElement
    { ... }
    cclass Point extends FigureElement
    { ... }
    cclass Line extends FigureElement
    { ... }
}
```

```
cclass ColouredFigureDisplay extends FigureDisplay
{ ... }

cclass BezierFigureDisplay extends FigureDisplay
{ ... }

cclass ColouredBezierFigureDisplay extends
    ColouredFigureDisplay & BezierFigureDisplay
{ ... }
```

Listing 3.10: CaesarJ virtual class example (left) and mixin example (right)

classes is abstract. The left side of Listing 3.10 shows an example of a virtual class, using the "classic" figure editor example (e.g. [Kiczales01]).

Another feature that further extends Object-Oriented Programming in Java is the *mixin-based* class composition. In CaesarJ, mixins can be used in simple classes and family classes, since they are propagated through inner classes. The right side of Listing 3.10 shows an example of a mixin example:

```
public class FigureDisplayAction implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        SpecialFigureDisplay display = new SpecialFigureDisplay();
        ...
        deploy display;
    }
}
public class FigureDisplayView extends JComponent {
    FigureDisplay display;
    ...
    public void close() {
        ...
        undeploy display;
    }
}
```

Listing 3.11: CaesarJ local deployment example

a display class than inherits behaviour from a coloured display and from Bézier curve[5] display class.

In terms of actual Aspect-Oriented Programming, CaesarJ uses the same pointcut language that AspectJ[6], but adds some new concepts. First of all, CaesarJ "aspects" are implemented in the same module as family classes: the `cclass`. In terms of static cross-cutting, it is possible to perform *bindings*, that is, class families can adapt classes from other families. Classes that perform bindings are called *wrapper* classes (while the adapted classes are called *wrappees*).

In terms of dynamic cross-cutting, CaesarJ differs from AspectJ in the sense that classes are not oblivious to aspects. "Aspects" can be instantiated, accessed, and *deployed* (a deployed aspect is an active aspect) by classes. There are three main types of deployment: *local deployment*, where aspects can be activated/deactivated through the `deploy`/`undeploy` keywords (see Listing 3.11); *thread-based deployment*, where an aspect instance is deployed for each thread, using the `deploy(..) {}` block (see the left side of Listing 3.12); and *static deployment*, where there is only a single instance of the aspect, and it is deployed during the life cycle of the program, this is achieved by either declaring the `cclass` as deployed, or by applying the deployed modifier to field declaration (see the right side of Listing 3.12). The static deployment is equivalent to a *singleton* AspectJ aspect. A CaesarJ aspect is garbage collected

5 A Bézier curve is a parametric curve, widely used in computer graphics to model smooth curves.
6 Actually, under the hood CaesarJ uses the AspectJ weaver.

```
public cclass LineFigureDisplayLogging {
   pointcut lineSetterExecution() : ... ;
   void around() : lineSetterExecution()
   {
      ...
      SetterLogging logging = new SetterLogging();
      deploy (logging)
      {
         proceed();
      }
   }
}
```

```
deployed public cclass FigureDisplayLogging
{
   ...
}
```

Listing 3.12: CaesarJ thread-based deployment example (left) and static deployment example (right)

when there is no reference to it, nor it is under deployment.

CaesarJ includes some support in terms of distributed programming. Using *remote deployment*, it is possible to deploy aspects through remote processes, using a specific API, which is built on top of the Java Remote Method Invocation (RMI) infrastructure. Using *distributed control flow deployment* is possible to distinguish the among different clients' control flow.

3.4 Native aspect-oriented support

As shown throughout this chapter, existing implementations of Aspect-Oriented Programming are provided as libraries or language extensions, and no programming language offers native support. However, programming languages have introduced new forms of separation of concerns throughout computer science history, and as such, some features are difficult to classify as aspect-oriented or not. An example of such is the C# programming language, with the concepts of *partial classes* and *static constructors*. Partial classes are classes which are written in multiple files, somewhat comparable to intertype declarations (for a single class). Static constructors are class constructors which are executed implicitly at the application start, somewhat like AspectJ's aspect constructors and CaesarJ's singleton constructors.

3.5 Summary

AspectJ was one of the first Aspect-Oriented Programming implementations, and has a significant user base, making it the benchmark language for Aspect-Oriented Programming. Furthermore, it has a continuous development and tool support, keeping it at the same level of other implementations. As such, it is the natural choice for the work described in this dissertation.

Implementations such as JBoss AOP and Spring AOP are focused in Java 2 Enterprise Edition applications, and aim at offering a simplified Aspect-Oriented Programming approch, by using a subset of AspectJ features. As such, they are not appropriate for this thesis. Component-based implementations such as FuseJ, although more complete, suffer similar issues, as they add nothing relevant for the proposed library.

There are some Aspect-Oriented Programming implementations though, that add relevant features, compared to AspectJ. Hyper/J and CaesarJ are such cases. Hyper/J offers a different approach, the Multi-Dimensional Separation of Concerns, making it on one hand a more flexible solution in terms of composition mechanism, but on the other hand involving more code writing, with more complexity, when compared with AspectJ [Chavez01]. The lack of development or tool support from IBM since 2003 makes it an uninteresting solution, from the engineering point of view (namely for the absence of Java 1.5 support). CaesarJ however, adds interesting new concepts, both in terms of Object-Oriented Programming and Aspect-Oriented Programming. Compared to AspectJ, the flexibility of deployment can simplify implementation, as well as improve performance. The pointcut language though, as in most

Aspect-Oriented Programming implementations, is the same as AspectJ.

In conclusion, both AspectJ and CaesarJ are good candidates for an implementation such as the proposed infrastructure of this thesis. In the end, the maturity, user base, and existing documentation of AspectJ, surpassed the new features of CaesarJ, for choice of implementation language. However, it would be an interesting experience to produce an equivalent implementation as proposed infrastructure in this thesis, using the CaesarJ language.

Having analysed these Aspect-Oriented Programming technologies allowed us to gain an insight on the strengths and limitations of these technologies, which allowed us to choose the one that better suits our purposes. In the next chapter, we head straight to the design and implementation of the prototype library.

Chapter 4
Library Design and Implementation

In this chapter, the Design by Contract for Java (DbC4J) library design and implementation are explained. First, the prototype objectives are presented. Second, the design decisions regarding the prototype construction are discussed. Next, the available features usage and implementation is shown. Finally, we will critically discuss our accomplishments.

In this chapter, UML class diagrams [OMG05a] are used, as well as *aspect diagrams*, a class diagram extension using the UML-based Aspect Engineering notation [Garcia04].

The prototype was implemented in the Eclipse IDE 3.2 on top of Java JDK 1.5, using the AspectJ Development Tools (AJDT) 1.5 plug-in.

4.1 Requirements

The main goal of our work is to produce a prototype to support the Design by Contract approach in Java applications with the following functional and non-functional requirements:

1. The prototype should be in the form of a library, and should be simple to use by an "average" Java programmer, that is, one which is not familiar with AspectJ or AOP in general.
2. The library should support the core features present in the Eiffel language: preconditions, postconditions, class invariants, contract inheritance, and the "result" and "old" special variables.
3. The contracts are to be written in standard Java code, although the library is implemented in an aspect-oriented language.
4. The library should be pluggable, meaning that an application written with the library should compile in the absence of the library, and display the same functional behaviour.
5. Finally, a mechanism for restraining contract verification should be available, with some of level of granularity, in terms of modules and/or assertion types.

4.2 Design decisions

In this section, the design decisions performed through the course of this dissertation are explained.

4.2.1 Executable code

The decision with the most impact was the use of executable code for expressing contracts. As shown previously, most existing *ad-hoc* Design by Contract solutions for Java choose to use some kind of metadata to express contracts. Choosing so allows a greater flexibility, since it is possible to control the language used for specification, and therefore its expressiveness. On the other hand one has to either implement an interpreter or use a third-party one, which restrains the evolution of the library in relation to subsequent releases of the Java language. Executable code in plain Java is more resilient to language changes, although it remains limited to the Java expressiveness.

An issue raised by using an interpreter for metadata contract specifications is that it is not transparent to the existent tools, i.e., we cannot compile, debug, trace, document, refactor, or profile its contract specifications with standard Java/AspectJ tools. In order to do so, we would have to develop specific tools, which would also suffer the evolution problem explained in the previous paragraph. Using executable code it is possible to take advantage of existing tools immediately, while being ready for upcoming tools.

Finally, there is the cognitive issue. In spite of the technical advantages of using a different language, it implies that the developers (especially the programmers) must learn a new language. Even if the learning curve is small (e.g. using the Groovy script language [Laforge04]), the programmer was to think in two languages in implementation.

In summary, while using Java executable code limits the expressiveness of the contracts, since we cannot syntactically extend the programming language; the development effort is reduced, and the adaptability to changes in Java is guaranteed.

4.2.2 Aspect-Oriented Programming

Another decision with a major impact is the choice of using Aspect-Oriented Programming for implementing the contract verification. As shown throughout Chapter 2, other *generative programming* [Czarnecki00] approaches exist, such as source code preprocessing/transformation and *bytecode* manipulation. Indeed, this was our starting point, here rationalized as a result of analysis.

Source code preprocessing offers some advantages, since it does change the compile or execution regular behaviour, as the compiled code is regular Java. However, the source code transformation step is not transparent, i.e., there is no traceability from the original code to the transformed code, unless a special development environment exists. Also, there is no inverse transformation, unless an inversion process is available, but such process would require extra metadata to support it. The development of such mechanisms would require a significant effort, and would be vulnerable to language evolution.

Bytecode manipulation addresses some issues of source code preprocessing, namely in terms of language evolution, since historically, Sun Microsystems has a politics against virtual machine major changes[1]. Nevertheless, this technique changes the compilation and the execution processes, through the use of Java reification mechanisms. This implies a detailed knowledge of the Java VM, even when following a simplified approach based on an API, such as the Byte Code Engineering Library (BCEL) [Apache06a].

To summarize, source code preprocessing is too restrictive and bytecode manipulation is too complex. The Aspect-Oriented Programming technique we opt for provides a transparent implementation, with no need to re-implement support tools, which means that it can take advantage of performance improvements on existing tools (e.g. the AspectJ weaver).

4.2.3 Programming language

Having decided to use Aspect-Oriented Programming, we must choose the language. Since the release of AspectJ, several Aspect-Oriented Programming implementations have been developed, as described in Chapter 3. We also showed, that most solutions try to use a subset of AspectJ's expressiveness, improving deployment and specification for a particular approach, typically Java 2 Enterprise Edition application servers or component-based architectures. Since the objective of this library is to be general purpose, it is necessary to a general purpose language, of which the only “real” alternatives to AspectJ are Hyper/J and CaesarJ. Hyper/J though, presents a major paradigm shift, which is unsupported and discontinued. CaesarJ is the only feasible alternative to AspectJ.

The reason for choosing AspectJ over CaesarJ was the maturity of the AspectJ tools, the number of users, as well as the existing bibliography and documentation. Notwithstanding, it would be interesting to repeat this experience using CaesarJ.

1 For example, in Java 1.5 the generic type mechanism was introduced, but the Java virtual machine specification does not directly support this concept, which limits its usability in some situations, the so-called type “erasure” [Bracha04] [Langer07].

4.2.4 “result” and “old” constructs

One of the major challenges of the library was the introduction of the “result” and “old” constructs. Since specifications are written in executable code, these constructs cannot be introduced as keywords, as in the OCL syntax [OMG05]. Two alternatives were considered, using method parameters, and the other using specific method calls.

For the “result” construct, the choice was to use it as the first argument of postconditions, except when dealing with a void method.

For the “old” construct, a similar solution could have been used. However, adding another special argument to the signature of the postcondition method would be cumbersome, as a wrongly written signature of an assertion will result in it being ignored at runtime. Instead, the choice was to use specific method calls. This simplifies the assertion writing, as it reduces the possibilities of programming errors, and it also as improves performance. On the other hand, it raises the issue of coupling with the static class, and it lacks type safety: the old() method returns an Object, which must be cast into the specific type. We studied other implementations, namely of using inter-type declarations in order to add a method or a field to the class under contract, but we deemed them too intrusive.

4.2.5 Boolean methods

Java now has an assert statement, which given a Boolean expression as an argument, throws an AssertionError when the expression evaluates false. It is possible to enable and disable the Java assertion checking on a per class basis.

One approach for the library implementation would be to use the assert statement in the (library) assertions, which would then be void methods. In terms of usage, there is no significant advantage on either approach. Technically, there is also no difference between using an AssertionError or RuntimeException (such as ContractException), since both extend Throwable, and are not required to be declared in the throws clause, but can be caught in a try/catch block. Since by using the assert statement, one has to rely on the assertion disabling mechanisms of Java, it would be confusing to mix this with the developed disabling mechanisms that depend on the type of the assertion. In terms of usage, the assert statement can in some situations make contract code more readable, such as with “artificial” contracts or contracts that deal with several conditions, as illustrated in Listing 4.1.

```
public class Point4D {

   public void setX(int newX) { /* ... */}
   public boolean preSetX(int newX) {
      ContractMemory.remember();
      return true;
   }
   public boolean postSetX(int newX) {
      Point4D old = (Point4D)
ContractMemory.old();
      return
         this.getY() == old.getY() &&
         this.getZ() == old.getY() &&
         this.getT() == old.getT();
   }
}
```

```
public class Point4D {

   public void setX(int newX) { /* ... */}
   public void preSetX(int newX) {
      ContractMemory.remember();
   }
   public void postSetX(int newX) {
      Point4D old = (Point4D)
ContractMemory.old();
      assert this.getY() == old.getY();
      assert this.getZ() == old.getY();
      assert this.getT() == old.getT();
   }
}
```

Listing 4.1: DbC4J alternative assertion implementation example

4.2.6 Exceptions

Assertion failures yielding exceptions are a reasonable solution for Java programming. Using another

Throwable, such as an Error subclass, was dismissed, since assuming that all assertion failures cannot be handled is too restrictive (e.g. an authentication failure may be treated by requesting the authentication input to the user). Using a logging or a console user output mechanism does not offer enough “psychological pressure”[2].

Some Design by Contract solutions offer the possibility of using a user-defined behaviour (or at least a user-defined exception) in case of an assertion failure, and we considered that possibility too. However, such alternative implementations had some drawbacks: that would be syntactically difficult to implement and use; and they offered no additional expressiveness, since the same behaviour can be achieved using a try/catch block. Also, Java's exception hierarchy (see Figure 4.1), we can take

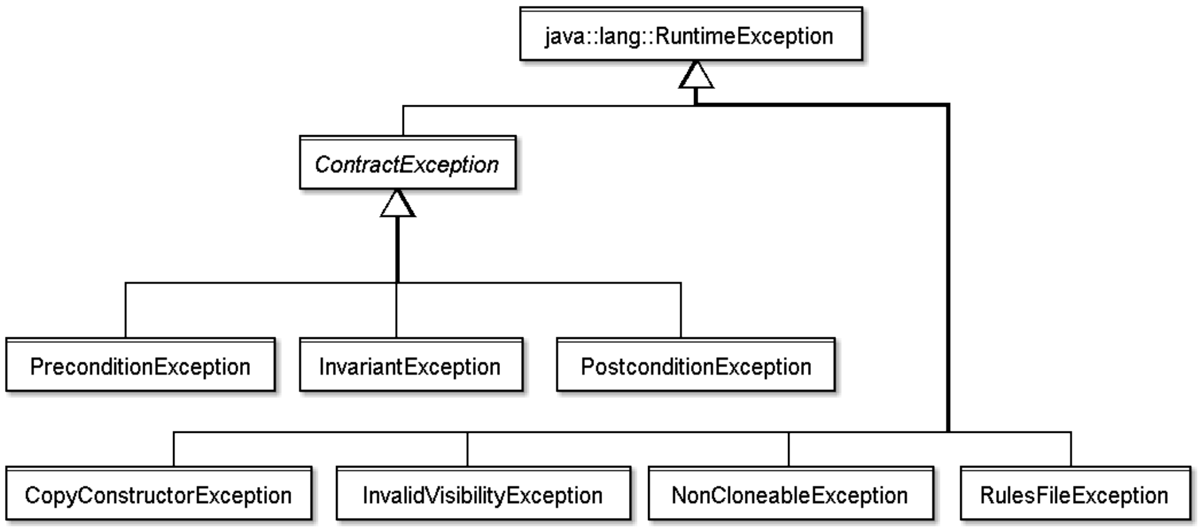

Figure 4.1: Exception class diagram

advantage of polymorphism to simplify exception handling.

4.3 Features

4.3.1 Preconditions, Postconditions and Invariants

Usage. Preconditions and postconditions are methods with the same name as the original method, prefixed with 'pre' and 'post', using *lower camel case*. These methods return a Boolean value, either the primitive type boolean or the wrapper class Boolean). The arguments of the assertion can be the same as the original method, or an ordered subset of it. The names are irrelevant, but as a convention we recommend using the same names as in the original method. In the case of the postconditions for non-void methods, these arguments must be preceded by a special argument, the result value of the method under contract. Listing 4.2 presents an example.

Invariants are methods named 'invariant'. These methods return a Boolean value, and take no arguments. Invariants will be checked before and after any method execution. In constructors, however, they are checked afterwards only, because invariants are meant to check the internal state of the class, which is set by the constructor. Listing 4.3 presents an example.

Methods under contract must be non-private. Static methods are evaluated by static contracts, and non-static methods are evaluated by non-static contracts.

Overloading of methods and contracts is allowed, as long as the contracts signatures can be unambiguously assigned to one method only.

Failed checks will produce an exception, as expected in regular Java programming. Depending on the

2 An average programmer will be more inclined to fix a problem that terminates the program with an exception, than a problem which merely outputs a message (and may not affect this particular execution of the program).

```
public int factorial(int n) {
    // ...
}
public boolean preFactorial(int n) {
    return n >= 0;
}
public boolean postFactorial(int result, int n) {
    if (n == 0)
        return result == 0;
    else
        return result >= 1;
}
```

Listing 4.2: Precondition and postcondition example under DbC4J

assertion check failure, a correspondent ContractException will be thrown: PreconditionException, PostconditionException, and InvariantException. These exceptions extend Java's RuntimeException, so that they do not need to be handled through a try/catch block. Furthermore, other exceptions exist to report specific anomalies in the contract writing, as shown in Figure 4.1.

Implementation. We use three types of pointcuts to implement contract assertions: constructor

```
public class Simple2DCoordinates {
    private int latitude; // degrees
    private int longitude; // degrees

    // ...

    public boolean invariant() {
        return
            (getLatitude() >= -90 && getLatitude() <= 90)
            &&
            (getLongitude() >= -180 && getLongitude() <= 180);
    }
}
```

Listing 4.3: Class invariant example under DbC4J

executions, static method executions, and non-static method executions. Each pointcut has two advices: before and after return. Exceptional terminations of a method or constructor are not evaluated. We need those three types of pointcuts because in some situations class invariants are not checked. For non-static methods, invariants are evaluated before preconditions and after postconditions, following the Eiffel specification. Before constructor execution the invariant is not yet set, so it can not be evaluated. Finally, static methods have no object instance, and so invariants do not apply. Figure 4.2 shows the class diagram with the interaction among DesignByContract, DesignByContractAspect, and the contracted classes.

The assertion methods are found using Java's reflection mechanisms, available in the java.lang.reflect package. The algorithm first tries to find an assertion with no arguments With the same name as the method and the required prefix, pre or post, and then keeps adding an argument until a method is found (if there is one at all). This algorithm is fragile in the presence of method overloading, because ambiguous contract signatures may be misused. Listing 4.4 shows an example of ambiguous contracts. For improving performance the assertion method is found by "trial and error", that is, by trying to execute the method without checking its existence[3].

It is also worth noticing that the contracted methods pointcuts throw several exceptions, namely:

- Method executions inside the library's package, to avoid infinite recursion

3 An alternative would be making a linear search through the class' available methods.

```
Class foo {
    /* original method */
    public void setX(int x) { ... }

    /* overloaded method */
    public void setX(String x) { ... }

    /* precondition for original method - ambiguous */
    private boolean preSetX() { ... }

    /* precondition for overloaded method - never executed */
    private boolean preSetX(String x) { ... }
}
```

Listing 4.4: Ambiguous contracts under DbC4J

(excludedSituation()).

- Method executions inside the Java 2 Standard Edition system packages, to improve performance – one cannot add contracts to these classes, not even through static cross-cutting (systemPackageExecution()).
- Assertion methods execution, since we do not want to contract contracts (assertionExecution()).
- The main method, since it is called by the system class loader implicitly (excludedSituation()).
- Methods inherited or overridden from the Object class, since these methods are inherent to the Java language usage – they are called often and have no need for contracts (objectMethodExecution()).

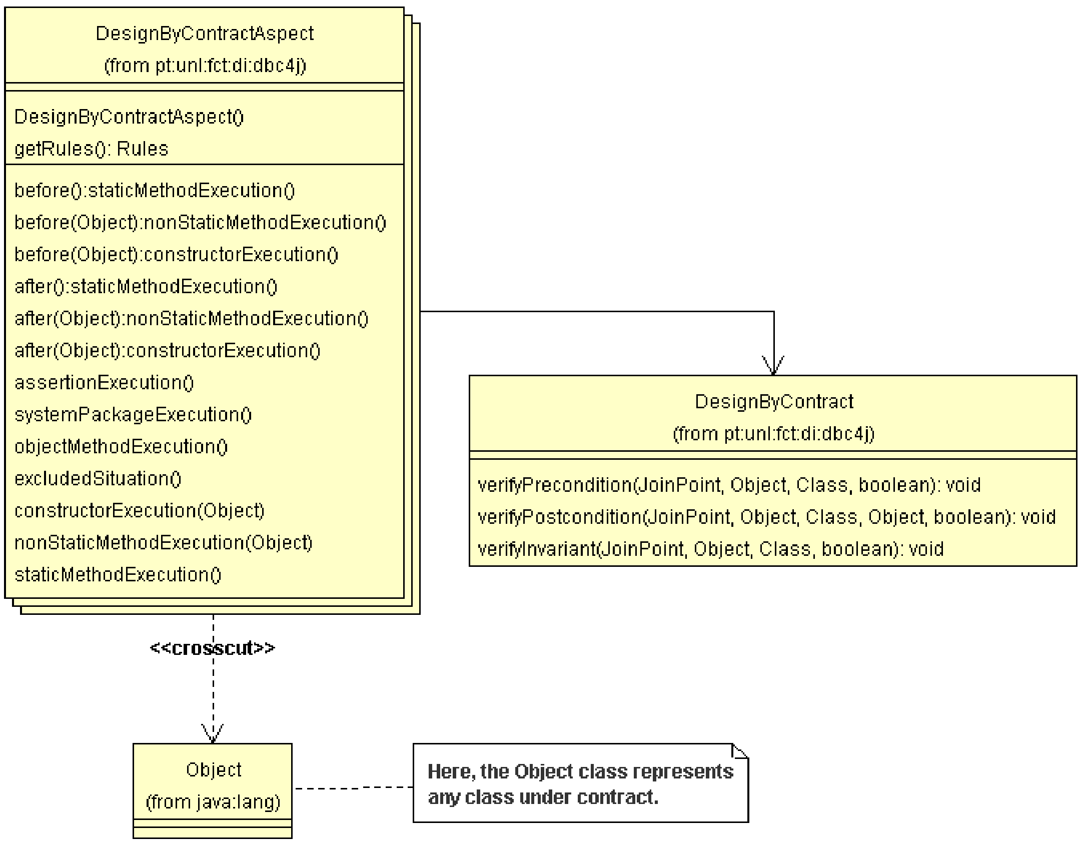
DesignByContractAspect
(from pt:unl:fct:di:dbc4j)
DesignByContractAspect()
getRules(): Rules
before():staticMethodExecution()
before(Object):nonStaticMethodExecution()
before(Object):constructorExecution()
after():staticMethodExecution()
after(Object):nonStaticMethodExecution()
after(Object):constructorExecution()
assertionExecution()
systemPackageExecution()
objectMethodExecution()
excludedSituation()
constructorExecution(Object)
nonStaticMethodExecution(Object)
staticMethodExecution()
DesignByContract
(from pt:unl:fct:di:dbc4j)
verifyPrecondition(JoinPoint, Object, Class, boolean): void
verifyPostcondition(JoinPoint, Object, Class, Object, boolean): void
verifyInvariant(JoinPoint, Object, Class, boolean): void
<<crosscut>>
Object
(from java:lang)
Here, the Object class represents any class under contract.

Figure 4.2: DesignByContract aspect diagram

4.3.2 The “Old” mechanism

Usage. The “old” mechanism is provided through the ContractMemory class. The base features are remember() and old() methods, and the argument is the this variable. remember() is called in preconditions, and old() in postconditions. Classes that use these features must implement the Cloneable interface, and consequently provide the clone() method.

An alternative is available for storing only primitive types, through the observe() and attribute() methods. This way, the class does not have to implement Cloneable. Analogously to remember() and old(), observe() is called in preconditions, and remember() in postconditions. These take as arguments the variable to store and a “tag”, which is a string key to identify the variable. As a convention, we recommend to use of the form className.attributeName (e.g. “Point.x”).

These method names mimic the solution proposed by Guerreiro's work on Design by Contract for C++ [Guerreiro00] [Guerreiro01], which in spite of its language idiosyncrasies provided some interesting ideas.

Implementation. The ContractMemory class (Figure 4.3) is actually a “mock” class: its behaviour is introduced by the ContractMemoryAspect. The aspect uses around advices to store object references and values of primitive types in *hashtables*. Using remember() and old(), the object is captured through the execution context and cloned, using the original reference hashValue() as key in hashtable. Using observe() / attribute(), the primitive types are stored as objects in the hashtable, using the tag hasValue() as key. The *autoboxing* feature of Java 1.5 simplifies this. In either case, method recursion is addressed through stacking of the hashtables. The ContractMemoryAspect reuses the DesignByContractAspect public pointcuts.

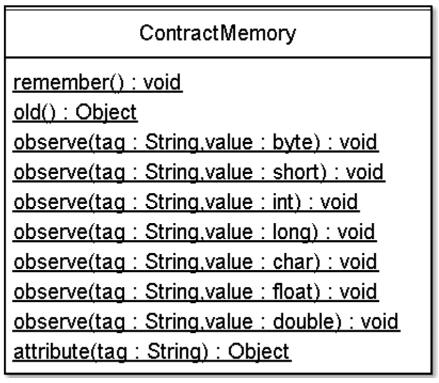
ContractMemory
remember() : void
old() : Object
observe(tag : String,value : byte) : void
observe(tag : String,value : short) : void
observe(tag : String,value : int) : void
observe(tag : String,value : long) : void
observe(tag : String,value : char) : void
observe(tag : String,value : float) : void
observe(tag : String,value : double) : void
attribute(tag : String) : Object

Figure 4.3: ContractMemory class

Figure 4.4 illustrates the implementation. For brevity, we omit methods observe() and attribute(), since the implementation is analogous, and some auxiliary pointcuts in the aspects, which are used for factorization.

The described approach is called the “Virtual mock” approach, typically used for test-driven development [Lesiecki02]. An pure object-oriented alternative would be the use of the “Null object” [Fowler99] and “Factory method” [Gamma94] design patterns.

4.3.3 Inheritance

Usage. Contract inheritance in DbC4J follows the Liskov Substitution Principle, for behavioural subtyping. This means that contracts are inherited through class extension. Redefined contracts are composed through disjunction with super assertions, in case of preconditions, and composed through conjunction with super assertions for postconditions and invariants. In order to understand this, one must distinguish two concepts: *partial* and *total* assertions. A partial assertion is a piece of code written in a class. A total assertion[4] is a runtime evaluation of an assertion for a specific object, taking into account the substitution principle. Thus, in order to evaluate a total assertion, one must evaluate a set of partial assertions. A total precondition is the logical disjunction of the set of partial preconditions. A

4 In Eiffel, this is called the “unfolded form” of the assertion.

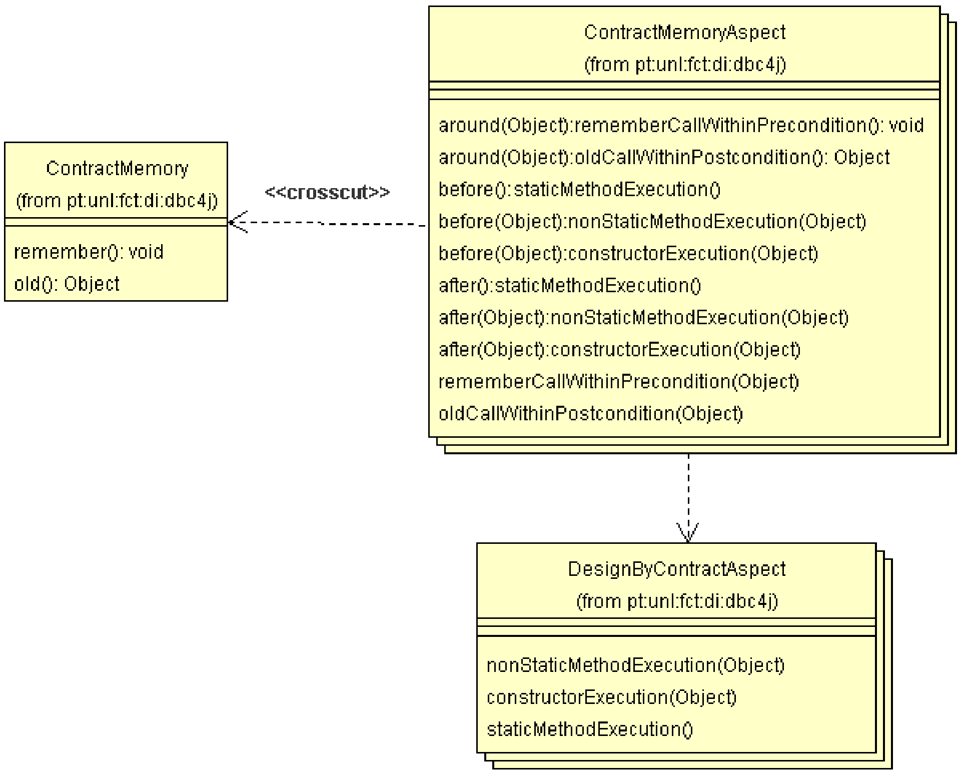
ContractMemory
(from pt:unl:fct:di:dbc4j)
remember(): void
old(): Object
<<crosscut>>
ContractMemoryAspect
(from pt:unl:fct:di:dbc4j)
around(Object):rememberCallWithinPrecondition(): void
around(Object):oldCallWithinPostcondition(): Object
before():staticMethodExecution()
before(Object):nonStaticMethodExecution(Object)
before(Object):constructorExecution(Object)
after():staticMethodExecution()
after(Object):nonStaticMethodExecution(Object)
after(Object):constructorExecution(Object)
rememberCallWithinPrecondition(Object)
oldCallWithinPostcondition(Object)
DesignByContractAspect
(from pt:unl:fct:di:dbc4j)
nonStaticMethodExecution(Object)
constructorExecution(Object)
staticMethodExecution()

Figure 4.4: ContractMemory aspect diagram

total postcondition is the logical conjunction of the set of partial preconditions that imply the correspondent partial postconditions. In each case, an empty set is equivalent to true. This is formalized in Figure 4.5.

$TotalPre_1 = \{ Pre_1 \vee Pre_2 \vee ... \vee Pre_n \}, n > 0$

$TotalPre_1 = true, n = 0$

$TotalPost_1 = \{ (Pre_1 \rightarrow Post_1) \wedge (Pre_2 \rightarrow Post_2) \wedge ... \wedge (Pre_n \rightarrow Post_n) \}, n > 0$

$TotalPost_1 = true, n = 0$

Figure 4.5: Total preconditions and postconditions

Independently of the visibility of the contracted methods, assertions can be either public or private. When extending a class, assertions must be private, unless the super class has a *copy constructor*.

Implementation. Before searching for an assertion for a given method, the object's hierarchy is obtained, with the exception of the Object class, the class from which every class is derived, either explicitly or implicitly. Then every assertion in the class hierarchy is evaluated: preconditions are evaluated bottom-up, and postconditions/invariants are evaluated top-down. Although at first glance not all partial assertions need to be checked (due to the *short-circuit effect*[5]), they actually must be, because of the “old” mechanism side-effect implementation. Also, in order to follow the relaxed postcondition rule, all partial preconditions must be evaluated and stored for postconditions evaluation. The preconditions are stored using special package visible methods in the ContractMemory class.

Public assertions represented a problem while evaluating inherited contracts through reflection, since public methods are always evaluated with respect to their runtime type [Gosling05]. Thus, if class A and B both have a public non-static method preFoo(), given an instance of B, it is not possible to

5 When evaluating a logical expression composed by disjunction, any “true” value implies that the whole expression is always true – a “greedy” evaluation can be used. On the other hand, “false” values short-circuit conjunctions.

execute the A.preFoo() using type casting. On the other hand, static and private methods are evaluated with respect to their static type [Lindholm99]. Since it is sometimes necessary to evaluate the internal state of the object, static methods are out of the question. Private methods are one solution, possible through the AccessibleObject.setAccessible(boolean) method [Sun04], that allows breaking Java's visibility protection[6]. Thus, we support contract overloading, as long as the contract methods are declared as private. An alternative solution is to call the super class copy constructor to create a replica of the object, and then calling the method for that instance. We also support this option in our prototype.

4.3.4 Contract rules

Usage. Some authors argue that contract validation should not be disabled in production builds, or at least not completely [Meyer97] [Mitchell02]. As such, the library supplies a mechanism which we call *contract rules* – the possibility to disable assertions based on their type or in the specific class to which they apply. There are two types of rules static rules and runtime rules.

Static rules are written in a properties file, as exemplified in Listing 4.5. With this file it possible to set the default assertion check level and the classes for which given check levels are to be used. The check levels are the following:

- no: no checks are performed;
- pre: only preconditions are checked;
- post: both preconditions and postconditions are checked;
- all: preconditions, postconditions and invariants are checked.

This configuration file was inspired in the Language for Assembly of Classes in Eiffel (LACE) files [Meyer91] [Meyer97]. The file is loaded at the library start-up; if the file is not found, a conservative configuration is assumed (all default checks, and no specific classes).

Runtime rules are simply an API to modify the static rule configuration loaded at start-up. As such, it is possible to modify all of these fields, through the ContractRules class, as shown in Figure 4.6 and Listing 4.6.

Implementation. The static rules are loaded in the constructor of the DesignByContractAspect, the one responsible for evaluating the contracts. Since in AspectJ the aspect's constructors are executed concurrently with the advices, the constructor body had to be *synchronized*[7].

```
default.checks=pre
no.checks.classes=acme.examples.Main
pre.checks.classes=
post.checks.classes=acme.examples.Xpto
all.checks.classes=acme.examples.Foo;acme.examples.Bar
```

Listing 4.5: DbC4J configuration file example

The decision whether an assertion should be checked is taken inside the advices. A consequence of this is that even with all assertions disabled, advices are still executed.

The ContractRules class is a "mock" class, that is, the actual code is performed by the ContractRulesAspect, which replaces the method call with an around advice that returns a reference to the Rules object.

4.3.5 Other features

Usage. In some situations, one might want temporarily violate a contract with an operation, only to

6 AspectJ's "privileged" keyword allows an aspect to access private members of a class, but not through reflection.

7 In Java, "synchronized" code is code that is not executed concurrently with any code in the same program.

```
import pt.unl.fct.di.dbc4j.ContractRules;

public class SomeClass {
   ...
   public static void someMethod() {
      ...
      Rules rules = ContractRules.getRules();
      rules.setDefaultChecks(CheckLevel.PRE);
      rules.addClass("Foo", CheckLevel.NO);
      rules.addClass("Bar", CheckLevel.ALL);
      ...
   }
}
```

Listing 4.6: Example on using the runtime API to change the contract rules

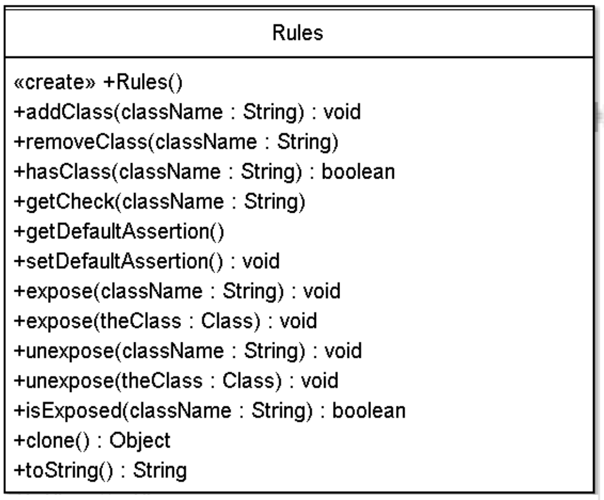
Rules
«create» +Rules()
+addClass(className : String) : void
+removeClass(className : String)
+hasClass(className : String) : boolean
+getCheck(className : String)
+getDefaultAssertion()
+setDefaultAssertion() : void
+expose(className : String) : void
+expose(theClass : Class) : void
+unexpose(className : String) : void
+unexpose(theClass : Class) : void
+isExposed(className : String) : boolean
+clone() : Object
+toString() : String

Figure 4.6: Rules class diagram

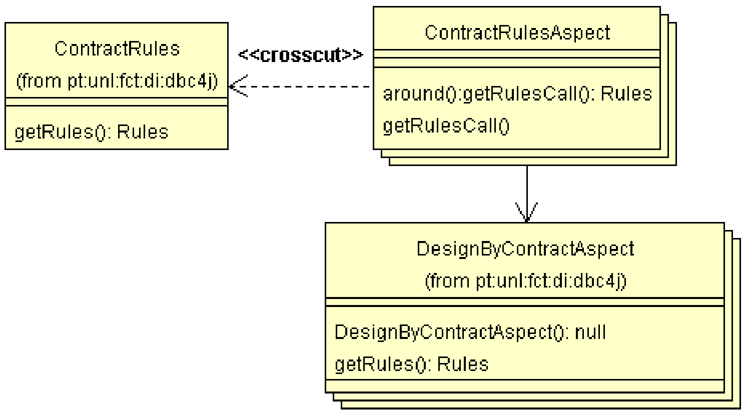

Figure 4.7: ContractRules aspect diagram

recover it in subsequent operations. As shown in Chapter 2, the Nemerle and Spec# languages support this through the expose statement. Our *exposed code* feature is supported in the ContractRules class. Unlike the previously mentioned languages, the DbC4J implementation takes a different level of granularity, by exposing a class instead of an object. Also, the semantics is different – contracts are not evaluated immediately after exposure. Listing 4.7 provides an example.

Applications written with DbC4J to be compiled as regular Java (or AspectJ) applications, are supported through a “mock” library. As such, *unplugging* the contracts is achieved by replacing the library JAR file by the “mock” library JAR file.

```
import pt.unl.fct.di.dbc4j.ContractRules;

public class Person {
    ...
    public void setAddress(String address) { ... }
}

public class Client extends Person {

    public void setIdNumber(Integer idNumber) { ... }

    public void setAddressAndIdNumber(String address, Integer idNumber) {
        Rules rules = ContractRules.getRules();
        rules.expose(Client.class);
        setAddress(address);
        setIdNumber(idNumber);
        rules.unexpose(Client.class);
    }

}
```

Listing 4.7: DbC4J's expose feature example

Implementation. The exposed code feature reuses the Rules class, as it has been implemented as a secondary feature. It works very similarly to the rules API, using a new *hashtable* for the effect. As it was an afterthought, it would be complex to force a re-evaluation of the contracts, since the contract checking is performed by a different aspect.

The unplugging feature implementation is very simple, since the "mock" library contains exactly the same classes has the full library. This happens because all public classes in the library that are used in contract bodies are merely mocks – they behaviour is accomplished through an aspect (ContractMemory → ContractMemoryAspect; ContractRules → ContractRulesAspect).

4.4 Prototype Limitations

We will now discuss some of the limitations of our implementation, to conclude that none of them seriously hinders its usefulness and usability. Some are due to its nature (that is, using executable code), some are due to simplifications made to attempt to reach a "clean" design, and finally some are simply because it would require more development time and effort.

Old mechanism. The current implementation of the "old" mechanism relies on explicit "store" calls. This can be cumbersome and error-prone. Furthermore, the "memory" features are available outside pre/postconditions and can be misused.

Side-effect free. This implementation does not verify if the contracts are side-effect free. Actually, the current "old" mechanism implementation is dependent on side effects.

Constructors. In the current prototype solution, constructors are treated like regular methods for polymorphism purposes. This solution however, hinders some issues. If class B extends class A, its constructor should honour A's constructor contract. Currently, this will only happen if the second constructor explicitly calls the first one. Preconditions are not guaranteed to be contravariant, nor postconditions to be covariant. Some existing solutions do not implement this garanty willingly (e.g. C4J), which raises the discussion on whether it is useful or not. The Java language specification states: "*Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.*" [Gosling05].

Furthermore, constructors whose first line is not a call to another constructor of the same class or of a superclass, have an implicit call to the default superclass constructor ("super();"). Thus, a correct implementation should ignore the contract of the superclass constructor, when being called in the constructor, either explicitly or implicitly.

Visibility checks. Contracts should be public (since they are to be “read” by the client), and should have access only to public members. The first part is possible, if copy constructors are available when extending classes. The second part is not checked.

Performance. The library takes a significant toll on execution time. We will return to this issue in Chapter 6.

Fault-tolerance. Errors in contracts result in confusing stack traces, since they reveal the library's infrastructure (aspect and reflection). Furthermore, errors in the rules configuration file are not addressed.

Expose. The expose mechanism has a high granularity (classes), and does not imply contract verifications upon its end. Furthermore, missing “unexpose” calls are not rectified at the end of the method (the classes will be permanently be exposed, unless otherwise stated).

Concurrency. The current implementation does not address any concurrency issues. It has not been analysed what would be the impact of concurrent programming with the library.

Cloning. Any class to be used with the “old” mechanism must be `Cloneable`. This is a Java issue, which cannot be addressed at this level of implementation.

Signature checking. Wrong signatures on contracts are verified, but simply ignored. For example, forgetting to put the “result” variable in a postcondition signature will result in it to be ignored, and the programmer may remain unaware of that and still think his program is properly contracted.

Interface support. There is no explicit support for writing contracts in interfaces. It is possible to write contracts for interfaces though, using inter-type declarations to write assertions. However, this does not address contract inheritance, nor it does accommodate overriding interface assertions in classes. For all effects, these interface assertions work as *in-lined* code in the class.

Reflection. Currently, contract checking is not performed for reflective calls. This happens because such calls are not captured directly by “regular” pointcuts, but only by specific pointcuts to the reflection library. So, it is possible to add such support by writing new joinpoints for equivalent reflective calls to the existent joinpoints, therefore duplicating the number of joinpoints in the prototype. We do not find it justifiable at this point, but it is implementable.

Loop contracts. Our approach of using executable code for writing contracts implies that we cannot address contracts for loops. However, many modern programming languages, such as C#, Java, and PHP feature “for each” loops, which by definition are guaranteed to terminate, as part of the iterator language specification contract.

Inner classes and anonymous classes. The particular kind of classes were not addressed.

4.5 Summary

In this chapter, we stated the objectives of our prototype. From these objectives, some design decisions had to be taken. We chose executable code as the form for contract writing, since it is the most natural form for programmers. We chose Aspect-Oriented Programming as the generative technique, due to its non-intrusive nature. We chose the AspectJ language, due to its maturity and available knowledge. We chose to design the “result” and “old” constructs through a parameter and a class, respectively, as the best compromise between performance and simplicity. We also chose Boolean methods for assertions as the simplest solution, although not the shortest in terms of code writing.

Next, we discussed the prototype features, both in terms of usage and implementation. Preconditions, postconditions, and invariants, are the base features for such a solution, which are captured through AspectJ pointcuts and evaluated through Java reflection mechanisms. The “old” mechanism raised several issues, and we took a difficult trade-off between simplicity and performance. Another challenge was polymorphism, as the substitution principle applied to assertions raises several technical issues, especially at such high-level of abstraction. We used reflective and visibility breaking techniques in order to achieve the desired semantics. Contract rules are the mechanism developed for the prototype, in order to disable certain assertions, either by type of class. This can be configured statically through a configuration file that is loaded at start-up, or at runtime through an API. Finally, the expose statement allows temporary disable of assertion checking in a code zone.

The prototype limitations are discussed in a specific section. Most limitations are directly perceived from the design decisions and implementation details, but others are a little more subtle, such as assertion side-effects, performance considerations, concurrent programming, etc. In the next chapter, we apply the prototype to some case studies.

Chapter 5
Case Studies

In this chapter, the DbC4J usage experience is discussed, by means of two case studies. The first case study is a “real-world” application – FPL Tower Interface –, and the second one consists in a set of classes proposed by the book *Design by Contract, by Example*.

5.1 FPL Tower Interface

5.1.1 Brief introduction to the system

The first case study, “FPL Tower Interface”, was developed in the context of the SOFTAS project (POSI/EIA/60189/2004), funded by the Portuguese Foundation for Science and Technology (http://aosd.di.fct.unl.pt/softas/). The “FPL Tower Interface” is a graphical user interface for air traffic control operator support, at *NAV Portugal*, the company responsible for civilian air traffic control in Portugal (and a partner of the SOFTAS project). Operators in the control tower can access and modify data corresponding to scheduled flights and aerodrome conditions. Flights are categorized into: *departures*, *arrivals*, and *overflights*. For each type there is a set of parameters, some of which are modifiable. This subsystem is a Java application running on top of an internally developed middleware system, which connects all the sub-systems used by NAV. Mashkoor *et al.* provide a summary description of the case study requirements [Mashkoor07]. Our work was performed in the context of the SOFTAS work package “Programming Languages for Aspect-Programming”.

5.1.2 Objectives and scope of the study

The purpose of this case study is to assess the usability of the DbC4J library in an existing Java application. Due to the nature of this system, namely the dependency on a middleware system, it was only possible to test the *data layer*[1], since at the time the middleware was not available to the project, nor the specifications to produce a mock-up middleware. In the application, the following classes were identified as data layer classes:

- FPL (abstract class);
- Arrivals (FPL sub-class);
- Departures (FPL sub-class);
- Overflights (FPL sub-class);
- Aerodromes (FPL sub-class).

The application's System Requirements Specification document supplies all the information regarding data classes. Refer to the “Acronyms and Definitions” annex at the end of this document for a summary of the parameters specification.

5.1.3 Analysis

Looking at the data definition classes source code, it is obvious the identification of defensive programming. Let's take Listing 5.1 as an example. This is a *setter* method of the Arrivals class, which

1 According to the *three-tier architecture*, that defines three levels of modularity: *presentation*, *logic*, and *data*.

```
1     /** sets the Estimated Time at Boundary
2      * @param etb The estimated time at boundary
3      */
4     public void setEtb(String etb) {
5        try {
6           Short e = new Short(etb);
7           this.etb = e.shortValue();
8        }
9        catch(NumberFormatException e)
10       {}
11    }
```

Listing 5.1: Arrivals.setEtb (defensive implementation)

has usually a very straightforward implementation[2]. This version though, has a try statement, in order to validate its input. Moreover, the empty catch statement on line 10 means that invalid input will be "silenced".

Using DbC4J, an alternative implementation was performed (Listing 5.2). This implementation adds a precondition to validate the input. In order to maintain the class interface, an overloaded method was

```
1  /** sets the Estimated Time at Boundary
2  * @param etb The estimated time at boundary
3  */
4  public void setEtb(String etb) throws NumberFormatException {
5      setEtb(Short.parseShort(etb));
6  }
7
8  public void setEtb(short etb) {
9      this.etb = etb;
10 }
11 public boolean preSetEtb(short etb) {
12     short hours, minutes;
13     hours = (short) (etb / 100);
14     minutes = (short) (etb - hours);
15     return hours >= 0 && hours <= 23 && minutes >= 0 && minutes <= 59;
16 }
```

Listing 5.2: Arrivals.setEtb (contracted overloaded implementation)

introduced. Note that the setter method now throws the exception (line 5) instead of silencing it (thus the responsibility of validating this is on the client). The overloaded method just does what a regular setter should do. The first method acts as a *defensive layer* (see Chapter 2).

However, while applying this technique to the remaining methods of the Arrivals class, the preconditions tend to repeat themselves, since most class members are time values. The underlying problem is that this class design does not properly typify the member variables, relying solely on primitive data types. As such, an alternative implementation is proposed (Listing 5.3). By correctly typifying the data type, no contract is required in Arrivals regarding input verification. Instead, the contract is factorized in the Time class (Listing 5.4). This auxiliary class is assured to have no invalid value through the class invariant (line 5), and the parser function precondition (line 18).

```
public void setEobt(Time etb) {
   this.etb = etb;
}
```

Listing 5.3: Arrivals.setEtb (typed implementation)

2 Some IDEs, such as Eclipse, generate these methods automatically.

```java
public class Time {
    private short hours, minutes;

    public boolean invar() {
        return hours >= 0 && hours <= 23 && minutes >= 0 && minutes <= 59;
    }

    public Time(short hours, short minutes) { ... }

    public static Time parseTime(String time) {
        short hours, minutes;
        hours = Short.parseShort(time.substring(0, 2));
        minutes = Short.parseShort(time.substring(2, 4));
        return new Time(hours, minutes);
    }
    public static boolean preParseTime(String time) {
        return time.matches("\\d\\d\\d\\d");
    }

    public short getHours() { ... }

    public short getMinutes() { ... }

    public void setMinutes(short minutes) { ... }

    public void setHours(short hours) { ... }

    public String toString() {
        String hoursStr = Short.toString(hours);
        String minutesStr = Short.toString(minutes);
        return FplDataUtil.appendZeros(hoursStr, 2) + FplDataUtil.appendZeros(minutesStr, 2);
    }
}
```

Listing 5.4: Time class

Note that this alternative solution breaks the original interface, since parameters are changed from short to Time. Nevertheless, *reliability* is assured for time value; as well as a *separation of concerns*, since it is now clear where is the checking code localized (contract), and who should check it (the client). The same applies to the remaining data types used in Arrivals: Runway and Stand.

Now that input verification has been factorized in the data type classes, contracts can be written regarding the actual *business logic*. Still in the Arrivals class, the specification specifies that:

1. The "Actual Time of Arrival" (ATA) value cannot be lower than any "Touch & Go" value.
2. The ATA value cannot be supplied without specifying the "Runway" (RWY) value, and the RWY value cannot be supplied without specifying the ATA value.

Both rules can be assured through the class invariant. However, since the class setters are public, they would violate the second rule of the invariant. As such, these particular setters must be rewritten as private, and new public method must exist, in order to violate the invariant only temporarily. This is show in Listing 5.5.

Although this seems to be a practical solution to the problem, it does not work in this particular case. (In fact, one might want to keep using the individual setters in both classes for situations that do not violate the invariant!) The problem is that the RWY setter is inherited from the FPL class, and the Java language does not allow overloading a method with reduced visibility. Of course, one could rewrite the FPL class method to private, or rewrite the RWY attribute to protected, but that would change the FPL class design, which we do not want to do, since the problem lies in its sub-class. The main issue here is

```
public boolean rule1() {
    if (getAta() == null || getTgTime().size() == 0)
        return true;
    for (Time tg: getTgTime()) {
        if (getAta().lessThan(tg))
            return false;
    }
    return true;
}

public boolean rule2() {
    return (getRwy() == null && getAta() == null) || (getRwy() != null && getAta() != null);
}

public boolean invariant() {
    return rule1() && rule2();
}

public void setAtaAndRwy(Time ata, Runway rwy) {
    setAta(ata);
    setRwy(rwy);
}
```

Listing 5.5: Arrivals class invariant

that we need to perform two operations that violate a contract when acting separately, but make sense together. In other words, we need to atomise these operations, in terms of contract verification. As such, we need to use the "exposed" mechanism, as show in Listing 5.6.

Classes Departures and Overflight have analogous contracts. In conclusion, contracts show improvements in this case study, at least in the analysed subset, the data definitions:

- input verifications are placed in the data types preconditions and invariants;
- business rules are placed in the data classes invariants.

Nevertheless, writing the contracts for the FPL Tower Interface subsystem was preceded in a redesign of the data classes. If this redesign had not been performed, contracts would have become larger and redundant. The redesign consisted in: correctly typing parameters with specific data types, instead of Java primitive types; updating the application to Java 1.5, in order to use some of the new features available in it, namely the Enumerations. Java's enumerations are particularly useful, since they eliminate the need of several contracts.

Furthermore, Unit Testing revealed to be particularly useful used in combination with contracts. Test units were written to address situations where the contracts are needed – in precondition and invariant violation situations (postconditions violations are not expected, except in the presence or supplier bugs). Since the DbC4J library is relatively transparent for Java/AspectJ tools, JUnit 4.0 was used for writing the test units, with no significant issues.

```
public void setAtaAndRwy(Time ata, Runway rwy) {
    Rules rules = ContractRules.getRules();
    rules.expose(Arrivals.class);
    setAta(ata);
    setRwy(rwy);
    rules.unexpose(Arrivals.class);
}
```

Listing 5.6: Arrivals.setAtaAndRwy (exposed)

5.2 Design by Contract, by Example

5.2.1 Objectives and scope of the study

The purpose of this case study is to port a set of classes described in *Design by Contract, by Example* [Mitchell02], from Eiffel to Java, using the DbC4J library. This way, it should be possible to compare contracts for basic data structures.

5.2.2 Analysis

Generic data structures are naturally reusable modules, so this is a perfect situation for applying Design by Contract. DbC4J revealed to be capable of writing “translated” contracts from Eiffel with minimum changes (except for language specific issues). Listing 5.7 presents an example comparison of a “simple version of stack, with few features and no protection against overfilling, but with carefully-written contracts” [Mitchell02]. The Java/DbC4J solution is similar to the Eiffel solution, especially if the old mechanism cumbersomeness is excluded[3].

Once again, Unit Testing was used. Since this case study was based on a non-existent Java code base, it was possible to write interfaces, contracts, and test units, before actually writing the actual code. This reveals an interesting use for contracts in Test-Driven Development. Indeed, this was to be expected, since many authors consider contracts part of interfaces [Mitchell97].

5.3 Syntactic contracts vs Behavioural contracts

As discussed previously in Chapter 2, contracts can be grouped in several levels, including syntactic and behavioural contracts. While the focus of this thesis is on behavioural contracts, these case studies (as well as a reflection over the author's development experience) revealed that many of the repeated contracts are simple checks that could be solved by syntactic constructs, namely:

- Non-null values – as in Eiffel, using the ? token; in JML, using the not_null keyword; or in Spec#, using the ! token[4].
- Ranged values – as in C#, using unsigned variables[5].

5.4 Summary

In this chapter two case studies were discussed. The first case study is the refactoring of a small-sized production application in the industry. Several traces of defensive programming were identified and some alternative contract-based implementations were suggested. Moreover, it was shown that contracts are very cumbersome are repetitive in a poorly object-oriented design. Finally, certain business rules justified the use of the expose statement.

The second case study is the rewriting of a set of Eiffel reusable data structure classes, proposed in a book. The prototype showed to be able to handle the necessary contracts for the proposed classes. In particular, this case study also showed a synergy between design by contract and unit testing.

Finally, some brief considerations were presented on the role of syntactic and behavioural contracts.

Having applied the prototype, it is now possible to better evaluate it. We do so in the next chapter.

3 For brevity, the class copy constructor and clone method are omitted in the Java version.

4 Interestingly, this is a complement to C# philosophy. Primitive type ('structs') variables cannot be assigned the “null” value unless explicitly declared as “nullable”.

5 Not to be mistaken with the “unsigned” keyword of C++, which is not a strongly typed language.

```eiffel
class SIMPLE_STACK[ G ]

creation
    make

feature {NONE}
    the_contents : ARRAY[ G ]

feature
    count : INTEGER

    item_at( i : INTEGER ) : G is
        require
            i_big_enough: i >= 1
            i_small_enough: i <= count
        do
            Result := the_contents.item( i )
        end

feature
    item : G is
        require
            stack_not_empty: count > 0
        do
            Result := the_contents.item( count )
        ensure
            consistent_with_item_at:
                Result = item_at( count )
        end

    is_empty : BOOLEAN is
        do
            Result := count = 0
        ensure
            consistent_with_count:
                Result = ( count = 0 )
        end

feature
    make is
        do
            create the_contents.make( 1, 100 )
        ensure
            stack_is_empty: count = 0
        end

feature
    put( g : G ) is
        do
            count := count + 1
            the_contents.put( g, count )
        ensure
            count_increased:
                count = old count + 1
            g_on_top:
                item_at( count ) = g
        end

    remove is
        require
            stack_not_empty: count > 0
        do
            count := count - 1
        ensure
            count_decreased:
                count = old count - 1
        end

invariant
    count_is_never_negative: count >= 0

end
```

```java
public class Stack<G> implements Cloneable {
    private Object[] contents;
    private int count;

    // ...

    public Stack() {
        contents = new Object[100];
        count = 0;
    }
    public boolean postStack() {
        return count() == 0;
    }

    public int count() {
        return count;
    }

    public boolean preItemAt(int i) {
        return i >= 0 && i < count();
    }
    public G itemAt(int i) {
        return (G) contents[i];
    }

    public boolean preItem() {
        return count() > 0;
    }
    public G item() {
        return (G) contents[count - 1];
    }
    public boolean postItem(G result) {
        return result.equals(itemAt(count()-1));
    }

    public boolean isEmpty() {
        return count == 0;
    }
    public boolean postIsEmpty(boolean result) {
        return result == (count() == 0);
    }

    public boolean prePut() {
        ContractMemory.remember();
        return true;
    }
    public void put(G g) {
        contents[count] = g;
        count++;
    }
    public boolean postPut(G g) {
        Stack old = (Stack) ContractMemory.old();
        return count() == old.count() + 1
            && g.equals(item());
    }

    public boolean preRemove() {
        ContractMemory.remember();
        return count() > 0;
    }
    public void remove() {
        contents[count-1] = null;
        count--;
    }
    public boolean postRemove() {
        Stack old = (Stack) ContractMemory.old();
        return count() == old.count() - 1;
    }

    public boolean invariant() {
        return count() >= 0;
    }
}
```

Listing 5.7: An example written in Eiffel (left) and Java/DbC4J (right)

Chapter 6 Evaluation

In this chapter, we present a critical evaluation of our accomplishments. First, we compare the present state of our system with other available solutions, using a detailed criteria. Next, the evaluation of the requirements stated in Chapter 4 is made. A performance experiment is performed and discussed. Finally, some technical details on the prototype are unveiled, and several ideas for future development are presented.

6.1 Criteria

A comparison criteria "emerged" in the course of this thesis, and is explained next. First, a complete list is shown, then each point is discussed. Note that these points are not all self-contained, that is, a feature or decision may affect others.

1. Release;
2. Required JVM;
3. Instrumentation approach;
4. Specification language;
5. Contract specification;
6. Contract identification;
7. Contract placement;
8. Contract inheritance;
9. Local variables;
10. Result variable;
11. Assertion tag name;
12. Old state;
13. First-order logic;
14. Invariant types;
15. Invariant evaluation time;
16. Evaluation visibility;
17. Contract recursion;
18. Side-effects free;
19. Concurrency;
20. Check assertion;
21. Loop contract;
22. Frame rules;
23. Assertion failure;
24. Disabling granularity;
25. Configuration;
26. Unit testing integration;
27. Automatic documentation;
28. GUI support;
29. Implementation license.

1. Release

Which release is under evaluation?

This is not exactly a criteria, but simply a free field with the indication of which version is the comparison based. If there is a standard available, the release will be the standard reference. The release date is also available.

2. Required JVM

What Java Virtual Machine version is required?

(Affects "Contract specification", and "Check assertion")

This field only appears for the Java *ad-hoc* solutions. A higher version limits the library applicability for legacy applications, but increases the number of features available. Namely, Java 1.4 introduced the `assert` statement, and Java 1.5 introduced annotations.

3. Instrumentation approach

What is the approach for contract verification?

This field only appears for the Java *ad-hoc* solutions. A simple approach is to do **preprocessing**. This way, the Java code is transformed to include the contract checks, such that the source code to compile is standard Java.

Preprocessing requires access to the source code, which may not be true, if the programmer is working third-party components, such as Components Off The Self (COTS), a common business approach for reusable software components. Therefore, a different approach is required. Using a **modified compiler** it is possible to insert the contract checks at *bytecode* level.

A less intrusive approach is to use a specialized **class loader** to actually perform the instrumentation at compile-time. This is possible, using the reflection mechanisms present in the Java technology.

A very recent approach is to use Java 6's pluggable annotation processing API (**JSR 269**). This way, a compiler plug-in is developed, which enables all Java 6 compliant tools to seamlessly use this solution.

A higher level approach is to use an **Aspect-Oriented Programming** technology (such as AspectJ). Depending on the technology, it is possible to either introduce the contract checks at compile or run-time.

4. Specification language

In which language is the contract specified?

Using the **base language** benefits the programmer in two ways: he/she can use a familiar language, thus decreasing the learning curve; and he/she can use the whole expressiveness of the base language, although some features may be limited.

On the other hand, using a **different language**, has some advantages if the language provides a cleaner syntax (e.g. OCL) or features not available in the language (e.g. universal quantifier).

5. Contract specification

How is the contract specified in the code?

(Affects “Contract identification” and “Contract placement”)

The most natural solution is to use **executable code**, as if the contract is a function. However, for *ad-hoc* solutions this may not be feasible.

Therefore, the simplest solution is to use **comments**, which are treated by a pre-processor. Other than creating a dependency of an external tool, this solution also implies a loss of traceability (e.g. for debugging), for the code is transformed before actual compilation.

Javadocs are a special type of comments in Java, which are processed by the 'javadocs' tool, and also by other tools. While this mechanism can be seen as lightweight form of metadata, it is rather limited, since it is not processed by the compiler.

Another option is to use **metadata**, if the language supports it (e.g. Java has a metadata facility called *annotations*). Unlike comments, metadata is a first-class mechanism which is compiled into the bytecode and therefore processable by for other tools.

6. Contract identification

How is the contract identified?

(Affects “Automatic documentation”)

The best solution is that the contract is a **first-class** mechanism (e.g. a `require` clause in Eiffel). This clearly identifies the contract for the human readers and eases the processing of the contract (either for checking or for documentation). However, for *ad-hoc* solutions this may not be affordable.

If a first-class mechanism is not available, an existing mechanism can be reused. This can be accomplished through a **naming convention** (e.g. the `preFoo()` function is the precondition of the `foo()` function).

If contract specification is through a comment, then its identification is also through a **comment**.

7. Contract placement

Where is the contract placed?

Since all programming languages analysed in this dissertation are object-oriented (or multi-paradigm, but with Object-Oriented features), contracts are normally placed with **classes**.

Nevertheless, if the language supports **interfaces**, placing the contract there, improves the reusability of the contracts (especially because most languages do not support multiple-inheritance).

An additional possibility is the use of placing the contracts in an **external** module or file.

8. Contract inheritance

How are contracts inherited and refined?

Polymorphism is one of the most used mechanisms in object-oriented programming. As such, contracts must be addressed when in the presence of polymorphism. The "Liskov substitution principle" defines how contract are inherited, but there are several strategies on how to implement it, as discussed in Chapter 1.

The most usual algorithm is the one defined in "**classic**" Eiffel [Meyer91] [Meyer97].

Findler et al. [Findler01] proposed a set of rules which are stricter than the Eiffel interpretation of the substitution principle.

Recently, the **Ecma** standard for Eiffel has relaxed the "classic" algorithm, regarding postconditions.

There are however, some **other** algorithms, typically for simplifying the implementation or to improve performance.

9. Local variables

Is it possible to use local variables in the contract assertions?

While most assertions are simple (e.g. verifying arguments values and ranges), having the possibility of declaring local variables can simplify the definition of complex contracts, and increase the expressiveness of the contract. A work-around is to use method calls from the contract, if that possibility is available.

10. Result variable

Is it possible to use the 'result' variable?

When defining postconditions it is useful to be able to refer to the `result` value of a method call.

11. Assertion tag name

It is possible to identify each assertion?

(Affects "Automatic documentation")

When defining assertions, it is interesting to be able to "tag" each one with a name and/or description, for documentation purposes.

12. Old state

How is it possible to access the 'old' state of the object (or its attributes)?

When defining postconditions it is useful to be able to refer to the `old` state of the object, that is the state of the object before the call of the method. There are several ways to do so.

One way is to be able to explicitly store local variables of **primitive types**, and retrieve them at postconditions. Usually, this is only used if a more elegant form is not available.

Since usually we are interested only in variables from the object itself, then a more straight mechanism is to simply be able to access the object at that state. This is done by accessing it through **cloning**, or through **reference** if a clone is not available.

The most complete solution is to use old for evaluating an arbitrary **expression** involving an object attribute or method call.

13. First-order logic

Which first-order logic quantifiers and operators are possible to use?

First-order logic, while not indispensable (it is always possible to emulate them through the language basic constructs), can make contracts more expressive, and more compact.

The **universal quantifier** (∀), usually called "for all", verifies if for all members of a set, a given condition is met.

The **existential quantifier** (∃), usually called "exists", verifies if for at least one member of a set, a given condition is met.

The **conditional operator** (→), usually called "implies", verifies that whenever condition A is true, then condition B must also be true.

Some solutions also offer the bi-conditional operator (↔).

14. Invariant types

Which invariant types are available?

The most common type of invariant is the **class** invariant, an assertion that must always be true (usually considering each method an atomic instruction).

However, some languages also support **method** invariants, an assertion that must always be true, before and after the execution of the method. Note that you can always have the same effect by specifying a precondition equal to a postcondition (or write the assertion in a method, and then call it in the pre and postcondition).

Additionally, some other solutions even support **attribute** invariants. This is usually irrelevant, since the same can be obtained by class invariants, but it is an alternative fine grain of invariants.

15. Invariant evaluation time

When is the invariant evaluated?

An invariant is a condition that must always valid in all the observable states during the lifetime of an object. Thus, it is commonly accepted that a method can temporarily violate the invariant, as long as it is restored at the end of the method call. Therefore, the most common evaluation time is **after the return** the method calls.

At first glance the previous solution may seem enough, but it may not be enough in some situations. A known problem is the *Indirect Invariant Effect* [Meyer97]. A simple way to avoid this is to perform invariant evaluation **before** the method calls. Some languages/solutions ignore this, in order to improve performance.

Normally, invariants are only checked after normal method calls, i.e., after method calls that completed by return. However, some solutions check invariants **after exception**. This addresses the situations where one wants to ensure that a specific state of the object is to maintained, even in the occurrence of an (expected) exception.

While so far it has been assumed that methods are atomic instructions, in terms of invariant evaluation, this grain may be increased or decreased, by explicitly declaring a part of the code as **exposed**. This means is that part of the code is considered "atomic", i.e., that no invariant evaluation is performed during it.

16. Evaluation visibility

Where is the contract evaluation performed?

(Affects “Invariant evaluation time”)

The most common approach is to evaluate contracts for **public** visibility method (or constructor) calls, since these features are the only ones available to a client class.

However, it seems reasonable to impose the contract (at least, invariants) to **private** calls, and some solutions do just that.

17. Contract recursion

How are method calls from contracts checked?

Infinite recursion loops can occur if method calls from contracts are checked. As an example, consider a class with a `count()` method, which is used by the class invariant. Whenever a method call is performed, the class invariant is evaluated, which calls the count method, which will force the invariant evaluation, and so on.

Some implementations simply **enable** contract recursion, by imposing the programmer to avoid such situations.

A common approach is to **disable** contract recursion. This way, although “innocent” checks are lost, infinite recursion is avoided.

A more elaborate approach is to enable **safe** recursion, through stack trace bookkeeping. Although this decreases performance by introducing an execution overhead, it can check more situations.

18. Side-effects free

Running the application with contract checks will produce the same result as without?

Contracts should not modify the state of the objects that they are operating. Some implementations check this at compile time.

19. Concurrency

Is there support for coherent concurrent calls to a method and contract?

Concurrent applications may produce a situation where a contract is being checked at the same time that a method that changes the internal state of the object. If this occurs, the contract may fail without cause (a condition was only temporarily violated), or the contract may not fail where it should. As such, a synchronization mechanism is required to avoid these situations.

20. Check assertion

Is there an explicit assertion statement?

An explicit assertion statement, that may be used at any point in the code, is an additional feature that helps reassuring the programmer that a certain assumed state of the program is actually valid. Many programming languages, with or without full fledged Design by Contract, support this.

21. Loop contract

Is there contract support for loop statements?

Since a common bug in software applications is an infinite loop, some solutions support some clauses for checking that loops are well-written and always end. Note that this can be achieved through regular language mechanisms. A loop invariant is a condition that must be true at every step of the iteration; a loop variant is a non-negative, decreasing integer expression.

22. Frame rules

Is there support for defining frame rules?

While common assertions specify values that certain attributes must have, it also possible to specify which attributes should not change. This is done via special assertions are called *frame rules* or frame conditions.

23. Assertion failure

What happens when an assertion check fails?

The simplest solution is program **termination**. This is acceptable, in general, since an assertion failure is a serious error.

However, at times, it is more practical to use the **exception** mechanism. It is more flexible, and it allows a more controlled program termination. It also can be used to retry an operation (which is useful for user input or network connections).

Some solutions simply display **warnings** during execution. This is not always adequate, since this way bugs can propagate or be ignored (the psychological weight of a warning is far lesser than of an error).

Additionally, some more elaborate solutions might include **user-defined handling**. This way, the programmer might define specific execution steps for a specific contract violation.

24. Disabling granularity

How can contracts be enabled and disabled?

(Affects "Configuration")

Most solutions offer a way to disable assertion checking (e.g. for production builds). The **simple** solution is to disable all assertions. This is usually achieved through a compilation flag.

However, one might want to enable assertions only for a specific set of **modules** (classes, packages, etc.). This can be useful, if you want check only a new part of the application, knowing that the remainder is stable. It is also useful if you want to disable checks in a performance sensitive area (e.g. a possible bottleneck) of your application.

In the same way, you might also want to restrict the **types of assertions** you want to check. This is useful for situations where are server classes are pretty stable, but you're client classes are not; you only need to check preconditions.

Additionally, other approaches may exist. A possible one is to assign **priorities** to assertions, and perform checks up to a certain level of checking.

25. Configuration

How is the disabling granularity configured?

If enabling/disabling of certain assertions is available, then this needs to be configured somehow. The simplest way is at start-up, through a configuration **file**.

Additionally, one might want a more complex type of control, and perform configuration through a **run-time API**.

26. Unit testing integration

Is the solution integrated with unit testing?

Some solutions offer some level of integration between contracts and unit tests, in order to explore the synergy between the two techniques. The main advantage is that both approaches are supported by the same tool.

On the other hand, when integration is not provided, it may still be possible to use them together.

27. Automatic documentation

Does the solution support automatic documentation for contracts?

Since the main goal of Design by Contract is to specify interface behaviour, it is a natural concern to have contracts in documentation. This is usually simpler for solutions where contracts are first-class mechanisms.

28. GUI support

Does the solution have a graphical user interface?

Some solutions provide a GUI, stand-alone or IDE plug-in. This can allow typical editing features (e.g. syntax highlighting), simplified configuration, and quick feedback from the interpreter/compiler.

29. Implementation license

What is the license type of the solution implementation(s)?

Industry solutions usually produce **proprietary** solutions. This means that the software is payed, or a free of charge (but restricted) version is available.

Academic or research solutions usually produce Free or Open-Source Software (**FOSS**) solutions. While quality is not necessarily better (or worse), its usage is free, the source code is available, and there is usually more detailed documentation available.

If the solution has reached a significant usage, then it is possible that is standardized. A **standard** solution is better, in the sense that it makes possible the appearance of more compiler and developer tools implementations.

6.2 Comparison with existent solutions

In this section, a comparison with existent solutions is presented, in terms of features provided and design options. The comparison is divided in two parts: the first one is between native solutions, and the second one between *ad-hoc* solutions.

Features or strategies are marked with '√' when they are available, with '±' when they are partly available, and with '?' when it was not possible to determine whether they are available. The results presented here are based in the existing documentation, as well as discussion with the correspondent community or tool developers. Actual experimentation was not performed on all the tools, since it would insufficient to extract the necessary information.

6.2.1 Native support

Eiffel, Blue, Chrome, D, Lissac, Nemerle, Sather, C, Java, PHP.

Since the languages compared here (Figure 6.1) support Design by Contract as a native feature, it is no surprise that they use executable code and have contracts as first-class constructs.

Eiffel, Blue and D follow the "classic" notion of subtyping as defined by Liskov. Chrome, Nemerle and Sather do not support contract polymorphism at all. Lissac though, as a more "low-level" programming language supports a less conventional form of contract subtyping, by allowing contracts to be inherited (without weakening or strengthening) or explicitly overridden.

It is interesting to notice that only Eiffel supports the tag feature, which allows naming contracts for documentation purposes. This happens because only Eiffel was designed for automatic documentation. Eiffel is also the only language to support loop contracts and postcondition frame rules.

Regarding the "old" statement, Blue store the whole object "image" for postconditions, allowing it to be used as a regular object. Eiffel and Chrome use a different technique: it evaluates method call expressions (pre-calculating them), instead of storing the whole object.

All programming languages support invariants at the class level. Invariants are checked after method call return, but Chrome, Nemerle and Sather do not check them before method call, rendering them vulnerable to some inconsistencies issues. Since Nemerle supports the expose statement, an extra check is performed after exposure.

Contract recursion is a typical poorly documented facet of the contract evaluation infrastructure. Eiffel disables this to avoid infinite recursion, whereas Chrome authors do not address this issue and allow it to occur. Possibly the remaining solutions follow one of these two approaches, since these are the simplest approaches.

It is worth noticing that none of the languages currently addresses concurrency in contracts, since it is a known issue for some time [Meyer92].

Some programming languages, such as Blue, Chrome and Nemerle, do not support the check assertion statement, perhaps underestimating its interest. Other mainstream languages, such as C, Java and PHP support this as the only feature related to contracts.

Assertion failures are treated very differently by these languages. The crudest strategy, program termination, is used by Blue, Lissac, Sather, and C. The most useful solution though, is throwing an exception, which is the approach used by Eiffel, Chrome, D, Nemerle and Java. Nemerle also allows the possibility of assigning user-defined handling. PHP, with its known flexibility, allows any of these approaches.

Most programming languages support some form of contract disabling mechanism. These solutions typically support a coarse grain of simple assertion enable/disable. Eiffel, though, allows a finer grain of module and assertion type. Lissac offers an alternative approach, by allowing the assignment of priorities to contracts, and consequently defining the granularity by a check level. Granularity control can be achieved simply by compilation flags. Eiffel allows the configuration to be explicit through a file, whereas Java and PHP provide a runtime API.

Only Eiffel and C are standardized. Lissac, Nemerle, and Sather are free or open-source licensed, whereas Chrome is proprietary. The D programming language has both implementations.

Feature		Native support							Other		
		Eiffel (ECMA-367)	Blue (1.0)	Chrome (1.5.3)	D (1.0)	Lissac (0.2)	Nemerle (0.9.3)	Sather (1.1)	C (ANSI)	Java (1.5)	PHP (5.1)
Contract specification	Executable code	√	√	√	√	√	√	√			
	Metadata										
	Comment										
Contract identification	First-class	√	√	√	√	√	√	√			
	Naming convention							√			
	Comment										
Contract placement	Class	√	√	√	√	√	√	√			
	Interface										
Contract inheritance	Findler										
	Classic		√		√						
	ECMA	√									
	Other					√					
Local variables					√	√					
Result variable		√			√	√	√	√			
Assertion tag name		√									
Old state	Primitive types										
	Reference										
	Cloning	√	√			±					
	Expression		√	√		√					
First-order logic	Universal quantifier										
	Existential quantifier										
	Conditional operator	√	√								
Invariant types	Class	√	√	√	√	√	√	√			
	Method										
	Attribute										
Invariant evaluation time	Before	√	√		√	√					
	After return	√	√	√	√	√	√	√			
	After exception										
	After exposure						√				
Evaluation visibility	Public	√	±	√	√	√	√	√			
	Protected			√							
	Package										
	Private										
Contract recursion	Enabled		?	√	?	?	?	?			
	Disabled	√									
	Safe										
Side-effects free											
Concurrency											
Check assertion		√			√	√		√	√	√	√
Loop contract		√									
Frame rules		√									
Assertion failure	Termination		√			√		√	√		√
	Exception	√		√	√		√			√	√
	Warning										√
	User-defined handling						√				√
Disabling granularity	Simple	√		√	√	√		√	√	√	
	Per module	√								√	√
	Per assertion type	√									
	Priority					√					
Configuration	Run-time API										√
	File	√									
Automatic documentation		√									
Implementation license	FOSS		√		√	√	√	√		√	√
	Proprietary			√	√						
	Standard	√							√		

Figure 6.1: Comparison of languages with native support

6.2.2 Language extensions for Java

Oak, ContractJava, Handshake, KJC.

Since all solutions are language extensions (Figure 6.2), the only instrumentation approach available is the use of a modified compiler, since contracts are first-order constructs.

In Oak, contracts where only available for classes. In ContractJava and KJC contracts can be placed in either classes or interfaces. Handshake defines a new module – the `contract` – in which contracts are placed.

In Oak, contract inheritance was simple, without the contravariance/covariance issues. In Handshake and KJC, the “classic” approach to the Liskov substitution principle is used, whereas ContractJava introduced Findler's proposal for the substitution principle.

Handshake and KJC follow the “standard” approach of support class invariants. Oak supported method and attribute invariants. ContractJava did not support invariants at all. All of these language extensions check invariants before and after method call. KJC also checks invariants after method termination by exception.

Oak allowed contract recursion without limitation, while KJC disallows such feature. The behaviour of the remaining extensions on this issue is not documented.

Finally, none of these languages supports a considerable set of features, perhaps due to its research nature, namely: side-effects freeness, concurrency, check assertion, loop contracts, frame rules, disabling mechanisms, unit testing integration, automatic documentation, GUI support.

Feature		language extensions			
		Oak not released	ContractJava no public release	Handshake no public release	KJC 2.2D - ??????
Required JVM		n/a	?	1.1	1.4
Instrumentation approach	Preprocessor				
	Modified compiler	√	√	√	√
	Specialized class loader				
	JSR 269				
	AOP				
Specification language	Base language	√	√	√	√
	Different language				
Contract specification	Executable code	√	√	√	√
	Metadata				
	Javadocs				
	Comment				
Contract identification	First-class	√	√	√	√
	Naming convention				
	Comment				
Contract placement	Class	√	√		√
	Interface		√		√
	External			√	
Contract inheritance	Findler		√		
	Classic			√	√
	Ecma				
	Other	√			
Local variables					
Result variable				√	√
Assertion tag name				√	
Old state	Primitive types				
	Reference				
	Cloning				√
	Expression				√
First-order logic	Universal quantifier				
	Existential quantifier				
	Conditional operator				
Invariant types	Class			√	√
	Method	√			
	Attribute	√			
Invariant evaluation time	Before	√		√	√
	After return	√		√	√
	After exception				±
	After exposure				
Evaluation visibility	Public	√		√	√
	Protected	√		√	√
	Package		√	√	√
	Private				
Contract recursion	Enabled	√			
	Disabled				√
	Safe				
Side-effects free					
Concurrency					
Check assertion					
Loop contract					
Frame rules					
Assertion failure	Termination				
	Throwable	√	√	√	√
	Warning				
	User-defined handling				
Disabling granularity	Simple				?
	Per module				
	Per assertion type				
	Priority				
Configuration	Run-time API				
	File				
Unit testing integration					
Automatic documentation					
GUI support					
Implementation license	FOSS				√
	Proprietary				
	Standard				

Figure 6.2: Comparison of language extensions for Java

6.2.3 "Traditional" ad-hoc solutions for Java

C4J, iContract2, Jass, Jcontract, jContractor, JML Tools, Modern Jass, STClass.

From this section forward, *ad-hoc* solutions for Java are discussed, namely using "traditional", i.e. non-Aspect-Oriented, approaches (Figure 6.3). There are various techniques. iContract2, Jass, and STClass use a pre-processor, for source code transformation. Jcontract and JML tools use a modified compiler. Finally, C4J and jContractor use a specialized class loader.

Most of these solutions use a different language for specifying contracts, in the form of comments. The exceptions are jContractor, which uses the base language executable code; and C4J, which uses both, mixing metadata and executable code.

Most solutions allow placing contracts in classes and interfaces, except for Jass and Jcontract. In C4J though, contracts are placed in an external class, which can be applied to either an actual class or interface.

The most common solution for the "old" construct is to use object cloning for this feature. JML tools and STClass follow Eiffel, in the sense that it is possible to evaluate an arbitrary expression. In C4J, the storage is explicit, which means that typically only primitive types are stored. Additionally, some solutions also are able to store the reference, namely C4J and iContract2.

Most solutions fully support the listed first-order logic quantifiers and operators, with the exceptions of C4J and Jcontract.

The invariant evaluation time is poorly documented for traditional *ad-hoc* solutions. C4J only evaluates after method return, Jass evaluates before and after method execution, and iContract2 also evaluates contracts if the method execution terminates by exception.

Regarding evaluation visibility, solutions range incrementally from private, to protected and package.

Contract recursion is also not thoroughly documented. Only iContract2 implements safe recursion, while the remaining solutions disable it or simply do not document it.

Jcontract is the only solution that claims to fully address concurrency, whilst iContract2 and Jass only partially address it.

Some Eiffel features, such as loop contracts and frame rules, are present in Jass and JML tools.

JML tools and Jcontract allow issuing warnings for assertion failure, instead of using a `Throwable` object. The latter also supports user-defined handling.

As to disabling assertions, both jContractor and Jcontract support all the granularity (simple, per assertion type, and per class) present in Eiffel. The latter is also the only one in this set to support a configuration file.

Unit test integration is present in Jcontract, JML tools and STClass, acknowledging the synergy between these two approaches.

Finally, Jcontract is the only solution in this set without a public release. Instead, it has a proprietary license.

Feature		“traditional” solutions							
		C4J 2.5.3 – 20061122	iContract2 1.0.0 – 20061123	Jass 2.0.14 – 20060707	Jcontract no public release	jContractor 0.1 – 20030202	JML Tools 5.3 – 20060517	Modern Jass 0.2 – 20070519	STClass 4.0rc3 – 20061016
Required JVM		1.5	1.2	1.3	1.3	?	1.5	1.6	1.5
Instrumentation approach	Preprocessor		√	√					√
	Modified compiler				√		√		
	Specialized class loader	√				√			
	JSR 269							√	
	AOP								
Specification language	Base language	√				√		√	
	Different language	√	√	√	√		√		√
Contract specification	Executable code	√				√			
	Metadata	√						√	
	Javadocs		√		√				√
	Comment			√			√		
Contract identification	First-class							√	
	Naming convention	√				√			
	Comment		√	√	√		√		√
Contract placement	Class		√	√	√	√	√	√	√
	Interface		√			√	√	√	√
	External	√							
Contract inheritance	Findler	√		√					
	Classic		√			√	√		√
	Ecma							√	
	Other								
Local variables		√		±		√	√	√	
Result variable		√	√	√	√	√	√	√	
Assertion tag name				√					√
Old state	Primitive types	√						√	
	Reference	√	√					√	
	Cloning		√	√		√			
	Expression						√		√
First-order logic	Universal quantifier		√	√		√	√	√	√
	Existential quantifier		√	√		√	√	√	√
	Conditional operator		√			√	√		√
Invariant types	Class	√	√	√	√	√	√	√	√
	Method								
	Attribute						√	√	
Invariant evaluation time	Before		√	√	?	?	?	√	?
	After return	√	√	√				√	
	After exception		√					√	
	After exposure								
Evaluation visibility	Public	?	√	√	?	√	√	√	√
	Protected		√	√		√	√	√	
	Package		√				√	√	
	Private							√	
Contract recursion	Enabled	√		?					?
	Disabled				√	√	√	√	
	Safe		√						
Side-effects free					?		√	±	√
Concurrency			±	±	√				
Check assertion		√		√	√	?	√		
Loop contract				√			√		
Frame rules				√			√		
Assertion failure	Termination								
	Throwable	√	√	√	√	√	√	√	?
	Warning				√		√		
	User-defined handling				√			√	
Disabling granularity	Simple	√	√	√	?	√			√
	Per module				√	√			
	Per assertion type		√		√	√			
	Priority								
Configuration	Run-time API								
	File				√				
Unit testing integration					√		√		√
Automatic documentation				√			√		√
GUI support		√			√		√	√	
Implementation license	FOSS	√	√	√		√	√	√	√
	Proprietary				√				
	Standard								

Figure 6.3: Comparison of "traditional" ad-hoc solutions for Java

6.2.4 Aspect-Oriented ad-hoc solutions for Java

DbC4J, Barter, ConFA, Contract4J, OVal, SpringContracts.

While the previous subsection discussed “traditional” *ad-hoc* solutions for Java, the following solutions (Figure 6.4) approach the problem through Aspect-Oriented Programming. Although present in the comparison table, DbC4J comparison is postponed to the next subsection.

Even though they use aspect-oriented technologies, the Barter and ConFA solutions still require a preprocessor. For Barter, the reason for this is the lack of standardized metadata mechanisms for Java at the time of its development; for ConFA the reason is the complexity of the solution architecture.

It is interesting to note that all the analysed solution use contract as first-order constructs in both classes and interfaces. Contract polymorphism implementation is less consensual: each solution uses a different approach.

First-order logic constructs are only supported in SpringContracts.

The recursion issue is not treated or documented in any of these solutions. The absence of side-effects is dealt only in OVal, while the check assertion is only available at Contract4J.

User-defined handling for assertion failures is somewhat popular in these solutions, supported by Barter, OVal, SpringContracts, and Contract4J.

None of these solutions provide an assertion disabling mechanism with the flexibility of Eiffel. SpringContracts supports a file configuration for such mechanism, while Contract4J supports both file configuration and run-time configuration.

Feature		aspect-oriented solutions					
		DbC4J 1.0 – 20071130	Barter 0.2.0 – 20020807	ConFA 20031122	Contract4J 0.7.0 – 20061231	OVal 0.8.0 – 20061215	SpringContracts 0.2 – 20061123
Required JVM		1.5	1.3	?	1.5	1.5	1.5
Instrumentation approach	Preprocessor		√	√			
	Modified compiler						
	Specialized class loader						
	JSR 269						
	AOP	√	√	√	√	√	√
Specification language	Base language	√		√			
	Different language		√		√	√	√
Contract specification	Executable code	√		√			
	Metadata				√	√	√
	Javadocs						
	Comment		√				
Contract identification	First-class		√	√	√	√	√
	Naming convention	√					
	Comment						
Contract placement	Class	√	√	√	√	√	√
	Interface		√	√	√	√	√
	External						
Contract inheritance	Findler			√			
	Classic						√
	Ecma	√					
	Other		√		√		
Local variables		√					
Result variable		√	√	√	√	√	√
Assertion tag name							
Old state	Primitive types	√			√		
	Reference				√		
	Cloning	√		√			√
	Expression			√	√		√
First-order logic	Universal quantifier						√
	Existential quantifier						√
	Conditional operator	√					√
Invariant types	Class	√	√	√	√		√
	Method						
	Attribute				√		
Invariant evaluation time	Before	√	√	√	√		
	After return	√	√	√	√		
	After exception						
	After exposure						
Evaluation visibility	Public	√	√	?	√	√	√
	Protected	√				√	
	Package	√				√	
	Private					√	
Contract recursion	Enabled		?		?		±
	Disabled	√					±
	Safe						
Side-effects free						√	
Concurrency							
Check assertion		±			±	±	±
Loop contract							
Frame rules							
Assertion failure	Termination						
	Throwable	√	√		√	√	√
	Warning						
	User-defined handling		√		√	√	√
Disabling granularity	Simple	√			√		√
	Per module	√	√		?		√
	Per assertion type	√	√		√		
	Priority						
Configuration	Run-time API	√			√		
	File	√			√		√
Unit testing integration							
Automatic documentation							
GUI support							
Implementation license	FOSS	√	√	?	√	√	√
	Proprietary						
	Standard						

Figure 6.4: Comparison of aspect-oriented ad-hoc solutions

6.2.5 Overall comparison with DbC4J

Finally, the prototype developed during this thesis must be compared with remaining solutions.

The first divergence that one finds when comparing DbC4J with the remaining AOP-based solutions, is the use executable code instead of annotations. This brings in some limitations which the other solutions avoid, namely the need of naming conventions in the assertions, and the inability to write contracts in interface modules. On the other hand, DbC4J is the only solution to implement the substitution principle as defined in the Eiffel Ecma standard. Finally, only DbC4J and Contract4J support all the assertion checking disable features present in Eiffel (full, class, and assertion type), both through a configuration file or a run-time API.

DbC4J shares the main advantage of most other AOP-based solutions: the fact that no additional tool or step is required for using the library. Another advantage is, again, the use of file or run-time configuration. On the other hand, many of these solutions offer contracts for interfaces, as well as first-order logic quantifiers, which DbC4J does not, at this stage. Also, we do not provide native support for automatic documentation or GUI, unlike a few of the others, namely Jcontract and Modern Jass. Other solutions support these features, but mostly as separate independent projects. Finally, unlike some solutions (ContractJava, Handshake, Jcontract, and ConFA), our prototype is publicly available.

6.2.6 Third-party evaluation

Plösch proposed a set of metrics to evaluate assertion support for the Java programming language [Plösch02]. As such, it is interesting to perform an evaluation to DbC4J using third-party metrics. Unlike our previously defined criteria (in section 6.1), Plösch's proposal is at a coarser granularity, yielding a more compact evaluation.

Each feature is evaluated in a four value scale: `1` (complete), `3` (partial), `5` (very incomplete), and `ns` (not supported). Table 6.1 summarizes the evaluation. These are the proposed metrics:

- Basic assertion support (BAS)
  - ➔ BAS-1 (Basic assertions) – Since Java 1.4 and later support the assertion facility, we do not feel to be necessary to introduce a specific mechanism for such feature.
  - ➔ BAS-2 (Preconditions and Postconditions) – In this area, pre/postconditions are fully supported, but do not guarantee side-effect free assertions.
  - ➔ BAS-3 (Invariants) – Same as BAS-2.
- Advanced assertion support (AAS)
  - ➔ AAS-1 (Enhanced assertion expressions) – Access to the original state of the object or arbitrary (primitive type) variables is supported, but not arbitrary expressions. Access to the parameter values in postconditions is also supported.
  - ➔ AAS-2 (Operations on collections) – No support for first-order logic other than the implication operator.
  - ➔ AAS-3 (Additional expressions) – The "expose" mechanism is supported, as well as a runtime API for configurability.
- Support for Behavioral subtyping (SBS)
  - ➔ SBS-1 (Interfaces) – It is possible to add contracts to interfaces, but only through the intertype declaration feature of AspectJ. This means it does not follow the subtyping principle.
  - ➔ SBS-2 (Correctness I) – Precondition weakening and postcondition strengthening in subcontracts is imposed.
  - ➔ SBS-3 (Correctness II) – The rules for behavioural subtyping are the same as specified in

the Eiffel standard.

- Runtime monitoring of assertions (RMA)
  - ➔ RMA-1 (Contract violations) – Runtime exceptions are used. While there are no monitoring features, since we believe it is up to the programmer to decide what to do in these situations, it would straightforward to implement this.
  - ➔ RMA-2 (Configurability) – Assertion enabling/disabling is supported, by assertion type and class. It would be interesting to support package level granularity, but method granularity seems to be "overkill".
  - ➔ RMA-3 (Efficiency) – There is a significant overhead of memory and (especially) processing usage, as it will be discussed in the next section.

Table 6.1: Plösch evaluation summary on DbC4J

BAS-1	BAS-2	BAS-3	AAS-1	AAS-2	AAS-3	SBS-1	SBS-2	SBS-3	RMA-1	RMA-2	RMA-3
3	3	3	3	ns	1	3	1	1	1	3	5

6.3 Performance

In this section, some performance considerations are discussed. First, the measurement method is presented. Second, the quantitative analysis' results of the measurements performed are shown. Finally, a qualitative analysis is performed.

6.3.1 Experiment subject

The experiment subject is a small text-mode Java application, which basically reads a Comma-Separated Value (CSV) text file, and inserts a pair of values into a hash table. The application is in two files: `Dictionary.java` (the supplier class) and `Main.java` (the client class). Three versions of the program were written: the first version is an optimistic implementation, in which contracts are only comments ("`foobar`"); in the second version contracts are in-lined assertions using Java's 1.4 `assert` facility ("`foobar-jaf`"); finally, the last version writes the contracts using DbC4J ("`foobar-dbc`"). While the second solution may seem unreasonable, it could be attained by using a specification language for contract writing and a code transformation tool to insert the `assert` statements.

6.3.2 Measurement method

The experiment was performed with the software described in Chapter 1, under the Fedora Core 4 GNU/Linux operating system. Twelve executions were made; the best and worst were removed; the resulting time value is an average of the ten remaining values. The time values are measured using the `System.currentTimeMillis()` method.

6.3.3 Quantitative results

The quantitative results of the experiment are summarized in Table 6.2. The measurement units used are: milliseconds (ms), lines of code (LOC)[1], and bytes (B). From this table, it is possible to observe some interesting facts.

First, the DbC4J version performance is about 6000 times slower than the original version. An extra test was performed, running the DbC4J version with all assertion check disabled: the execution time was 49 milliseconds, about 7 times slower than the original. This confirms that performance degradation is essentially caused by the infrastructure Java code rather that the AspectJ code.

1 Lines of code include comments, metadata, but not blank lines.

Secondly, execution time in the JAF version is similar to the original version[2]. This shows that the assert language feature is optimized by the Java compiler and virtual machine.

	Foobar			Foobar-jaf			Foobar-dbc		
	Client	Supplier	All	Client	Supplier	All	Client	Supplier	All
Execution time (ms)	-	-	7	-	-	7	-	-	45082
Source code size (LOC)	54	122	177	63	122	185	54	172	226
Source code size (B)	1464	2932	4396	1827	2932	4759	1464	3942	5406
Bytecode size (B)	2340	2193	4533	2497	2193	4755	6041	15696	23819

Table 6.2: Performance experiment quantitative results

Finally, it is possible to see the increase on code size when using contracts, in the cells shaded in grey. While in the JAF version the increase is on the client side, in the DbC version the increase is on the supplier side. This means the DbC version would scale better. Also note that in case of "foobar-dbc" experiment the total bytecode produced (shaded in yellow) is larger than the sum of client and supplier classes' bytecode, since each closure produces a class.

6.3.4 Qualitative analysis

It is a challenge to obtain an unbiased performance experiment. This one is not an exception. The experiment involves some limitations:

- *application size* – larger applications incur in less overhead;
- *design patterns* – the more "inlined" the code, the less method calls, and the less overhead; the same is true when using iteration instead of recursion;
- *number of assertions* – more assertions mean more executed code;
- *configuration* – disabling assertion checking for certain types or classes greatly improves performance.
- *level of polymorphism* – deeper inheritance trees involve more effort on the algorithm for contract evaluation;

The "foobar" application is a CPU-bound application. Most "real-world" applications involve a significant use of I/O calls, network connections, database access, user interface, etc. As such, it is not uncommon for an application to spends very few time (e.g. less than 10%) in CPU processing. Thus, CPU execution time is only a small fraction of the real execution time of an application.

Why is DbC4J performance so low? The largest bottleneck is the use of reflection, an expensive mechanism. We could decrease this bottleneck by reducing the use of reflection, either by algorithm optimisations, stricter pointcuts, or simply by changing the library user syntax. The second bottleneck is AspectJ's overhead. In spite of continuous performance improvements in AspectJ's releases, there is little that can be done here, other than limiting the scope of the pointcut expressions. Finally, we must acknowledge that one of the biggest advantages of incorporating Design by Contract in the Java language instead of using an *ad-hoc* solution, is the possibility of using first-order artefacts, which could be used to perform low-level compiler/JVM optimisations (similarly as it happens with the *assert* facility).

6.4 Evaluation of the proposed requirements

In chapter 4, the objectives for the prototype developed in this thesis were proposed, in the form of a set of informal requirements. Lets review and evaluate them:

1. *The prototype should be in the form of a library, and should be simple to use by an*

2 In this experiment, it is undistinguishable, but on a larger CPU-bound program, the overhead would be visible.

"average" Java programmer, that is, one which is not familiar with AspectJ or AOP in general.

- ➢ The prototype is available in the form of a JAR library, which is the standard approach for distributing libraries in Java. The library is dependent on the AspectJ runtime library, which is also distributed as a JAR file. Other than that, there are further procedures. This means that this objective was accomplished.

2. *The library should support the core features present in the Eiffel language: preconditions, postconditions, class invariants, contract inheritance, and the "result" and "old" special variables.*

   - ➢ The prototype supports the core features of Design by Contract in Eiffel. Some other features are not supported, namely the only clause of postconditions, which allows to define "frame rules"; and loop variants and invariants. Other features not available in Eiffel but supported, are the "expose" areas and the runtime contract configuration. As such, this objective was accomplished.

3. *The contracts are to be written in standard Java code, although the library is implemented in an aspect-oriented language.*

   - ➢ The class contracts are written using only standard Java code. Interface contracts can only be written by explicitly using the intertype declaration feature of AspectJ. Since this is not a core feature of the library, the objective was partly accomplished.

4. *The library should be pluggable, meaning that an application written with the library should compile in the absence of the library, and display the same functional behaviour.*

   - ➢ While using the core features of the library, the unplugging procedure involves only removing the library from the project. However, if the "expose", runtime configuration, or "old" mechanism features are used, the application is coupled with the library. The workaround procedure consists in replacing the main library with a "mock" one. Since the procedure is straightforward and does not involve writing code, the objective was partly accomplished.

5. *Finally, a mechanism for restraining contract verification should be available, with some of level of granularity, in terms of modules and/or assertion types.*

   - ➢ There are two mechanisms for restraining contract verification, either statically using a configuration file, or dynamically using a runtime API. Since this mimics the granularity level of Eiffel LACE files, this objective was accomplished.

6.5 Prototype technical details

The developed prototype is a relatively small sized program[3], as shown in Table 6.3 and Table 6.4.

Table 6.3: Program size in bytes

Form	Size (KiB)
Bytecode (JAR with compression)	41
Bytecode (class files)	106
Source code	63

3 In this context, lines of code do not include examples, unit tests and configuration files. Kibibyte (KiB) is an ISO unit for base 2 value measuring.

Table 6.4: Program size in lines of code

	Size (LOC)	Ratio (%)
Full code	2166	100
Java code	1532	71
AspectJ code	634	29

6.6 Further ideas

Most of the proposed ideas aim to address the current implementation limitations in the library, as explained in Chapter 4. Also, the development of some auxiliary tools is proposed.

6.6.1 Library features

Old mechanism. The current old mechanism implementation is cumbersome. It would be interesting to have an implementation without explicit store. A naïve solution would be to store the state of the object before any execution, but this would be far too resource demanding. The issue is that we need a more explicit pointcut. It would be interesting to research if the current pointcut designators are expressive enough to be able to capture the call of a method with a postcondition assertion that uses the old

```
class ClientPool {

   private int numberOfClients;

   @AtomicContract
   public void method(...) {
      method body
   }
   public boolean preMethod(...) {
      return numberOfClients <= 100;
   }
}
```

Listing 6.1: A possible solution for concurrent programming with contracts

mechanism. If not, an *heuristic* could be used instead, such as a pointcut that captures an execution of a method with a postcondition in a Cloneable class, possibly using the hasMethod() pointcut designator, an undocumented experimental feature on AspectJ 1.5.

Visibility enforcement and absence of side-effects. As discussed in [Meyer97], preconditions are to be seen by clients and as such should only use public features. In the current implementation, the visibility scope is not enforced. This could be achieved through an aspect with static cross-cutting, by issuing a compile-time warning or error, if a precondition method uses a non-public method or attribute. Such aspect would not interfere with the current design of the library. A solution to the absence of side-effects in contracts could be developed using an similar solution.

Interface support. Interface support for the library works as a by-product of *ad-hoc* mixins obtained through AspectJ's inter-type declarations. It would be interesting to research how we can add explicit support for interface contracts can be added, namely in terms of polymorphism. The theory for such an endeavour is discussed in more detail in the following chapter.

First-order logic. First-order logic, such as in OCL [OMG05], can help improving the expressiveness of some contracts. The current library design includes a ContractLogic class, but it only implements the implication operator. It would interesting to provide forAll() and exists() methods, to implement the universal and existential quantifiers, as shown in Figure 6.5. The predicate parameter forces the

```
class Point4D {
    double x, y, z, t;

    public void setX(int x) {
        method body
    }
    public boolean preSetX() {
        ContractMemory.remember();
        return true;
    }
    public boolean postSetX() {
        Point4D old = ContractMemory.old();
        return
            y == old.y && z == old.z && t == old.t;
    }
}
```

```
class Point4D {
    double x, y, z, t;

    public void setX(int x) {
        method body
    }
    @Only("x")
    public boolean postSetX() {
        return true;
    }
}
```

Listing 6.2: Current implementation (left) and possible solution for the frame problem (right)

programmer to write a specific class, since it is not possible to write an anonymous function[4] in Java (a *lambda function*), as in Eiffel ("agents"), C# ("delegates"), or functional programming languages.

Exceptional postconditions. Some Design by Contract solutions offer a possibility to define postconditions for situations in which methods terminate through an exception. While the implementation of such feature is not complex (we could use after throwing advices), we think it does not add any value to library at its present stage.

Concurrent programming. The current implementation has no support for concurrent programming. By this we mean that, while one may declare a method as syncronized, there is no guarantee that the respective assertions are not being executed concurrently with another method that will invalidate them, making the method context invalid, without being detected. The issue is that method execution and respective contract checking are not *atomic*. One solution to this problem would be by replacing before/after advices in aspects with around advices whose body (or part of it) is inside a syncronized statement, to enforce atomicity. Since in practise this would be too restrictive, it would be better to allow the programmer to decide which methods need to be atomized with contracts (e.g. using an annotation, as shown in Listing 6.1), and make the advices perform an if statement for a synchronized and a non-synchronized branch.

Frame conditions. Sometimes it is useful to express that a method execution will only modify a certain set of class variable members. While this could be performed in postconditions by simply using the old mechanism, this is not a scalable solution (see Listing 6.2), since there is a need to enumerate which variables do not change, instead of which variables are allowed to change (which is not the same, if new members are added). To support this, Eiffel offers the only clause for postconditions, and JML supports the so called *frame conditions*. Supporting such mechanism in the solution proposed in this thesis is not simple, since assertions written in Java executable code are *imperative*, not *declarative*. A possible solution is the use of annotations in postconditions, as show in Listing 6.2. Under the hood, a mechanism similar to old would be used to store the variables and compare them later. This would suffer the issues of the old mechanism, and a new one, since this would introduce non-executable assertions, which are not visible in debugging, tracing, etc. Finally, this mechanism could not actually guarantee that variables did not change, it could only guarantee that they would be the same at method entry and exit (that is, they could change and then return to the original value, without being detected). However, for most uses it is enough the semantic that guarantees that the variables *apparent* value did not change, similarly to invariant checking.

Inner classes. Contract semantics for *inner classes* (and enclosing classes) were not addressed in this dissertation. Inner classes should behave as any regular classes regarding contracts, with the difference that they can affect the outer class' state, and as such, invalidate its invariant. The solution then would

4 However, it is possible to write anonymous classes.

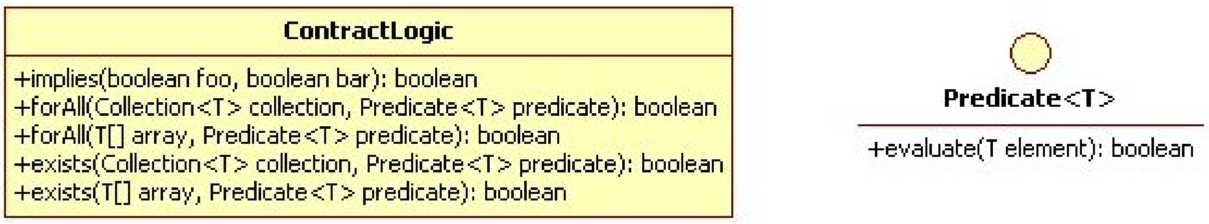
ContractLogic
+implies(boolean foo, boolean bar): boolean
+forAll(Collection<T> collection, Predicate<T> predicate): boolean
+forAll(T[] array, Predicate<T> predicate): boolean
+exists(Collection<T> collection, Predicate<T> predicate): boolean
+exists(T[] array, Predicate<T> predicate): boolean
Predicate<T>
+evaluate(T element): boolean

Figure 6.5: A possible improvement over the ContractLogic class

be simply to evaluate a class' invariant any time an inner class of it invokes a method, similarly as done in JMSAssert [MMS06].

Abstract methods. There is no special semantics for abstract methods' contracts or abstract contracts. However, the concept of *abstract contracts* is discussed by Meyer [Meyer97], and can be solved using programming techniques.

6.6.2 Auxiliary tools

Documentation generator. Eiffel supports documentation generation, through the short tool, which generates a class skeleton with interface and contracts. This is feasible for Java (by using a parser such as JavaCC [Norvell02]), but extending the javadoc tool is more practical. It would be interesting to assess the changes needed to the javadoc tool, in order to recognize contract methods as first class artefacts. Unlike previous features, this would be limited to the specific version of Java, that is, new releases of the Java language would imply changes in the tool.

GUI. In order to improve productivity, it would be interesting to have some GUI support, extending the Eclipse IDE's source/refactoring support, in a form of a plug-in.

Chapter 7 Contracts for Aspect-Oriented Design

This chapter is based on the work performed during my MSc degree scholar part, namely in the course project "*Tópicos Avançados de Engenharia de Software*" (Advanced Topics on Software Engineering). The work has been expanded, refined and published as a research paper into an international conference [Agostinho08]. In the context of this dissertation, it stands as a complement to the actual thesis.

In this chapter, it is discussed how the Design by Contract approach from Object-Oriented Software Development can be extended and applied to Aspect-Oriented Software Development. The base for this discussion is the Java programming language. On one hand, it is shown how the current Object-Oriented Design by Contract can be modified to address aspect advising. On the other hand, it is discussed how to actually extend the Design by Contract approach for the aspect modules. Next, some brief considerations about associated Java Aspect-Oriented Java extensions are presented, namely on AspectJ, CaesarJ, and FuseJ. Finally, some issues on how to adapt the prototype for described purposes are presented.

7.1 Aspect-aware Contracts

We have found that that "classical" object-oriented Design by Contract is not suited for aspect-orientation, for reasons that we will now explain. Then, we will address the issue of adapting Design by Contract so that it fits in Aspect-Oriented Design.

7.1.1 Classic View

The sequence flow of calling a method on a Design by Contract solution is usually as follows:

1. the method under contract is called;
2. class invariants are evaluated;
3. method preconditions are evaluated;
4. method body is executed;
5. method postconditions are evaluated;
6. class invariants are evaluated.

With a failure of an assertion, an exception is thrown, and blame is assigned to the responsible module (class), as shown in Table 7.1. The client is the class that calls the method, and the supplier (also called server) is the class that offers the method.

Table 7.1: Classical responsibilities in Design by Contract

	client	**supplier**
precondition	√	
postcondition		√
invariant		√

7.1.2 Issues

Now, what would happen if an aspect is advising the supplier object? More accurately, in which step of execution would the aspect advice be weaved? Behaviour introduction between steps 3 and 4 or between steps 4 and 5 could easily break the class contract, and no blame would be assigned.

The issue here is where to "draw the line" for aspect behaviour modification. According to Filman et al. [Filman01], one of the main characteristics of AOP is the *obliviousness* between classes and aspects. Following this premise there should not be restrictions on aspect behaviour introduction. However, this is not consensual in the Aspect-Oriented Software Development community [Dantas06]; recent work [Wampler07] claims that the obliviousness characteristic is too broad and naïve, and proposes that it should be restricted to *non-invasiveness*. This way, there are things that aspects should be restricted to do, namely breaking class contracts.

7.1.3 A Solution

Our proposal is that contracts should be checked twice: before and after aspect advising. For this, AspectJ-like advices `before`, `after`, and `around` are being considered. While the first two are straightforward, the latter is more complex. In `around` advices with a call to `proceed()` the code would function as a `before` advice until the call, and as an `after` advice after the call. On the other hand, in `around` advices with no `proceed()` call, assertion evaluation would function as the advice is the actual method body.

As such, the sequence flow of calling a method on a Design by Contract "aspect-aware" solution would be something like this:

1. the method under contract is called;
2. class invariants and method preconditions are evaluated;
3. `before` advices and pre-`proceed` code from `around` advices are executed;
4. class invariants and method preconditions are evaluated;
5. method body or a non-`proceed` `around` advice is executed;
6. method postconditions and class invariants are evaluated;
7. `after` advices and post-`proceed` code from `around` advices are executed;
8. method postconditions and class invariants are evaluated.

This sequence is a simplification, since it assumes only one aspect with only one advice. In general, with multiple aspects and advices, steps 3-4 and 6-7 must be repeated for each advice. This presents a major performance overhead. In section 6 we discuss ways to improve this. Anyway, it is generally not a good idea to have overlapping advices, as their interactions can be difficult to foresee.

Table 7.2 shows the blame assignment for this proposal. Notice that three new roles are defined, for each type of advice.

Table 7.2: Responsibilities in aspect-aware Design by Contract

	class		**aspect advice**		
	client	**supplier**	**before**	**after**	**around**
precondition	√		√		√
postcondition		√		√	√
invariant		√	√	√	√

```
class WebStoreSales
{
   // …

   public void processFooSale(Sale obj)
      require {
         assert obj.getValue() > 0.00;
         assert obj.getValue() <= 50.00;
      }
   {
      // "foo" sale logic…
   }
}
```

```
aspect ThisWeekDiscount
{
   pointcut allSales(Sale sale):
      execution(void process*Sale(Sale))
      && args(sale);

   before(Sale sale) : allSales(sale)
      require {
         assert sale.getValue() > 0.00;
      }
   {
      double valueWithDiscount =
         sale.getValue() * 0.95;
      sale.setValue(valueWithDiscount);
   }
}
```

Listing 7.1: Web store example class (left) and aspect (right) with contracts

7.2 Contracts for Aspects

In the previous section we focused on how class contracts must be evaluated in the presence of aspects, but not on how actual contracts for aspects must proceed. In AspectJ-like AOP languages, aspects introduce new types of mechanisms, in terms of *dynamic cross-cutting* (pointcuts and advices) and *static cross-cutting* (intertype declarations).

7.2.1 Dynamic cross-cutting

Advices are obvious candidates for being contracted, since they are used for writing behavioral code in aspects, analogously to methods in classes. However, unlike methods, advices are not called by a client, but by the aspect. Preconditions restrict the situations where a method can be executed. For advices there is already a mechanism for specifying the situations where it can be executed: pointcuts. Thus, if we are not able to satisfactorily restrict the execution of an advice, then the problem resides on the lack of expressiveness of the Pointcut Designator language, not on the on the absence of a suitable mechanism.

While there is no need for advice assertions *per se*, in certain situations it might be useful that a before advice can run on a situation that would break a precondition, but modify the state of the object (or the parameters) in a way that preserves the contract. The inverse situation is also possible, method postconditions can be insufficient if the changes made by the after advices can invalidate them.

7.2.2 An Example

Consider a web store system where a sale is classified according to the amount of money spent by the client, and as such different methods are executed according to the buyer's profile. One way to implement this is to write a precondition on the value of the sale for each of these methods, in order to ensure the correct profiling. Now, suppose promotions and discounts are very volatile and developers decided to implement them using aspect before advices. A certain profile requires that the buy amount is inferior to €50, which is written as precondition. This week, all book buys have a 5% discount, which is implemented by a before advice. Executing the profile method with the value of exactly €51 would normally violate the precondition, but the discount would "fix" it, since the actual value would be €48.45. A similar example could be written for postconditions, using taxes and delivery expenses for sales.

7.2.3 Behavioural Subtyping

Wampler [Wampler07] proposes that the Liskov Substitution Principle [Liskov93] be extended for advices. The proposed extended principle ("advice substitution principle") states that advising a class follows the same rules than subtyping. Considering the AspectJ-like advice types, this means that: before and around advices can only maintain or *weaken* preconditions, after and around advices can only maintain or *strengthen* postconditions, and all advices must *maintain* the invariants of the class.

As a generalization, overlapping advices are substituted similarly to class inheritance. That is, if aspect X and Y advise class A (in the same join point), and if X advises before Y, then X must maintain or weaken Y's preconditions, in the same way that Y must maintain or weaken A's preconditions. That is, X's contract is a subcontract of Y's contract. The order is assured by composition ordering mechanisms of the AOP language.

Consequently, the sequence flow defined previously is refined in the following (steps 2 and 8 are changed):

1. the method under contract is called;
2. advice invariants and preconditions are evaluated;
3. before advices and pre-proceed code from around advices are executed;
4. class invariants and method preconditions are evaluated;
5. method body or a non-proceed around advice is executed;
6. method postconditions and class invariants are evaluated;
7. after advices and post-proceed code from around advices are executed;
8. advice postconditions and invariants are evaluated.

Responsibility assignment remains unchanged from Table 2.1. Note that the subcontract invariants are named advice invariants instead of "aspect invariants", since they are not invariants on the state of the aspect (if it has one), but invariants regarding a certain advice (or pointcut, if it is used in multiple advices).

Listing 7.1 exemplifies how extending the substitution principle to aspect contracts could solve the "Web store" problem presented earlier.

While simple, the extended principle raises some issues. On one hand, the assertions increase the coupling of the aspect with the classes, especially if they use private variables or methods. On the other hand, contracts implemented as aspects have been criticized [Balzer06], since one of the important characteristics of contracts is that they are an integral part of the classes interfaces. However, these are both know issues of Aspect-Oriented Programming, which can be mitigated through tool support, namely IDE visualization and documentation generation, similarly to the support offered by the AspectJ Development Tools (AJDT) for Eclipse [Colyer04].

True aspect polymorphism is non-existent in AOP solutions, since advices are unnamed[1]. As such, partial pre/postconditions are composed through the OOP inheritance "axis" and then through the AOP advising "axis". Advice subcontracting order is given by the AOP solution aspect precedence mechanisms, i.e. $pre_{A.foo()} \rightarrow pre_{B.foo()}$.

7.2.4 Mixins and multiple inheritance

A mixin [Bracha90]is a module with features meant to be inherited as module extension, not as type refinement. As such, the behavioural subtyping principle is not applicable for mixins. However, inherited methods must respect its respective pre/postconditions, as well as the destination class'

1 In some solutions, such as AspectWerkz and annotation-based AspectJ, advices are named, but only for maintaining compatibility with standard Java, the names are not used.

invariants.

Multiple inheritance only occurs in Java when using interfaces. If interfaces are allowed to have contracts, then is necessary to address the issue of multiple inheritance with the substitution principle. The Eiffel language solves this problem by forcing routine renaming in the occurrence of name clashes. Another solution [Findler01] is to generalize the behavioural subtyping principle, such that a subtype must conform to every super type subtyping rules. That is, if C is a subtype of A and B, and if method `C.foo()` is overriding/implementing `A.foo()` and `B.foo()`, then `C.Foo`'s precondition must be contravariant to `A.preFoo()`'s precondition and to `B.preFoo()`'s precondition. Postconditions and invariants follow analogous rules.

7.3 Language Specific Considerations

The previous discussion was based on AspectJ-like advices, but it did not specify the actual AOP language. In this section, we specialize for some AOP languages.

7.3.1 AspectJ

The aspect module in AspectJ [AspectJTeam03] is essentially an extension to the class module, in terms of metamodel [Han04]. As such, there is a need to verify OOP assertions in aspects. Pre/postconditions and invariants must be checked for aspect methods, including the aspect super types, since aspects can extend classes. The original Liskov Substitution Principle is applicable. Regarding constructors, there is no need to check preconditions, since aspects are instantiated by the system implicitly, and as such cannot be restrained in which situations are applicable.

Regarding the extended Liskov Substitution Principle, the declare precedence mechanism defines the aspectization order. A problem with this mechanism is that in the absence of precedence declaration for overlapping aspects, the ordering is arbitrary. This is quite problematic for a possible extension of the language.

AspectJ intertype declarations used with interfaces are an *ad-hoc* form of mixins, and as such follow the rules presented earlier.

7.3.2 CaesarJ

The CaesarJ language [Aracic06] does not feature a specific module for aspects, but features the virtual class module (called `cclass`), which functions both as class and aspect. As such, most of the said in last subsection for AspectJ applies for CaesarJ as well. The main difference is that, unlike in AspectJ, CaesarJ features several modes of deployment, including explicit deployment. That is, in CaesarJ “aspects” are explicitly instantiated (deployed) by “classes”. Therefore, CaesarJ “aspects” require contract evaluation for constructors (unless they are of `singleton` type, in which case they work just like in AspectJ). More, failures in “aspect” constructor preconditions will be blamed on the client, just like in a regular class.

The declare precedence mechanism of AspectJ is supported in CaesarJ. Intertype declarations are not supported, but instead all virtual classes (`cclass`) are mixins. This way, CaesarJ must follow the rules discussed before regarding mixin contracts.

7.3.3 FuseJ

FuseJ [Suvée06] is a Component-Based Software Development / Aspect-Oriented Programming language, whose authors claim that there is no need for an aspect module. More, they claim that choosing between a component and aspect compromises the evolution of a system. As such, they propose that cross-cutting concerns should be implemented as components, and that the composition itself can be performed in later stages of implementation, either as a regular component or as an aspect.

FuseJ introduces new modules to support this: *services*, a set of operations that a component will provide and can expect to be available by the environment where it will be deployed; and *configurations*, a set of *linklets*, a composition rule for mapping a set of components/services into a service. The component operations are expressed in regular Java interfaces.

The linklet mechanism is particularly interesting, as it features an optional when clause, which is used to specific preconditions regarding the mapping. A possible approach to extending FuseJ to Design by Contract could be adding an additional clause for postconditions. Invariants are not likely to be useful, since linklets link components, and as such the internal state of the classes that implement them is not available[2]. However, in order to correctly enforce assertions, contracts must part of interfaces and not classes. The extended Behavioural subtyping would work as defined earlier.

Since FuseJ aspectization is performed only on components, contracts among classes would proceed as in regular Java Design by Contract. Finally, FuseJ is focused on component composition, and as such does not support introductions or mixins.

7.4 Related Work and Discussion

7.4.1 Open Modules / Cross-cutting Interfaces

Open Modules [Aldrich05] is a module system proposal for aspects, which enables software evolution with AOP while maintaining the expected behaviour for the module's clients. This is accomplished by a design approach: advices which are external to the module can only operate on the exported (i.e. public) pointcuts of that module.

Cross-cut programming interfaces (XPI) [Griswold06] is an approach based on the separation of advices and pointcuts into different aspects, using only AspectJ constructs (in concrete, but can applied to any AspectJ-like language). One aspect contains the actual aspect implementation (i.e. the advices), based on the pointcut exposed by the "interface" aspect. A third aspect enforces that no aspect can break the XPI contract, by advising the code through pointcuts other than the provided in the "interface" aspect.

The Open Modules proposal is somewhat limiting, and thus is more suited for component based development. The XPI proposal on the other hand, is flexible, but verbose, an issue that could be addressed with the creation of language construct. Both of these proposals relate to our work in the sense that both address the issue of maintaining module behaviour in the presence of aspects, relying (either explicitly or implicitly) on contracts.

7.4.2 Pipa

Pipa [Zhao03] is a behavioural interface specification language for AspectJ. Pipa extends JML [Burdy05], a behavioural interface specification language for Java. The authors of Pipa developed a tool to transform AspectJ/Pipa code into Java/JML, in order to take advantage of the JML support tools. While this ensures a good tool support, it is doubtful how the results are to be presented, namely in terms of blame assignment.

The authors define a distinction between module-level specifications, assertions on the behaviour of advice, introductions, and methods; and aspect-level specifications, contracts on the aspect state. Pipa supports pre/postconditions (including frame conditions) on advices and introductions, as well as invariants on aspects. However, the semantics of aspect contract inheritance are justified, since the aspect contract inheritance appears to be covariant (thus violating Liskov Substitution Principle); on the other hand, the issue of extending Liskov Substitution Principle for advising relations is not addressed. Consequently, advice pre/postconditions appear to test the actual behaviour of the advice code, instead

2 Nevertheless, properties could be assumed using a JavaBeans-like convention that components with "getVar()" / "setVar()" methods have a "var" variable.

of the advised method code.

In conclusion, the Pipa proposal is more focused on fitting AspectJ tool support into JML than actually addressing the issues of extending Design by Contract to Aspect-Oriented Programming.

7.4.3 ConFA

Contracts for Aspects (ConFA) [Skotiniotis04] [Lorenz05] is a tool for adding Design by Contract support to Java, and adding Design by Contract support to Aspect-Oriented Programming. The actual infrastructure is based in Aspect-Oriented Programming. ConFA extends the Java and AspectJ syntaxes with support for Design by Contract. Moreover, this work presents in-depth study on the evaluation sequence flow and blame assignment, by characterizing aspects in three types (extending again the AspectJ syntax[3]): agnostic, obedient, and rebellious. ConFA authors recognize that aspect advices may temporarily violate contracts, but do not enforce the Liskov Substitution Principle.

The issue is that in ConFA, like in Pipa, advice contracts are seen as advice assertions *per se*, not as Liskov Substitution Principle-like overridden assertions of the advised classes. Finally, ConFA does not address around advices or (advice) invariants.

In conclusion, the ConFA proposal provides an interesting study on blame assignment, but does not address the Liskov Substitution Principle. The tool infrastructure is also cumbersome, since it involves DemeterJ [Hulten98] preprocessing of extended Java/AspectJ source code (where contracts are mapped into aspects), prior to the AspectJ weaving.

7.4.4 COW

Contract Writing language (COW) [Shinotsuka06] is a constraint language for AspectJ, using predicate logic. As such, it presents a verification approach called Weaving by Contract. The language is based on the contract module which restricts a certain class, and declares pre/postconditions and invariants for each method. It is also possible to use predicates, of which a large number of primitive predicates are already supplied.

However, this work does not address contract inheritance or (as acknowledged by the authors) intertype declarations. The infrastructure is somewhat complex, as it includes an AspectJ analysis library, a predicate repository, as well as an API for Java and GNU Prolog [Diaz07] interoperability. Moreover, the textual separation of contracts from modules violates of one the premises of Design by Contract [Meyer97] that contracts are inherently part of the modules. Finally, the work fails to relate their proposal with previous work, such as Pipa and ConFA.

In conclusion, while the approach is novel by fertilizing Aspect-Oriented Programming with predicate logic, in this author's opinion it fails to demonstrate the applicability of the proposed solution.

7.5 Prototype Integration

Neither “aspect-aware contracts” or “contracts for aspects” are implementable in this thesis prototype, not without compromising the prototype architecture. For aspect-aware contracts it would be necessary to have an aspect advising before every other advice, a kind of “meta-aspect”. Such is not currently possible in AspectJ, due to the inexpressiveness of the precedence setting mechanism. A possible solution would be making a pre-processing step to add the meta-aspect, somewhat similar to ConFA's solution. The around advices would be the most complex part, since the advice code would have to be split through the proceed statement, as well as distinguished from advices with no proceed. “Contracts for aspects” features would require an extra effort, since an advice ordering tree needed to be reified, for implemented the extending substitution principle.

The XPI proposal is the only applicable approach to our library prototype, since it is essentially a

3 AspectJ 1.5 or later would not wield this syntax change, since (metadata) annotations could be used.

design pattern. An interesting exercise would be to apply the XPI pattern along with our prototype in a case study.

7.6 Summary

When incorporating Design by Contract in Aspect-Oriented Software Development, two issues must be addressed: how class contract checking must deal with aspect interaction, and how to have contracts for aspects. Class contract checking deals with aspect interaction by adding redundant checks between class code and aspect code, in order to correctly "blame" the faulty module. Regarding aspect contracts, it is not clear whether it makes sense for aspects to have contracts *per se*, given their nature. However, it seems interesting that contract checking can be relaxed for aspect advising. An existing proposal for extending the subtyping principle to incorporate aspect advising is discussed. Also, mixins and multiple inheritance in general raise some issues worth discussing.

Regarding language specific considerations, it is possible to conclude that AspectJ and CaesarJ are not very different. The differences reside in the deployment mechanism of CaesarJ which adds constructor assertions for aspects, and the fact that this language has a full mixin mechanism, unlike AspectJ's *ad-hoc* mechanism. FuseJ, as a strongly Component-Based solution requires a lighter approach, for aspectization is achieved by composition mechanisms.

Currently, there is significant research on addressing some issues, but no solution completely addresses them. Also, in technical terms, the implementations are rather cumbersome, with complex architectures.

Finally, the possible integration of these ideas in the developed prototype is discarded, as it would compromise the current architecture and underlying philosophy.

Chapter 8
Conclusions

This dissertation proposed an Aspect-Oriented Programming infrastructure for implementing Design by Contract approach in the Java programming language.

The Design by Contract approach is a prominent feature of Eiffel, but its full-fledged implementation has yet to be incorporated into subsequent mainstream programming languages, such as Java or C#. Nevertheless, other emerging general purpose programming languages have incorporated this approach, such as Chrome and D. More specifically, the Java community (among others) has seen many third-party extensions and libraries to support the approach, through different technological solutions. One such technology is Aspect-Oriented Programming. Moreover, we were presented to some projects which intend to complement the Design by Contract approach run-time evaluation with static evaluation, using more expressive contract languages.

Aspect-Oriented Programming improves the separation of concerns. The AspectJ language extension to Java, in particular, accomplishes this by encapsulating cross-cutting concerns in the aspect module. Alternative Aspect-Oriented Programming languages and middleware systems were considered, but none provided a considerable community as AspectJ.

A prototype was developed for the proposed infrastructure, in the form of a Java/AspectJ JAR library. Some decisions were made, namely in terms of: how to write assertions, how to evaluate, how to implement the evaluation. Additionally to the core features, some secondary features were also discussed.

In order to validate the thesis proposal, the prototype was applied to a set of case studies. Next, the prototype was submitted to engineering evaluations, both in terms of the coverage set of features, as well as non-functional requirements, such as integration and especially performance. The features offered in this prototype provide a solid bootstrap, which can be extended to larger set, although with some engineering trade-offs. Integration with other tools is simple, although in some cases intrusive. Performance is acknowledged as an issue of the prototype. This was properly quantified and discussed, but was not a priority in this work.

Additionally, an adjacent topic to this thesis was discussed, namely on how to introduce the Design by Contract approach into Aspect-Oriented Programming. This provided an interesting discussion on Aspect-Oriented Design, which presents itself as a fertile ground for further research.

In conclusion, Design by Contract in Java can only be truly usable in practice either through a language change, which seems very unlikely; or through annotation support, the privileged extension mechanism for Java, which has now has become the tendency for the Java community. Sun Microsystems seems to be following this line, as demonstrated in Java 6's release, which added support for JVM plug-ins for annotation processing. As stated in the release notes: "*It is becoming a running joke in Java technology circles, at least some that contain us, that for every wished-for feature missing in Java technology, there's a budding annotation that will solve the problem.*" [Sun06]. While we doubt this to be a hard-truth, since syntactic sugar can be very appealing (compare Java with C#), this fits nicely into the Java language philosophy of elegance and simplicity.

On a personal note, I have to say that writing a thesis dissertation is an interesting experience. Among many things, I came to realize that a thesis is never truly finished. However, every thesis has a time window of opportunity, either in technological or research terms. Thus, in order to prevent this thesis from becoming obsolete, I now decide to end it. There are many things that are left to be done, but a milestone is finished. A working prototype was completed, some interesting issues were raised and discussed, and a proof of concept was accomplished. This document now is part of the large body of knowledge of academic research, whether for good or bad [Agostinho08] [Agostinho08a]. Personally, I've come to rethink and appreciate many things in software development, namely the Object-Oriented paradigm. Regarding the Aspect-Oriented paradigm, I think there are some interesting issues which are raised with this way of problem thinking, and whether or not Aspect-Oriented programming languages prevail, some of its mechanisms will certainly incorporate future mainstream programming languages.

References

[Abercrombie02] P. Abercrombie and M. Karaorman *jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java*, Electronic Notes in Theoretical Computer Science, volume 70, Elsevier Science Publishers, 2002.

[Abrial80] J. Abrial, S. Schuman, and B. Meyer *A Specification Language*, in On the Construction of Programs, Cambridge University Press, eds. R. McNaughten and R. McKeag, 1980.

[Agostinho08] S. Agostinho, P. Guerreiro and A. Moreira *Contracts for Aspect-Oriented Design*, 6th Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD'08, Brussels, Belgium, March 31, 2008.

[Agostinho08a] S. Agostinho, P. Guerreiro and H. Taborda *An Aspect for Design by Contract in Java*, 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS) at ICEIS 2008, Barcelona, Spain, July 13, 2008.

[Aksit92] M. Aksit, L. Bergmansand, and S. Vural *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, European Conference on Object-Oriented Programming (ECOOP'92), 1992.

[Aldrich05] J. Aldrich *Open Modules: Modular Reasoning about Advice*, ECOOP 05 Proceedings, LCNS 3586, Springer, 2005.

[Apache06] The Apache Software Foundation *The Jakarta Project - Commons JEXL* , 2006, available at http://jakarta.apache.org/commons/jexl/.

[Apache06a] The Apache Software Foundation *The Apache Jakarta Project - BCEL manual*, 2006, available at http://jakarta.apache.org/bcel/manual.html.

[Apache07] The Apache Software Foundation *The Apache Ant Project*, 2007, available at http://ant.apache.org/.

[Aracic06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann *An Overview of CaesarJ*, Transactions on Aspect-Oriented Software Development, 2006.

[AspectJTeam03] The AspectJ Team *The AspectJ Programming Guide*, 2003, available at http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[AspectJTeam05] The AspectJ Team *The AspectJ Development Environment Guide*, 2005, available at http://www.eclipse.org/aspectj/doc/released/devguide/index.html.

[AspectJTeam05a] The AspectJ Team *The AspectJ 5 Development Kit Developer's Notebook*, 2005, available at http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html.

[AspectResearch07] Aspect Research Associates *Contract4J: Design by Contract for Java - web site*, 2007, available at http://www.contract4j.org/contract4j.

[Balzer06] S. Balzer, P. Eugster, B. Meyer *Can Aspects Implement Contracts?*, Proceedings of Rapid Integration of Software Engineering techniques (RISE) 2006, Geneva, Switzerland, 13-15 September 2006.

[Barnett04] M. Barnett, K. Leino and W. Schulte *The Spec# Programming System: An Overview*, Construction and analysis of safe, secure and interoperable smart devices (CASSIS), 2004.

[Bartetzko01] D. Bartetzko, C. Fischer, M. Möller and H. Wehrheim *Jass - Java with Assertions*, Electronic Notes in Theoretical Computer Science 55 No. 2, 2001.

[Beck03] K. Beck *Test-Driven Development by Example*, Addison-Wesley, 2003.

[Beck94] K. Beck *Simple Smalltalk Testing: With Patterns*, October 1994, available at http://www.xprogramming.com/testfram.htm.

[Beck99] K. Beck *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Benoit04] S. Benoît and B. Jérôme *Lisaac v 0.2 Programmer's Reference Manual*, Loria UMR 7033, 2004.

[Bergström06] J. Bergström *C4J web site*, 2006, available at http://c4j.sourceforge.net/.

[Beugnard99] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins *Making Components Contract Aware*, Computer, Volume 32, Issue 7, July 1999.

[Bjørner78] D. Bjørner and C. Jones *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, Vol. 61, Springer-Verlag, 1978.

[Bonér04] J. Bonér *AspectWerkz 2: An Extensible Aspect Container*, November 2004, available at http://www.theserverside.com/tt/articles/article.tss?l=AspectWerkzP1.

[Bracha04] G. Bracha *Generics in the Java Programming Language*, July 5, 2004, available at http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[Bracha90] G. Bracha and W. Cook *Mixin-based Inheritance*, European Conference on Object-Oriented Programming and Conference on Object Oriented Programming Systems Languages and Applications (ECOOP/OOPSLA'90) Proceedings, October 21-25, 1990.

[Brichau05] J. Brichau and M. Haupt (editors) *Survey of Aspect-oriented Languages and Execution Models*, AOSD-Europe-VUB-01, 17 May 2005.

[Burdy03] L. Burdy, A. Requet and J. Lanet *Java applet correctness: A developer-oriented approach*, In D. Mandrioli, K. Araki and S. Gnesi (editors), FME 2003, volume 2805 of LNC, pages 422-439, Springer-Verlag, 2003.

[Burdy05] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino and E. Poll *An overview of JML tools and applications*, International Journal on Software Tools for Technology Transfer, June 2005.

[Castagna95] G. Castagna *Covariance and contravariance: conflict without a cause*, ACM Transactions on Programming Languages and Systems, Vol 17, No 3, May 1995.

[Chalin05] P. Chalin, J. Kiniry, G. Leavens and E. Poll *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05), 2005.

[Chavez01] C. Chavez, A. Garcia, and C. Lucena *Some Insights on the Use of AspectJ and Hyper/J*, Tutorial and Workshop on Aspect Oriented Programming and Separation of Concerns, Lancaster, UK, August 23-24 2001.

[Colyer04] A. Colyer, A. Clement, G. Harley, and M. Webster *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison-Wesley, 2004.

[Czarnecki00] K. Czarnecki and U. Eisenecker *Generative Programming - Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Dahm98] M. Dahm *Byte Code Engineering with the BCEL API*, Technical Report B-17-98, Institut füt Informatik, Freie Universität Berlin, 1998.

[Dantas06] D. Dantas, and D. Walker *Harmless Advice*, Annual Symposium on Principles of Programming Languages (POPL'06), South Carolina, USA, January 11-13, 2006.

[Darcy06] J. Darcy *JSR 269: Pluggable Annotation Processing API*, 2006, available at http://www.jcp.org/en/jsr/detail?id=269.

[Davidson04] D. Davidson *OGNL Language Guide*, 2004, available at http://www.ognl.org/2.6.9/Documentation/pdf/LanguageGuide.pdf.

[Detlefs03] D. Detlefs, K. Leino, G. Nelson and J. Saxe *Simplify: A theorem prover for program checking*, Technical Report HPL-2003-148, HP Labs, July 2003.

[Deveaux02] D. Deveaux, J. Cam and A. Despland *Software Components Development and Follow-up: the "Design for Trustability" (DfT) Approach*, Proceedings of the Information System Technology Panel Symposium, RTO-MP (ed.), Pages 16-1, Bonn (Germany), September 2002.

[Diaz07] Daniel Diaz *GNU Prolog - Edition 1.8 for GNU Prolog version 1.3.0*, University of Paris, January 4, 2007.

[DigitalMars06] Digital Mars *The D programming language web site*, 2006, available at http://www.digitalmars.com/d/index.html.

[Dijkstra70] E. Dijkstra *Notes On Structured Programming*, EWD 249 Technical U. Eindhoven, 1970.

[Dinkelaker06] T. Dinkelaker, M. Haupt, R. Pawlak, L. Navarro, and V. Gasiunas *Inventory of Aspect-Oriented Execution Models*, AOSD-Europe-TUD-4, 28 February 2006.

[Diotalevi04] F. Diotalevi *AOP@Work: Contract enforcement with AOP*, 15 July 2004, available at http://www-128.ibm.com/developerworks/library/j-ceaop/.

[Duncan98] A. Duncan, and U. Hölzle *Adding Contracts to Java with Handshake*, Technical report TRCS98-32, University of California, December 8, 1998.

[Dzidek05] W. Dzidek, L. Briand, and Y. Labiche *Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java*, Workshop on Tool Support for OCL and Related Formalisms at ModELS 2005, 2005.

[ECMA06] ECMA *Standard ECMA-334 C# Language Specification (4th edition)*, June 2006, available at http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[ECMA06a] ECMA *Eiffel: Analysis, Design and Programming Language (2nd Edition)*, Standard ECMA-367, June 2006.

[EiffelSoftware07] Eiffel Software *Building bug-free O-O software: An introduction to Design by Contract*, 2007, available at http://archive.eiffel.com/doc/manuals/technology/contract/.

[Enseling01] O. Enseling *iContract: Design by Contract in Java*, 16 February 2001, available at http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html.

[Fähndrich06] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi *Language Support for Fast and Reliable Message-based Communication in Singularity OS*, Proceedings of EuroSys2006, Leuven, Belgium, April 2006.

[Filman01] R. Filman, D. Friedman *Aspect-Oriented Programming is Quantification and Obliviousness*, RIACS Technical Report 01.12, May 2001.

[Filman04] R. Filman, T. Elrad, S. Clarke, M. Aksit *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[Findler01] R. Findler, and M. Felleisen *Contract Soundness for Object-Oriented Languages*, Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Florida, USA, 2001.

[Findler01a] R. Findler, M. Latendresse, and M. Felleisen *Behavioral Contracts and Behavioral Subtyping*, Proceedings of ACM Conference Foundations of Software Engineering, 2001.

[FirstPerson94] FirstPerson, Inc *Oak Language Specification*, Confidential, 1994.

[Fleury06] M. Fleury, S. Stark, and R. Norman *JBoss 4.0 - The Official Guide*, Sams Publishing, 2006.

[Fowler99] M. Fowler *Refactoring. Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gamma94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Garcia04] V. Garcia, D. Lucrédio et al. *Uma Ferramenta CASE para o Desenvolvimento de Software Orientado a Aspectos*, XVIII Brazilian Symposium on Software Engineering (SBES Conference), 2004.

[Gleichmann06] M. Gleichmann *SpringContracts Quick Start*, ?, 2006.

[Gleichmann06a] M. Gleichmann *SpringContracts Reference (draft)*, ?, 2006.

[Gosling05] J. Gosling, B. Joy, G. Steele and G. Bracha *The Java Language Specification (third edition)*, Prentice-Hall, 2005.

[Gouda91] M. Gouda and T. Herman *Adaptive Programming*, IEEE Transactions on Software Engineering Volume 17 Issue 9, September 1991.

[Griswold06] W. Griswold, M. Shonle, et al. *Modular Software Design with Crosscutting Interfaces*, IEEE Software (Vol. 23, No. 1), January/February 2006.

[Guerreiro00] P. Guerreiro *Another Mediocre Assertion Mechanism for C++*, TOOLS Europe 2000, Mont Saint Michel, France, Proceedings, pages 226-237, IEEE, 2000.

[Guerreiro01] P. Guerreiro *Simple Support for Design by Contract in C++*, TOOLS USA 2001, Proceedings, pages 24-34, IEEE, 2001.

[Guttag93] J.V. Guttag and J.J. Horning, et al *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, January 19, 1993.

[Han04] Y. Han, G. Kniesel and A. Cremers *A Meta Model for AspectJ*, Technical Report IAI-TR-2004-3, Computer Science Department III, University of Bonn, October 2004.

[Harrison93] W. Harrison and H. Ossher *Subject-Oriented Programming - A Critique of Pure Objects*, ACM SIGPLAN Notices Volume28, Issue 10, October 1993.

[Hoare69] C. Hoare *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Volume 12, Number 10, October 1969.

[Hoare85] C. Hoare *Communicating Sequential Processes*, Prentice Hall, 1985.

[Hoare89] C. Hoare and C. Jones *Essays in Computing Science*, Prentice Hall, 1989.

[Hulten98] G. Hulten, K. Liberherr, J. Marshall, et al. *Demeter/Java User Manual 0.7*, draft, May 11, 1998.

[Inprise99] Inprise Corporation *Delphi 5 Object Pascal Language Reference*, ?, 1999.

[Interface06] Interface21 *Spring Framework web site*, 2006, available at http://www.springframework.org.

[ISO06] ISO *ISO/IEC 23271, Common Language Infrastructure*, , 2006.

[Jézéquel01] J. Jézéquel, D. Deveauxy and Y. Traonz *Reliable Objects: Lightweight Testing for Java*, IEEE-Software, 18(4):76-83, 2001.

[Johnson07] R. Johnson, J. Hoeller, A. Arendsen, et al. *Spring - Java/J2EE Application Framework 2.0.3*, 2007, available at http://static.springframework.org/spring/docs/2.0.x/spring-reference.pdf.

[JRubyTeam07] The JRuby team *JRuby - Java powered Ruby implementation*, 2007, available at http://jruby.codehaus.org/.

[Karaorman03] M. Karaorman and P. Abercrombie *jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation*, Technical Report TRCS98-31, University of California, Department of Computer Science, Santa Barbara, March 26, 2003.

[Karaorman99] M. Karaorman, U. Hölzle and J. Bruno *jContractor: A Reflective Java Library to Support Design By Contract*, Technical Report TRCS98-31, University of California, Department of Computer Science, Santa Barbara, 1999.

[Kernighan88] B. Kernighan, D. Ritchie *The C Programming Language (second edition)*, Prentice Hall, 1988.

[Kersten05] M. Kersten *AOP@Work: AOP tools comparison*, 8 February 2005, available at http://www-128.ibm.com/developerworks/java/library/j-aopwork1/.

[Kiczales01] G. Kiczales, E. Hilsdale, J. Hugunin, et al. *An Overview of AspectJ*, 15th European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary, June 18-22, 2001.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, et al. *Aspect-Oriented Programming*, 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, June 9-13, 1997.

[Kölling98] M. Kölling, and J. Rosenberg *Blue Language Specification (Blue version 1.0; Manual revision 1.1)*, Technical Report TR97-13, Monash University, Department of Computer Science and Software Engineering, 1998.

[Kramer98] R. Kramer *iContract - the Java design by contract tool*, TOOLS 26: Technology of Object-Oriented Languages and Systems, 1998.

[Lackner02] M. Lackner, A. Krall, and F. Puntigam *Supporting Design by Contract in Java*, Journal of Object Technology, Vol.1, No. 3, Special Issue: TOOLS USA 2002 proceedings, 2002.

[Laddad03] R. Laddad *AspectJ in Action: Pratical Aspect-Oriented Programming*, Manning, 2003.

[Laforge04] G. Laforge *JSR 241: The Groovy Programming Language*, 2004, available at http://www.jcp.org/en/jsr/detail?id=241.

[Langer07] A. Langer *Java Generics Frequently Asked Questions*, 2007, available at http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.pdf.

[Leavens00] G. Leavens, K. Rustan, M. Leino, E. Pool, C. Ruby and B. Jacobs *JML: notations and tools supporting detailed design in Java*, Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2000), Minnesota, USA, August 2000.

[Leavens06] G. Leavens and Y. Cheon *Design by Contract with JML*, draft paper, September 28, 2006.

[Leavens06a] G. Leavens, E. Poll, C. Clifton, et al. *JML Reference Manual*, ?, May 2006.

[Leavens99] G. Leavens, A. Baker, and C. Ruby *JML: A Notation for Detailed Design*, adapted from H. Kilov, B. Rumpe, and I. Simmonds (editors), "Behavioral Specifications for Businesses and Systems", Kluwer Academic, 1999.

[Leroy05] X. Leroy *The Objective Caml system release 3.09 - Documentation and user's manual*, , 2005.

[Lesiecki02] N. Lesiecki *developerWorks: Test flexibly with AspectJ and mock objects*, 01 May 2002, available at http://www.ibm.com/developerworks/java/library/j-aspectj2/.

[Lindholm99] T. Lindholm and F. Yellin *The Java Virtual Machine Specification (second edition)*, Prentice-Hall, 1999.

[Liskov93] B. Liskov and J. Wing *Family Values: A Behavioral Notion of Subtyping*, MIT/LCS/TR-562b, Carnegie Mellon University, 16 July 1993.

[Liskov99] B. Liskov and J. Wing *Behavioral Subtyping Using Invariants and Constraints*, Technical Report CMU CS-99-156, Carnegie Mellon University, July 1999.

[Lorenz05] D. Lorenz, and T. Skotiniotis *Extending Design by Contract for Aspect-Oriented Programming*, Technical Report NU-CCIS-04-14, College of Computer and Information Science, Northeastern University, Boston, January 2005.

[Martin91] J. Martin *Rapid Application Development*, Macmillan Coll Div, 1991.

[Mashkoor07] A. Mashkoor and J. Fernandes *Deriving Software Architectures for CRUD Applications: The FPL Tower Interface Case Study*, International Conference on Software Engineering Advances (ICSEA 2007), 2007.

[Meyer91] B. Meyer *Eiffel: The Language*, Prentice-Hall, 1991.

[Meyer92] B. Meyer *Applying Design by Contract*, Computer (IEEE), October 1992.

[Meyer97] B. Meyer *Object-Oriented Software Construction (second edition)*, Prentice-Hall, 1997.

[Mezini03] M. Mezini, and K. Ostermann *Conquering Aspects with Caesar*, AOSD 2003, Boston, USA, March 17-21, 2003.

[Microsoft05] Microsoft corporation *C# Version 2.0 Specification*, September 2005, available at http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx.

[Microsoft06] Microsoft corporation *Microsoft .NET Framework*, 2006, available at http://www.microsoft.com/net/.

[Mitchell02] R. Mitchell and J. McKim *Design by Contract, by Example*, Addison-Wesley, 2002.

[Mitchell97] R. Mitchell, J. Howse and A. Hamie *Contract-oriented specifications*, Proceedings TOOLS24, IEEE, 1997.

[MMS06] Man Machine Systems *Design by Contract for Java Using JMSAssert*, , 2006.

[Norvell02] T. Norvell *The JavaCC Tutorial*, 2002, available at http://www.engr.mun.ca/~theo/JavaCC-Tutorial.

[OMG04] OMG *CORBA specification, v3.0.3*, 2004, available at http://www.omg.org/cgi-bin/doc?formal/04-03-12.

[OMG05] OMG *Unified Modeling Language (UML) 2.0 OCL convenience document*, 2005, available at http://www.omg.org/cgi-bin/doc?ptc/05-06-06.

[OMG05a] OMG *Unified Modeling Language (UML) 2.0 Superstructure Specification*, 2005, available at http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[Ossher99] H. Ossher and P. Tarr *Hyper/J: Multi-Dimensional Separation of Concerns for Java (Demonstration Proposal)*, 1999, available at http://www.research.ibm.com/hyperspace/HyperJ/ECOOP99-Demo-Proposal.html.

[Owre96] S. Owre, S. Rajan, J. Rushby, N. Shankar and M. Srivas *PVS: Combining specification, proof checking, and model checking*, In R. Alur and T. Henziger (editors), "Computer Aided Verification", number 1102 in LNCS, pages 411-414, Springer-Verlag, 1996.

[Parasoft05] Parasoft Corporation *Using Design by Contract to Automate Java Software Testing*, ?, 2005.

[Pawlak03] R. Pawlak *The AOP Alliance: Why Did We Get In?*, White paper draft, July 11 2003.

[PHPDocumentationGroup06] PHP Documentation Group *PHP Manual*, , 2006.

[Plösch02] R. Plösch *Evaluation of Assertion Support for the Java Programming Language*, Journal of Object Technology, Vol. 1, No. 3, Special Issue: TOOLS USA 2002 proceedings, 2002.

[Rainsberger04] J. Rainsberger *JUnit Recipes: Practical Methods for Programmer Testing*, Manning, 2004.

[RedHat07] Red Hat *JBoss AOP Reference Documentation 1.5*, 2007, available at http://labs.jboss.com/portal/jbossaop/docs/index.html.

[RedHat07a] Red Hat *JBoss AOP User Guide 1.5 - The Case For Aspects*, 2007, available at http://labs.jboss.com/portal/jbossaop/docs/index.html.

[RemObjects06] RemObjects Software *Chrome web site*, 2006, available at http://www.chromesville.com/.

[Rieken07] J. Rieken *Design By Contract for Java - Revised (master thesis)*, Carl von Ossietzky Universität - Correct System Design Group, April 24th, 2007.

[Rossum06] G. van Rossum *Python Reference Manual*, Release 2.5, 19th September, 2006.

[Shinotsuka06] S. Shinotsuka, N. Ubayashi, N., Shinomi, and T. Tamai *An Extensible Contract Verifier for AspectJ*, Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development (AOAsia 2), 2006.

[Skotiniotis04] T. Skotiniotis, and D. Lorenz *Conaj: Generating Contracts as Aspects*, Technical Report NU-CCIS-04-05, College of Computer and Information Science, Northeastern University, Boston, March 2004.

[Sommerer98] A. Sommerer *The Java Archive (JAR) File Format*, September 1998, available at http://java.sun.com/developer/Books/javaprogramming/JAR/.

[Stroustrup00] B. Stroustrup *The C++ Programming Language (Special Edition)*, Addison-Wesley, 2000.

[Sun04] Sun Microsystems *Java 2 Platform Standard Edition 5.0 API Specification*, 2004, available at http://java.sun.com/j2se/1.5.0/docs/api/.

[Sun06] Sun Microsystems *What's New in Java SE 6*, 2006, available at http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/beta2.html.

[Suvée05] D. Suvée, B. De Fraine, and W. Vanderperren *FuseJ: An architectural description language for unifying aspects and components*, L. Bergmans, K. Gybels, P. Tarr, et al (editors), Software Engineering Properties of Languages and Aspect Technologies, March 2005.

[Suvée06] D. Suvée, B. De Fraine, and W. Vanderperren *A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development*, Component-Based Software Engineering (CBSE) 2006, Västerås, Sweden, 2006.

[Szathmary02] V. Szathmary *Barter - beyond design by contract*, 2002, available at http://barter.sourceforge.net.

[Tarr00] P. Tarr and H. Ossher *Hyper/J User and Installation Manual*, 2000, available at http://www.dcs.bbk.ac.uk/~yhassoun/teaching/hyperj-user-manual.pdf.

[Tarr99] P. Tarr, H. Ossher, W. Harrison, et al. *N degrees of separation: multi-dimensional separation of concerns*, Proceedings of the 21st international conference on Software engineering, Los Angeles, United States, 1999.

[Thomschke06] S. Thomschke *OVal - object validation framework for Java 5 or later*, 2006, available at http://oval.sourceforge.net/.

[UniversityOfCalifornia96] International Computer Science Institute at Berkeley University of California *Sather home page*, 1996, available at http://www.icsi.berkeley.edu/~sather/.

[UniversityOfWroclaw06] Computer Science Institute of the University of Wroclaw *Nemerle web site*, 2006, available at http://nemerle.org.

[Walls03] C. Walls and N. Richards *XDoclet in Action*, Manning, 2003.

[Walls05] C. Walls and R. Breidenbach *Spring in Action*, Manning, 2005.

[Wampler06] D. Wampler *Contract4J for Design by Contract in Java: Designing Pattern-Like Protocols and Aspect Interfaces*, Industry Track at AOSD 2006, Bonn Germany, March 22, 2006.

[Wampler06a] D. Wampler *The Challenges of Writing Reusable and Portable Aspects in AspectJ: Lessons from Contract4J*, Industry Track, AOSD 2006, Bonn Germany, March 22, 2006.

[Wampler06b] D. Wampler *AOP@Work: Component design with Contract4J*, 11 April 2006, available at http://www-128.ibm.com/developerworks/java/library/j-aopwork17.html.

[Wampler07] D. Wampler *Aspect-Oriented Design Principles: Lessons from Object-Oriented Design*, Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, British Columbia, March 12-16, 2007.

[Zhao03] J. Zhao, and M. Rinard *Pipa: A Behavioral Interface Specification Language for AspectJ*, Proceedings of Fundamental Approaches to Software Engineering (FASE'2003), 2003.

Alphabetical Index

A
annotation....9, 15, 17, 18, 24-26, 28-30, 32, 35, 37, 38, 40, 42, 69, 70, 84, 89, 99, 102
aspect-oriented....
  design....2, 7, 10, 91, 99, 107
  programming....1, 2, 7, 9, 10, 26-30, 34, 35, 40, 42, 45-47, 50, 60, 70, 82, 94, 95, 97, 99, 100, 102-105
  software development....1, 7, 35, 91, 92, 98, 101, 103, 106, 107
Assembly....10, 57
C
C programming language....11
C# programming language....1, 10, 32-34, 46, 60, 67, 89, 99, 103, 105
C++ programming language....10, 13, 32, 55, 103, 106
class loader....21, 39, 54, 70, 80
Cloneable interface....19, 22, 55, 60, 88
component....
  -based software development....42, 95, 106
composition filters....35
concern....
  cross-cutting....1, 7, 35, 42, 95, 99
  separation of....1, 35, 43, 46, 65, 99, 102, 105, 106
concurrency....20, 33, 60, 69, 73, 75, 78, 80
constructor....7, 10-12, 16, 23, 28, 32, 35, 36, 38, 41, 46, 52, 53, 56, 57, 59, 60, 73, 95, 98
contract extension....7
contravariance....3, 4, 59, 78, 95, 102
correctness....6, 84, 102
coupling....51, 94
covariance....3, 4, 59, 78, 96
cross-cutting....
  dynamic....9, 11, 36, 45, 93
  static....9, 36, 38, 41, 45, 54, 88, 93
D
debug....5, 11, 13, 23, 49, 70, 89
DemeterJ programming language....27, 97
deployment....
  distributed control flow....46
  hot....42
  local....17, 45
  remote....46
  static....17, 45
  thread-based....17, 45
destructor....10
dominant decomposition....43
E
Eclipse....
  Foundation....35, 39, 44
  IDE....39, 42, 49, 90
Error class....20, 52
Exception class....13, 30, 36
existential quantifier....19, 24, 30, 33, 72, 88
expose....12, 17, 32, 58-61, 66, 67, 72, 75, 84, 87, 96
F
first-order logic....15, 19, 22-26, 30, 33, 69, 72, 80, 82, 84, 88
frame rule....17, 20, 25, 32, 69, 73, 75, 78, 80, 87, 89, 96
G
garbage collection....10
general purpose programming language....99
H
hashtable data structure....55, 59

heuristic....88
hypermodule....43
hyperslice....43, 44
I
inheritance....
  contract....3, 4, 7, 9, 11, 12, 14-17, 19, 21, 23, 27-30, 49, 55, 60, 69, 71, 78, 87, 96, 97
  multiple....11, 15, 32, 36, 94, 95, 98
intertype declaration....46, 84, 87, 93, 95, 97
J
Java....
  Archive....2, 15, 39, 58, 87, 99, 106
  Enterprise Edition....30, 40, 42, 46, 50
  language....1, 14, 16, 20, 23, 49, 54, 59, 65, 86, 90, 99
  Modeling Language....6, 9, 23-25, 31, 67, 80, 89, 96, 97, 102, 104, 105
  Runtime Environment....39
  Standard Edition....54
  Virtual Machine....24, 39, 69, 105
javadoc....18, 20, 21, 24, 26, 39, 70, 90
L
Liskov Substitution Principle....7, 10, 11, 14, 17, 19, 21, 23, 25-27, 30, 32, 55, 71, 78, 94-97
loop....
  invariant....7, 73
  variant....7, 8, 20, 73, 87
lower camel case....52
M
metadata....26, 28, 33, 38, 42, 49, 50, 70, 80, 82
method....
  abstract....17, 44, 90
  getter....17, 24
  pure....17, 24, 33
  setter....63-65
mixin....11, 17, 41, 45, 80, 88, 94-96, 98, 102
model variables....24
N
non-functional requirement....49, 99
O
Object class....6-9, 28, 32, 33, 45, 46, 51, 54, 56, 71, 91, 100-107
Object Constraint Language....5, 6, 19, 51, 70, 88, 103, 105
object-oriented....
  design....67, 91, 107
  programming....9, 45, 46, 71, 101, 102, 104
  software development....3, 7, 91
overloading....17, 38, 52, 53, 57, 64, 65
overridding....11, 32, 44
P
partial type....46
polymorphism....44, 52, 59, 60, 71, 75, 82, 86, 88, 94
privileged....38, 99
programming....
  adaptive....35, 103
  contract....10
  defensive....5, 6, 9, 17, 63, 67
  extreme....101
  generative....50, 60, 102
  structured....102
  subject-oriented....35, 104
Prolog programming language....97, 102
property file....57
proxy-based AOP....42
R
Rapid Application Development....9, 105
recursion....10, 15, 18, 30, 53, 55, 69, 73, 75, 78, 80, 82, 86
refactoring....67, 90, 103

reflection....28, 53, 56, 60, 67, 70, 86
reification....50
reliability....1, 3, 5, 7, 65
RuntimeException class....19, 51, 53
S
side-effect....10, 15, 16, 24, 33, 56, 59, 61, 69, 73, 78, 82, 84, 88
singleton pattern....45, 46, 95
stack data structure....13, 55, 60, 67, 73
synchronized....5, 20, 57, 89
syntactic sugar....24, 99
T
test-driven development....6, 55, 67, 101
thread....17, 19, 20, 45
Throwable class....36, 51, 52, 80
trace....13, 17, 20, 49, 50, 60, 67, 70, 73
type....
  generic....102, 104
  primitive....28, 35, 52, 55, 64, 66, 71, 80, 84, 97
U
Unified Modeling Language....2, 6, 49, 105
unit testing....6, 9, 10, 21, 24, 26, 31, 66, 67, 69, 74, 78
universal quantifier....19, 24, 30, 33, 70, 72
V
verification....
  runtime....31, 34
  static....3, 9, 31, 34
W
weaving....
  binary....39
  compile-time....38
  load-time....39
.
.NET platform....9, 12, 13, 33, 105

Annex A

FPL Tower Interface Acronyms and Definitions

In this annex, the technical acronyms corresponding to the FPL Tower Interface case study can be found, along with the respective data type definitions. Note that a time data type is a four digit value in the “HHMM” format, in which 00 ≤ HH ≤ 23 and 00 ≤ MM ≤ 59.

parameter	description	value type
ADEP	Aerodrome of Departure	Text string
ADES	Aerodrome of Destination	Text string
Aircraft	Aircraft identification	1 character + 3 digit number
ATA	Actual Time of Arrival	Time
ATD	Actual Time of Departure	Time
AOBT	Actual Off-Block Time	Time
ATON/ATD	Actual Time Over entry point / Actual Time of Departure	Time
ATOX/ATA	Actual Time Over exit point / Actual Time of Arrival	Time
C/S	Call sign	Text string (2 characters) + integer number (2 or 3 digits)
CTOT	Calculated Take-Off Time	Time
Divert	-	Boolean
(Divert) Aerodrome	-	Text string
(Divert) PrevADES	Previous Aerodrome of Destination	Text string
(Divert) Remarks	-	{ “NoDiv”, “Fuel”, “Wx Wind”, “Wx Visibility”, “Technical”, “Medical”, “Other” }
(Divert) Time	-	Time
End Training	-	Boolean
EOBT	Estimated Off-Block Time	Time
ETA	Estimated Time of Arrival	Time
ETON/ETD	Estimated Time Over entry point / Estimated Time of Departure	Time
ETOX/ETA	Estimated Time Over exit point / Estimated Time of Arrival	Time
First Contact	-	Time
FPLCat	FPL Category	{ “Inbounds”, “Outbounds”, “Internal”, “Overflights” }
FPLKey	FPL Database key	Integer
FPLStatus	FPL Status	{ “Inactive”, “Active”, “Terminated” }
Go Around	-	Boolean
Last Contact	-	Time
Low Pass	-	Boolean

parameter	description	value type
LVO	Low Visibility Operations	Boolean
QNH	Q-code designation for atmospheric pressure at mean sea level	Decimal number in range [925.0, 1065.5]
RegMark	Registration Mark	Alphanumeric text string (up to 14 characters)
RWY	Runway	Integer number (2 digits)
SID	Standard Instrument Departure	Text string (2 characters) + integer number (2 or 3 digits)
SlotMSG	Slot Message	Text string
SSR	Secondary Surveillance Radar	Octal number (4 digits)
STD	Stand	Text character + integer number (2 digits)
Strip	Print Strip command	Boolean
SU	Start Up	Boolean
Touch & Go	-	Boolean
Touch & Go Time(s)	-	Time
Training	-	Boolean
WTC	Wake Turbulence Category	{ “High”, “Medium”, “Low” }

Annex B
Sun Microsystems' statement on Design by Contract support for Java

The following is transcribed from *Programming With Assertions*, Sun Microsystems, 2002, http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html:

> Question
>
> Why not provide a full-fledged design-by-contract facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?
>
> Answer
>
> We considered providing such a facility, but were unable to convince ourselves that it is possible to graft it onto the Java programming language without massive changes to the Java platform libraries, and massive inconsistencies between old and new libraries. Further, we were not convinced that such a facility would preserve the simplicity that is the hallmark of the Java programming language. On balance, we came to the conclusion that a simple boolean assertion facility was a fairly straight-forward solution and far less risky. It's worth noting that adding a boolean assertion facility to the language doesn't preclude adding a full-fledged design-by-contract facility at some time in the future.
>
> The simple assertion facility does enable a limited form of design-by-contract style programming. The assert statement is appropriate for nonpublic precondition, postcondition and class invariant checking. Public precondition checking should still be performed by checks inside methods that result in particular, documented exceptions, such as `IllegalArgumentException` and `IllegalStateException`.

Prologue

Final remarks

It is now over three months since the first print of this dissertation. Since then, several events have happened. Two research papers based on this dissertation have been published and presented, which has given me the opportunity to travel and to get in touch with the actual research community. I was invited to write an extended version for both of these papers. My master thesis was discussed and had an interesting feedback. I am now officially a full time software developer in the so-called "industry", which is the place I want to be for the time being. Thus, a cycle is completed. Having said that, the door for future contact with the research community is not closed.

Regarding the actual thesis dissertation, there are some points that I would like to write down for the interested reader (I assume that you are one since your reading this non-indexed section). In technological terms, time has not been kind. Java 1.6 has been out for some time, and AspectJ 1.6 followed. A new programming language with DbC support has appeared: **Fortress**. In terms of state of the art for DbC, there are a couple of tools that slipped through me and remained unexplored:

- **Nice** (http://nice.sourceforge.net/), a programming language with native DbC support.
- **CodePro AnalytiX** (http://old.instantiations.com/codepro/analytix/default.htm), a "traditional" *ad-hoc* solution.
- **Custos** (https://custos.dev.java.net/), an AOP *ad-hoc* solution.

In terms of research papers, there are also a couple of works that evaded me:

- K. Yamada, and T. Watanabe *Moxa: An Aspect-Oriented Approach to Modular Behavioral Specifications*, Workshop on Software-Engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD 2005, Chicago, Illinois, March 2005.
- G. Leavens *JML's Rich, Inherited Specifications for Behavioral Subtypes*. Technical Report TR #06--22, Computer Science, Iowa State University, 2006.

Some mention to the whole "Contract-Oriented Software Development for Internet Services" (COSODIS) project (http://www.ifi.uio.no/cosodis/) could also have been done.

Thus, the story ends.

Sérgio Agostinho, Barreiro, July 5, 2008