Universidade de Aveiro Departamento de
Electrónica, Telecomunicações e Informática
2012

Pedro Filipe do Amaral Goucha Francisco

Contract-Java: Programação por Contrato em Java

Contract-Java: Design by Contract in Java

Universidade de Aveiro Departamento de
2012 Electrónica, Telecomunicações e Informática

Pedro Filipe do Amaral Goucha Francisco

Contract-Java: Programação por Contrato em Java

Contract-Java: Design by Contract in Java

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Miguel Augusto Mendes Oliveira e Silva, Professor do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos / acknowledgements

Agradeço à minha família, aos meus amigos e aos conhecidos que, de forma continuada, ocasional ou única, me apoiaram e incentivaram ao longo do meu percurso académico.

Não posso também deixar de agradecer ao meu orientador, Professor Miguel Augusto Mendes Oliveira e Silva, por todo o apoio dado ao longo da realização deste trabalho.

Palavras-chave Programação por Contrato, DbC, Java, Contract-Java

Resumo

A programação por contrato é uma metodologia de programação que implementa mecanismos de correcção de forma bem adaptada à programação orientada por objectos, facilitando a construção de software correto e robusto, permitindo também a sua documentação e especificação e a construção de programas tolerantes a falhas. No entanto, ao contrário da programação orientada por objectos, a programação por contrato tem uma difusão bastante reduzida. Uma das razões para tal facto é a quase completa ausência de suporte para a metodologia na grande maioria das linguagens de programação usadas actualmente, nas quais se inclui a linguagem Java. Apesar de existirem algumas ferramentas para tentar suprir essa omissão da linguagem Java, são aproximações incompletas que não permitem usufruir de todas as vantagens e capacidades da programação por contrato.

Neste trabalho pretende-se definir quais as características necessárias numa linguagem de modo a permitir a implementação completa da metodologia, avaliando as falhas que as ferramentas existentes possuem e, de seguida, definir e construir uma nova linguagem, "Contract-Java", definida como uma extensão da linguagem Java, que permita usar a programação por contrato na sua totalidade.

Keywords Design by Contract, DbC, Java, Contract-Java

Abstract

Design by Contract is a programming methodology which implements correction mechanisms well adapted to object-oriented programming, easing the construction of correct and robust software, as well as allowing its documentation and specification and the construction of fault-tolerance programs. However, unlike object-oriented programming, Design by Contract has a very low distribution. One of the reasons for such is the lack of support for it on most programming languages currently in use, in which Java is included. Although a few tools attempt to workaround such lack of support, they all present incomplete approaches which do not support all the advantages and capabilities of Design by Contract.

In this work, we intend to define which characteristics are necessary in order to fully implement the methodology, evaluating the faults of existing tools and, afterwards, defining and constructing a new language, "Contract-Java", defined as an extension of the Java language, which allows to use Design by Contract in its entirety.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In this work we intend to implement a new Java extension for Design-by-Contract programming. Design-by-Contract programming, or officially *Design by Contract*®[1] (DbC) is a development methodology inspired both by studies on formal programming and also, in many ways, by the way contracts work in the "real world". It was born in 1986[Meyer 86] [Mandrioli 92, p. 1-50] [Meyer 92a] and first implemented within the Eiffel language (1988) [Meyer 88a]. Several approaches exist to extend Java to support this methodology but all have achieved just a portion of the features required by the methodology.

In this work we intend to identify and discuss the requirements posed to provide a complete support for DbC and design and implement a compiler for an DbC extension to Java.

1.1 Motivation

We will detail the two main motivations for this work. We believe Design-by-Contract is a enhancement to the current way software is developed and we do not recognize most of the Design-by-Contract requirements and advantages to be fulfilled by existing tools in Java, nor believe it can be achieved using approaches taken by existing approaches.

1.1.1 Design-by-Contract as an improvement to correction, documentation and debug

Software development methods have been refined over the years, with some success in reducing the cost of software development. Some of those methods are behavioral, like better planning. Others are at the software level: structured programming and object-oriented programming have eased the programmer's ability to grasp his/her or others code; such developments and, more recently, the increasing usage of unit tests, brought benefits in terms of required maintenance and lower bug count. We can consider that software is slowly but increasingly being more logically organized, allowing for the easy development of unit tests, which strengthens the advantages of code modularity.

Even so, software is not expected to work well on release. It is not expected to do its job well until one or two releases have gone by. Companies regularly apply the "ship now, fix later" mentality. Sometimes this is a strategy designed to reduce time to market; for example, most Google® software, in the form of web applications, is released with the “beta” label,

[1]trademarked by Eiffel Software in the United States

reaching maturity on later stages of development. However, the public perception is that most times the first version of a software is indeed in “beta” state, with the purchasers of the first versions of a software finding a bigger number of bugs than it would be acceptable for a final product in any other kind of market. Though beyond the scope of this discussion, the fact is that consumer software is usually accepted "as is". Furthermore, consumer market demands themselves may not focus on quality but on more features or better interface. Even when things go wrong, the lack warranty for consumer market means the software vendor has reduced incentives to provide a *bug-free* product from the start. On the professional market however there is the economic incentive to demand better software.

With the increased competition, software ubiquity and lower time to market, the only way software companies can survive is to embrace all possibilities of enhanced software production available. In either case, if we consider that the cost of a programming error increases proportionally to the time it takes until the problem is detected and fixed, we can consider that strategies that further reduce code maintenance demands potentially increase the profits associated with it.

It is therefore of interest to everyone involved in the ecosystem of software development to ask the following question: “How to achieve that demanded higher quality?”. The Design-by-Contract programming methodology may be one of the possible answers.

Design-by-Contract programming has similarities with the logic of unit-testing, in which we test a method input and output. However, unlike unit-testing, in which tests are made separately from the code (both in terms of temporal as well as 'spatial' location), the Design-by-Contract programming methodology focuses on what is already common on current programming, input validation and associated error handling, separating it from the main code to a preceding, logically distinct, code block.

Mirroring real-world contracts, when constructing a method the programmer establishes a contract, in which its clauses are called preconditions, which must be fulfilled by the calling code. By not having to deal with malformed input and by separating clearly input validation logic from method work logic, the method's code is easier to understand. And since the input is guaranteed to be valid once we enter the method, the code flow is unique.

Design-by-Contract programming also allows for an invariant to be established: a more or less rigid set of rules which defines the class conceptually, that can be defined as a class-wide verification of the internal logic, in what can be seen as a contract of the class with itself – or an executable validation of the class as an ADT[2] . An invariant violation means the class does not have the properties expected from it, as a result of the programmer's error, and is therefore unreliable.

Besides the preconditions and the invariant, a method should also check the return it provides, validating its output using contracts (called 'postconditions') established by the supplier detailing not only what the client can expect from its work but also what the class expects to have changed having ended that method. Again, a failure of these validations mean the contracted method has made an error and cannot be trusted.

These assertions, invariants, preconditions and postconditions, provide a valuable tool in the early detection of programming errors by allowing the programmer to write a specification for the program which allows for the verification of its correctness (or more precisely, the lack of errors according to such specification). However, they also provide automatic documentation, a fault-tolerance mechanism, a clear responsibility distribution and, finally, promoting a

[2]Abstract Data Type; we present a definition in section 2.2.1

disciplined and systematic programming approach.

1.1.2 Design-by-Contract implementations for Java

Several Design-by-Contract programming frameworks exist for Java but all of them treat contracts as an after-thought, being usually implemented using annotations. However, annotations fail at convening a close coupling between a class, its methods and contracts enforcement. The proliferation of annotations lead to a certain "information overload" on the part of the programmer which can undo many beneficial effects enabled by Design-by-Contract.

On the other hand, an Eiffel-like approach, such as the one we mirrored on the Contract-Java implementation, presents a cohesive method flow, natural to the Design-by-Contract concept and avoiding a separate handling of contracts and normal code, both relative to its location and time of coding. It thus promotes a disciplined programming approach which yields better specification and correctness, documentation and eases code reuse.

1.2 Objectives

We will define a new language, denominated “Contract-Java”, which allows for the following of a Design-by-Contract methodology on top of Java. Such objective depends on the support of various requirements, namely the support of various kinds of assertion instructions (contracts), differentiated according to their location on the code and thus their semantic meaning, as well as an unified code flow. It has to feature support for contracts as normal class entities, allowing for a mechanism of disciplined fault-tolerance to be implemented naturally. We will define a prototype to allow for the testing of our language, which should support all of the above requirements and should implement inheritance.

1.3 Methodology

After surveying the existing options available, as well as determining their current support status and how they worked, we decided to pick a system to generate a parser generator, having looked at both ANTLR and Bison/Yacc. We tested the usage of the compiler translating Java to Java; after doing so successfully, we’ve started implementing skeleton code as to provide a simple exception mechanism which allowed us to implement a exception mechanism with a minimum amount of overhead. Finally, using a minimal, non-automated, test suite, we enhanced the code generation using real-world examples and proceeded to write the present document.

1.4 Document organization

This document is organized as follows. In chapter 2, we will present a introduction to Design by Contract, followed by an analysis of the state of the art on chapter 3. Afterwards, a coverage of the steps taken, along with decisions made and problems solved, will be presented. Finally, we conclude with an overall analysis of our work and whether we have reached the goals we defined for the work. In Appendix A we present a first approach of a CJ Library contractualizing the Java API.

Chapter 2

Design by Contract

"I believe that the use of Eiffel-like module contracts is the most important non-practice in software world today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. ... This sort of contract mechanism is the sine-qua-non of sensible software reuse.", Tom DeMarco, IEEE Computer, 1997 [North 97]

2.1 The meaning of DbC

From all software quality factors, two are of particular importance within the process of program development: correctness and robustness. The first measures the ability of software to meet its specification and the second is its capacity to deal with unexpected conditions (for example, an hardware malfunction or program errors). Design by Contract [Meyer 88b] (DbC) is a methodology aiming for a substantial improvement of those two factors.

The essential idea behind DbC is to explicitly attach meaning to software elements, specially to the most important ones. When a programmer develops a new function there is no doubt that in his mind there is always a desired meaning attached to it (the function semantics). Consider, for example, a function `minimum` implemented in Java:

```java
static int minimum(int v1, int v2)
{
   int result = (v1 < v2 ? v1 : v2);

   return result;
}
```

The meaning of this function is the fact that its result should always be equal to the lowest value of its two arguments. That is exactly what a client of the function expects to happen when he chooses to use it. However, this precise meaning can only be found in the function's name and on its implementation. If, by any chance, the function is incorrect, such error could easily bypass the function execution and only show its presence far away from its origin (somewhere else in the clients code). From a formal point of view, there is also nothing in the function that states explicitly its meaning so one could (theoretically) argue that the function is not incorrect because it is not formally specified. We could (and should), of course, attach a comment to the function making a more explicit statement of its desired meaning. However, comments may also be the source of undesirable programming errors because nothing guarantees that the comments correctly specify the attached software element.

Table 2.1: Business contract comparison (extracted from [Meyer 92a])

provide telephone service	OBLIGATIONS	BENEFITS
Client	Provide letter or package of no more than 5kgs. Pay 100 francs.	Get package delivered to recipient in less than 4 hours
Supplier	Deliver package to recipient in less than four hours.	No need to deal with deliveries too heavy or unpaid

Recognizing that all software elements have meaning, a more powerful technique is necessary to make it explicit. That is what DbC is all about.

```
static int minimum(int v1, int v2)
{
   int result = (v1 < v2 ? v1 : v2);

   assert result == v1 && result <= v2 || result == v2 && result <= v1;

   return result;
}
```

In this modification of function `minimum`, a native Java's approach is taken to make such a meaning both explicit and formal. Its meaning is expressed as a boolean expression (an assertion) – required to be always true in a correct program – and checked by an `assert` instruction which launches an exception when the assertion is false (otherwise it does nothing).

Unlike previous discussed approaches, this one not only makes the function's meaning explicit, but also makes it testable regardless of the existence of incorrect comments or the clients code. This is a first example of a DbC contract (involving a so called postcondition).

Design by Contract® can be traced back to Meyer's book "Object-oriented software construction" [Meyer 88b] and to its developed programming language Eiffel [Meyer 92b]. The first published references appear in 1986 [Meyer 86] (other relevant publications can be found in [Mandrioli 92, pg. 1-50] and [Meyer 92a]).

The concepts in which Design by Contract is based are present in the works of Turing [Turing 89], Floyd [Floyd 67], Hoare [Hoare 69], Dijkstra [Dijkstra 76], Gries [Gries 87], Jones [Jones 80, Jones 86] and also Goguen [Goguen 78].

DbC takes the approach of viewing software elements as contracting parties with obligations and benefits, akin to a business contract. In a normal business contract, there is a client and a supplier, and both agree on certain conditions to be fulfilled before and after an exchange between the two, as exemplified in table 2.1.

Contracts not only gives the possibility to explicitly state what is desired – hence making it unambiguously clear what is a program error – but also makes it absolutely clear the responsibilities of both parts: the supplier and the client.

When adapting contracts for software elements – as DbC does – the same goal is aimed. Software contracts not only specify the meaning/correctness of the entities to which they are attached to, but also clearly distribute the responsibility of errors when a contract fails. From a practical engineering point of view, this achievement makes a significant contribution to ease software development.

2.1.1 Hoare's triplets

As the given example makes it clear, an assertion poses an obligation upstream, to the preceding code (in the example: the function implementation), and a guarantee downstream (to all clients of the function). Hence, depending on the position of the assertion regarding the software element it can be seen as a *precondition* or as a *postcondition*.

In its origin, the formalism of preconditions and postconditions in programming was proposed by Hoare as his famous triplets [Hoare 69]. According to Hoare, correctness is a relative notion directly dependent upon a specification [Hoare 69]. Hoare triplets fill that gap providing a formal specification for a program block:

$$\{P\}\ Q\ \{R\}$$

"If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion." [Hoare 69]

Being Q a set of instructions, P and R are, respectively, the preconditions and postconditions. If the method, given arguments which respect P, after computation, respects R, we can reasonably assume the computation is correct [Bolstad 04].

Given its importance for software developing, methods are the more important application of Hoare's triplets.

```
static String minimum(String w1, String w2)
{
    assert w1 != null && w1.length() > 0: "Missing word in argument w1";
    assert w2 != null && w2.length() > 0: "Missing word in argument w2";

    String result = (w1.compareTo(w2) < 0 ? w1 : w2);

    assert result == w1 && result.compareTo(w2) < 0 ||
           result == w2 && result.compareTo(w1) < 0;

    return result;
}
```

In this new `minimum` function, unlike in the previous one, it makes sense the define a precondition stating the existence of nonempty object strings. Both the preconditions and the postconditions completely specify the minimum function.

From the examples given, the first two DbC requirements for an appropriate language support can be stated:

R1-*different assertions*: Different assertions for different types of contracts (preconditions, postconditions, invariants and others): the existence of a `assert` instruction, along with different kinds of assertions for the different types of contracts we are establishing, namely preconditions, postconditions and invariants. These assertions, which convey the enforcement of the defined contracts, assume different roles depending on their kind, carefully assigning responsibility to different parts of the program.

R2-*locality*: Contracts should be defined near to the entities they specify. The meaning (specification) of a software entity the contracts should be defined near the classes they contractualize; they are integral part of the code.

2.2 DbC and Object-Oriented Languages

Taking a closer look to the last example code, a new problem can be identified: if those assertions specify the function, how come they are part of the function implementation and not its interface? Clearly, function contracts should be part of its declaration, not its implementation.

This problem, directly linked to the meaning and purpose of functions (methods in Java) within a program, it further enhanced when we consider the other important object-oriented structuring mechanism: classes, and its related abstraction mechanisms: subtype polymorphism.

2.2.1 Abstract Data Types

An *Abstract Data Type* (ADT) [Liskov 74, Meyer 88b] is defined as a type that is completely defined by the external operations it provides and its semantics. A *class* is the implementation, possibly partial, of an ADT [Meyer 97, pg. 142].

And finally, an object-oriented program is a structured collection of ADT implementations [Meyer 97, pg. 147].

Hence, ADTs are the most important abstraction blocks within object-oriented programming.

However, ADTs without explicit semantics (as provided by non DbC languages such as Java) suffer from the same serious problems as methods without contracts, increased by a scale factor because an ADT exports multiple methods (and not just one), and contains a (possible abstract) data representation.

Since a class is much more than the sum of its public methods, a new contract is required to express such semantics. That is the role of *invariants*, which express assertions that are always true when the class's instances (objects) are in an observable state (named: stable times [Meyer 97, pg. 364]).

The set of contracts (preconditions and postconditions) of all the class's public methods together with the class invariant form the class contract. This contract is the most important contract in object-oriented programming.

If we take a broader view of many of concepts presented so far – ADTs, contracts, methods and classes – we can recognize that they all fit perfectly together. ADTs define the class interface. The class contract implements the ADT's semantics. The class is defined as a set of public methods glued by a common invariant. Methods contracts implements the method semantics.

R3-*interface*: Contracts are part of the interface (not implementation): contracts are expected to be readily available to anyone, with or without access to the source code of a contracted program: they are part of a program's interface with the rest of the program.

Figure 2.1 shows a native Java attempt to implement an `Date` ADT, regardless of its possible internal representation.

```
public class Date
{
  // invariant: valid(day(),month(),year());

  public static boolean validMonth(int month)
  {
    boolean result;
    ...
    assert result && (month >= 1 && month <= 12) ||
           !result && (month < 1 || month > 12);
  }

  public static int monthDays(int month, int year)
  {
    assert validMonth(month);
    int result;
    ...
    assert result >= 28 && result <= 31; // inc.
  }

  public static boolean valid(int day, int month, int year)
  {
    boolean result;
    ...
    assert result && validMonth(month) && (day >= 1 && day <= monthDays(month,year) ||
           !result && (!validMonth(month) || day < 1 || day > monthDays(month,year));
  }

  public Date(int day, int month, int year)
  {
    assert valid(day,month,year);
    ...
  }

  public int day() {...}
  public int month() {...}
  public int year() {...}

}
```

Figure 2.1: Class contracts for `Date` ADT.

When methods are part of an ADT, their meaning is not simply the one presented previously (see 2.1.1). What is missing is the ADT's invariant.

To include the invariant of a class as part of the method's contract using Hoare's triplets, we need to check the invariant before checking for the method precondition and after checking for the method postcondition [Meyer 88b, pg. 127].

$$\{\text{invariant} \ \wedge \ \text{precondition}\} \text{ method-body } \{\text{postcondition} \ \wedge \ \text{invariant}\}$$

2.2.2 Liskov's substitution principle

Liskov's substitution principle states that, on object-oriented programming, any property which is verified on a supertype also holds for its subtype objects.

> Let ϕ(x) be a property provable about objects x of type T . Then ϕ(y) should be true for objects y of type S where S is a subtype of T [Liskov 94].

In the context of DbC, this implies that class contracts must be inherited. As Meyer states [Meyer 88b] it is possible to redefine contracts on child classes as long as certain conditions are met. The precondition of the child class must be equal or weaker than that of the parent class and, in the case of invariants and postconditions, the child class must abide at least by the parent class, meaning it can further restrict its invariant and/or its output (postconditions), but never to weaken them.

Since contracts must be taken in consideration by the child classes in order to not change the ADT associated with the parent class, we further strengthen the need for requirement `R3`: the contracts must be part of the class interface and not implementation – otherwise, the semantic meaning of the class would be partially hidden from the outside view, stripping the added value which contracts bring on defining ADT. To underline such dependence upon the parent class contracts, we thus define our fourth requirement for successfully implementing DbC:

R4-*inheritance*: Contracts are inherited: a derived class must fulfill at least all contracts of its parent class, as well as its method's postconditions; preconditions can, but don't have to, be loosened.

The typical implementation of the above is done applying 'or' clauses to the preconditions and 'and' clauses to the invariant and postconditions. This approach is not without its flaws, namely on the validation of contract inheritance redefinition correctness [Findler 01b], but such issues are beyond the scope of this work.

2.2.3 Command-query separation

Command-query separation is the concept of, as the name implies, having well defined effects for a method: it can either be a command and thus possible to affect the object it is applied to, a query, in which case the object is not affected.

Command-query separation refers to the clear separation of a command and a query. A command is a void method which is expected (but not necessarily always) to change the state of the program. A query is a method which returns a value, being an observer of the object state, and can, but does not have to, change the state of the program. A query which never changes the state of the program is called a "pure query".

It can be seen as a compromise between functional programming language properties, which allow for the application of mathematical techniques (at the cost of not allowing imperative language constructs) and the typical imperative paradigm used in Java, C and C++, in which a method call can be both a query and a modifier [Feldman 05]

This separation is very important in Design by Contract due to the fact of a contract cannot ever change the state of the program. As such contracts are to be implemented using only pure queries, which validate the program state but do not change it. To implement contracts properly, a clear way of identifying a query as pure is recommended[1].

2.3 Error Handling

The first Law of software contracting [Meyer 88c] states that a routine can only finish in two different ways: either it guarantees its contract or it doesn't. No other way for the routine

[1]Contract-Java does not deal with this issue

to finish is possible.

It shouldn't be possible to return to the caller pretending nothing happened during the callee execution, as it is possible in the large majority of languages. For instance, those which based their fault mechanism on the `try/catch` instruction.

In summary, it is impossible to ignore an exception: it has to be dealt with. The method must be fully run, meaning it must reach the end of the method. To be able to do this, a method must therefore be able to select a different approach to do its job, or clearly fail, with no manual throw required: the failure of reaching the end of the method means the failure of the whole method. The end of the method without any failure not contract violation means the method ended successfully. This is the principle of the Disciplined Exception Mechanism.

The Second Law of software contracting [Meyer 88c] states that if a callee fails, then the caller also fails to fulfill its contract.

This leads to the question, then, on how to deal with a routine failure. It is considered [Meyer 88c] only two possible approaches exist: the method must decide that either it is impossible to continue and, after guaranteeing its internal state, report failure to the caller, having then the caller then to take the same decision (the "organized panic" approach) or the "resumption" approach, in which the method attempts to fix the conditions which triggered its incapability of guaranteeing its contract and retries its body again. For example, in Eiffel, only such two responses are possible.

The DbC approach to exceptions is in stark contrast to Java's approach. In Java, an exception is possible to be ignored. The caller routine, by failing to implement the `catch` clause correctly, will resume its operation without having dealt with the error properly.

We are thus able to extract the fifth requirement:

R5-*DbC exceptions*: we need the support for fault tolerance without disruption to the code flow with a disciplined exception mechanism; a contract failure must never be possible to ignore (and any failure while executing the methods' code is also a contract failure); we need a mechanism similar to the `rescue` clause in Eiffel in opposition to the `try/catch` mechanism in Java, which allows to ignore exceptions.

2.4 DbC as Programming methodology

"Reuse without a precise specification mechanism is a disastrous risk." [Jazequel 97]

We can extract from the our description so far of DbC that, first and foremost DbC is a programming methodology, well adapted to object-oriented programming. In addition to defining a class hierarchy and methods, loosely defining their semantic using the method names and comments (which may or not be accurate), we further define those same methods with additional assertions which truthfulness is always verified at runtime. Those assertions bring only the need of being able to query the object without modification ("pure queries"); in exchange, they allow for the definition of contracts which, when thoroughly defined, allow for a simpler method body, since the method just has to deal with a very well defined condition. They allow for a clear separation of the method's real work from everything else and allow of a more efficient programming fault detection, as near as possible to the faulty section of the program.

Besides the programming methodology itself, contracts can be used to a variety of applications. For example Contracts allow the simplification of unit-tests by clearly limiting the domain of possible input, and allowing for generation of unit tests [Meyer 09] and optimizing the number of unit tests [Hakonen 11].

By allowing a stronger semantic definition of classes and methods, DbC allows for a higher level of modularization, enhancing the bottom-up approach of object-oriented programming.

Single product principle Design by Contract in its full form also allows for the "single product principle": the product is the software. All specification and documentation is in, or extracted automatically from, the software [Meyer 07].

In order to be useful, the documentation for a class must present the full overview for such class. This means that the documentation must be presented in flat form: it must contain not only the contracts that were defined on that class and its methods but as well all contracts inherited from the implemented interfaces and from its superclass, recursively. The flat documentation allows for a complete documentation of that class' semantics in one place.

R6-*documentation*: The documentation must be not only included in the code but also validated with the code as much as possible and thus be at least partially extracted from the defined contracts, forming the class and method specification. In order to properly support contracts, the documentation must support inheritance. In addition, to completely document the method/class, the documentation should feature a flat view of the documented class. Full documentation support for contracts is not only desirable but a requirement to implement Design by Contract. Contracts (and thus, the documentation they provide) are validated at every program run.

2.5 DbC versus Defensive programming

Defensive programming is considered to be an effective way to detect and handle programming errors and is the traditional approach [Aho 95, pg. 64] on the C family languages (C, C++ and now Java) being widely used on those languages libraries.

Its main idea is to detect and handle all possible errors within the program itself. For instance, a function is required to accept all possible arguments and code must be provided to test for error conditions.[2]

As such, it is considered that all possible errors must be checked and, when present, the program must provide a way to handle it. But since the error handling code belongs to the program itself, there is no clear separation between correct and erroneous programs. It encourages handling the error as late as possible, further complicating blame assignment and delaying error detection. In addition, the lack of separate error detection forgoes the possibility of clearly and programatically define the function's specification.

Listing 2.1: Example of Defensive programming: calculating sqrt

```
double calculateSqrt(double val)
{
   double value;

   if (val < 0) {
      return -1;
   } else {
```

[2]As opposed to DbC approaches, in which the function needs only to be concerned with true preconditions.

```
        value = sqrt(double x);
    }

    return value;
}
```

Listing 2.2: Example of Defensive programming: calculating sqrt

```
double calculateSqrt(double val)
{
    double value;

    if (val < 0) {
        errno = EDOM;
        return -1;
    } else {
        value=sqrt(double x);
    }

    return value;
}
```

Listing 2.3: Example of Defensive programming: calculating sqrt

```
double calculateSqrt(double val)
{
    double value;

    if (val < 0) {
        throw new Exception("val is not valid");
    } else {
        try {
            value=sqrt(double x);
        } catch {
            throw new Exception("something failed");
        }
    }

    return value;
}
```

There are several ways of implementing defensive programming. The most known is using return values to indicate unexpected conditions. The return value can specify the exact kind of error (listing 2.1) or it can be coupled with the usage of a global variable, like is usual in C programming languages with the integer global variable `errno` (listing 2.2). Another approach on defensive programming is to use exceptions to handle errors. As soon as an error is found, an exception is thrown, which should be handled by someone in a higher layer (listing 2.3).

The way all these strategies are implemented is usually through the usage of an if clause. If a certain condition occurs, depending on the programmer's approach, leave the method by throwing an exception or returning a value (possibly setting `errno`). It is also worth noting that the Java Language Specification [3] refers

[3]`http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html`

> Along similar lines, assertions should not be used for argument-checking in public methods. Argument-checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.

Such statement clashes directly with Design by Contract approach, but can be considered a side-effect of both Java implementing defensive programming throughout its libraries, as well as, more importantly, the fact that assertions in Java being possible to disable.

The programmer is encouraged to think about everything that is possibly wrong and be prepared to deal with it in the middle of the normal code flow without a clear separation. The supplier class will detect the error and return the appropriate construct, for which the client class will then check for its value and act accordingly. Since in Java the usual error reporting mechanism is through an Exception, typically a *checked exception*, this involves extra code in the client and supplier classes and, at worst case scenario, such validation is spread out throughout the method's code, hindering the code clarity and causing additional bookkeeping for the programmer.

We can consider Defensive programming as the equivalent to a Design by Contract approach in which all contracts are considered as true: in such case the method has to deal with all non-conforming input while on the opposite situation, with a strong precondition, the method only has to deal with correct input. Thus, DbC takes an opposite approach to error handling: a contract violation is always a programming error. In DbC, the input is validated as well but on a separated clause which is the contract of the supplier: the client must know the requirements (preconditions) of the supplier class and should not call it if he can't meet its demands. If it does, an exception is thrown. Also, on Design by Contract the input is to be checked at only one place, namely in the requires clause. This leads to clear code, since the implementation (work) code is free of validation code.

On the Design by Contract approach, defensive programming may only be of use when using code which is dependent on several factors which may hasten input, such as input devices or some other error prone code path.

2.5.1 Exceptions as a defensive programming construct

Exceptions in Java deal with a range of issues. For example, exceptions are used for things such invalid argument[4], which is an issue with the program, `IOException`, which can be the responsibility of the environment of the program, and internal checks[5][6]). The class hierarchy is not clear and programmers can and do develop their own exception wrapper class.

Although contracts use exceptions internally they are presented as semantically different. Exceptions should deal with something out of the ordinary, exceptional. Input validation can be dealt separately using preconditions and class and methods specifications by preconditions, postconditions and invariants. Clearly separating abnormal exceptions from normal allows for clear code, both on the client as well as the supplier and allows to establish the regular advantages of the Design by Contract programming as we refer in section 2.7.

As such, we can consider the usage of contracts, in comparison with traditional, generic exceptions, provides a greater clarity regarding the role of the exception, allows for the contract to be expressed on the interface of the class, prevents cluttering the code with excessive

[4]in Java, `IllegalArgumentException`

[5]in Java, `AssertionError`

[6]although JLS discourages usage of assertions to validate input values (2.5)

Table 2.2: Exception handling anti-patterns

Exception swallowing is the name given to a programming error in which an exception is caught and not handled properly. For example, if the catch clause is empty.

Unintended Exception subsumption is a programming error in which an exception intended to be thrown is consumed (subsumed) by a another, more generic, handler; the exception throwing is vulnerable to an overly reaching catch clause, forcing the programmer to be aware of every existing exception handling code paths.

Exception overriding is a programming error in which an exception intended to be thrown is caught and then replaced by another which is then thrown; the original one is dropped and cannot be recovered.

try/catch (for example, Eiffel's rescue clause), simplifying the understanding of the code and forces the verification of the postconditions (a badly handled, generic, exception throw would, for proper handling, require that the try/catch block to be in a loop in which the routine would be retried until successfully fulfilling the post condition). In addition, the usage of contracts emphasizes the role of the contract as a subordinate of the method and not as a use of the method and avoids the cluttering of code with try/catch to be used in regular, well-defined patterns, providing a clear blame assignment of what failed and who is the responsible for such failure (the client or the supplier).

2.5.2 Java: checked exceptions versus unchecked exceptions

In Java, an exception can be checked or unchecked. Checked exceptions have to to be declared at the method's signature and have to be caught at the level of the client method (or declared to be thrown). Unchecked exceptions have neither of such requirements. Checked exceptions also force the programmer to make a decision when he may not have full information on what the best decision may be; in addition, certain errors, such as programming errors, should not throw checked exceptions, as the client class has no way to handle them.

Checked exceptions are vulnerable to some design anti-patterns which are detrimental to their handling in which, due to programmer error, the exception is not handled properly or not handled at all [Rebêlo 11]. We detail such anti-patterns in table 2.2

As such, contract failures should be represented by unchecked exceptions, as contracts violations are expected not to ever happen in a correct program. The avoidance of required catch clauses minimizes, although does not prevent, the occurrence of such issues. Furthermore, since contracts are redundant to the main logic of the program, we should not need to add the code to deal with checked exceptions.

Finally, it is worth to refer that C# uses only unchecked exceptions [Venners 03].

2.6 Frame rules

Postconditions are used to specify what changes are supposed to happen in a class after a method is run; for example, a `pop()` on a stack is supposed to leave the stack with one less element than before. It would useful to be able to specify which attributes are not supposed to change during a method's invocation

Frame rules[7], or frame condition, allow for specifying which data is allowed to be modified by a given method, further strengthening contracts.

2.7 Summary of requirements for DbC language support

To successfully follow the Design by Contract philosophy on any language, we need various requirements.

R1 (**different assertions**) assertions define different contracts depending where in the code they are.

R2 (**locality**) contacts must be defined near the methods or classes they contractualize.

R3 (**interface**) contracts are part of a class interface, having the same importance as the class methods and variables; even if the source is not available, contracts must be.

R4 (**inheritance**) contracts are inherited: a derived class must fulfill all contacts of its parent class, as well as its method's postconditions; preconditions can, but don't have to, be loosened.

R5 (**fault tolerance**) the language must present a mechanism for dealing with faults.

R6 (**documentation**) contracts are part of the documentation which is automatically generated.

[7]Contract-Java does not support frame rules

Chapter 3

Existing language support for DbC

Various languages have support for DbC programming. On some, like Eiffel, that support is native. On others, that support may be incomplete (for example, as Java `assert` allows) or may be attempted through third party implementations or libraries. The native support has the advantage of guaranteeing the DbC support is always in sync with new language features, while the usage of a third party library adds another possible point of failure on the maintainability of existing programs. We will focus on approaches to DbC for Java, as well as some reference languages.

3.1 Analysis of native support in languages

3.1.1 Eiffel

Eiffel is a programming language designed by Bertrand Meyer which supports design by contract by design.

A typical routine is written as shown in listing 3.1[1].

We can see it supports all of the Design-by-Contract constructs, such as class invariants (`invariant` keyword) and method preconditions and postconditions (`require` and `ensure` keyword, respectively).

In addition to the support of the language to implement contracts at routines (methods) and classes, Eiffel additionally supports inheritance and contracts for deferred (abstract) classes.

Not only are contracts always inherited but there exists also a form of subcontracting, implemented using the keywords `require else` and `ensure then` at derived classes. Additionally, Eiffel supports the `old` keyword in postconditions to allow to verify an expected change occurring after the method being executed regarding a variable value prior to execution. Eiffel also implements the `result` keyword where the function return must be saved, as there is no `return` instruction.

3.1.2 Native Java

Java, as well as many other languages, provides simple assertion facilities to implement executable assertions since Java 1.4 [Rogers 01]. The keyword `assert` implements the same

[1]“deferred” is the equivalent to “abstract” in Java

Listing 3.1: Eiffel code with contracts

```
deferred class
    ACCOUNT

(...)

feature -- Element change

    deposit (sum: INTEGER)
            -- Add 'sum' to account.
        require
            non_negative: sum >= 0
    deferred
        ensure
            one_more_deposit: deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
    end;
invariant
    consistent_balance: balance = all_deposits.total

end -- class ACCOUNT
```

functionality as the `check` keyword on Eiffel. But the `assert` keyword is seen as a debug feature rather than part of the language itself and though useful on development phases, it lacks flexibility to be used as a Design by Contract. On the table 3.1 [Meyer 07] we refer the inadequacies of the `assert` keyword towards achieving the requirements in section 2.7. To summarize, the implementation of the `assert` keyword in Java is inadequate towards DbC and fails to allow the usage of contracts in a top-down implementation of a module construction.

This means a simple assertion has no semantic meaning whatsoever. As a final note, we can add that Eiffel has three main facilities which implement internally an assertion, namely preconditions, postconditions and invariants, but which are coupled with semantic meaning. Furthermore, Eiffel still has the `check` keyword, which just checks an assertion. Such check instruction is the one instruction in Eiffel which resembles Java's assert. Therefore, the simple usage of assert cannot convey the full meaning of preconditions, postconditions and invariants, but only the most basic of its meanings, that a boolean has not the expected value.

In addition to not having the keywords constructs to support Design-by-Contract, Java presents several other issues which stem not only from the language from the specification but

Table 3.1: Reasoning of assert instruction as inadequate towards implementing contracts

1. Clients cannot see asserts as part of the interface (doesn't fulfill R3 from section 2.7)
2. Not explicit whether an assert represents a precondition, post-conditions or invariant. (doesn't fulfill R1 from section 2.7)
3. Asserts do not support inheritance. (doesn't fulfill R4 from section 2.7)
4. Asserts do not yield automatic documentation. (doesn't fulfill R6 from section 2.7)

Listing 3.2: Oak attributes with contracts

```
class Calender {
  static int lastDay[12]=
    {31,29,31,30,31,30,31,31,30,31,30,31};
  int month assert(month>=1 && month<=12);
  int date assert(date>=1 && date<=lastDay[month]);
}
```

from the design philosophy of the Java platform itself.

Java violates the command-query principle separation in the language and libraries, preventing a clear separation of functionality and difficulting the implementation of sound contracts.

Most native libraries use defensive programming, adding to the complexity of the code and platform method call handling.

Java does not support natively programming by contract and none of the features described on section 2.7. Although Java uses classes and as such allows to define what could be an ADT, it can do so only using the method's names, which are insufficient to fully define the ADT, as method names are insufficient to define the semantic associated with an ADT. If contracts were possible, they would allow to define not only the semantic but to extract the class specification.

Additionally, there is no support for DbC exceptions. The `try/catch` mechanism is insufficient to properly implement contracts, as we detailed in section 2.5.1.

Oak

One of the early drafts of Java specification (being at the time the language called "Oak") supported contracts, namely defining preconditions, postconditions and class invariants, using the `assert` clause [Oak 12].

On code 3.2 we can see that the 'invariant' definition is linked to the internal representation of the class, which binds such assertions to a specific implementation, instead of making it representation invariant which would be satisfiable by any implementation of a valid date.

On snippet 3.3 we see the method contracts were to be defined in the methods code. This presents a problem when defining contracts to be applied to interfaces and abstract classes, which have no methods.

However, inherited methods could not redefine the preconditions nor postconditions [Oak 12].

3.1.3 Others

Languages such as Blue, Chrome, D, Lisaac, Nemerle and Sather support Design by Contract natively. However Blue is purely an academic programming language and Sather's development has stopped [Agostinho 08a].

Languages such as C, C++, Java (as mentioned above), Python, PHP, .NET and OCaml have the 'assert' keyword [Agostinho 08a].

C# C# supports Design by Contract, named "Code Contracts" as an add-on[2] [Cauldwell 09].

[2]`http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`

Listing 3.3: Oak methods with contracts

```
class Stack {
  int length;
  Element element[];
  boolean full() {
    /* . . . */
  };

  boolean empty() { return length==0; }

  Element pop() {
    precondition: !empty();
    /* . . . */
    postcondition: !full();
  }

  void push(Element x) {
    precondition: !full();
    /* . . . */
    postcondition: !empty();
  }
}
```

Listing 3.4: C# Code Contracts example

```
public void Transfer(Account from, Account to, decimal amount)
{
  Contract.Requires(from != null);
  Contract.Requires(to != null);
  Contract.Requires(amount > 0);
  Contract.Ensures(from.Balance >= 0);

  if (from.Balance < 0 || from.Balance < amount)
      throw new InsufficientFundsException();

  from.Balance -= amount;
  to.Balance += amount;

}
```

On listing 3.4 we can see the contracts are defined on the method's body. The consequence of this is the impossibility to easily define contracts to interfaces or abstract classes.

Ada Ada 2012 will support Design by Contract natively [Barnes 11a], without the need for any add-on; we show an example on listing 3.5 [Barnes 11b].

3.2 Java extensions for DbC

Several implementations have existed since Java inception. Most of them have been abandoned, some have remained research projects and a select few have achieved reconnaissance.

Listing 3.5: Ada 2012 example with contracts

```
generic
   type Item is private;
package Stacks is
   type Stack is private;
   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;
   procedure Push(S: in out Stack; X: in Item)
      with
         Pre => not Is_Full(S),
         Post => not Is_Empty(S);
   procedure Pop(S: in out Stack; X: out Item)
      with
         Pre => not Is_Empty(S),
         Post => not Is_Full(S);
private
      ...
end Stacks;
```

We will analyze if the most well-known or more recent solutions fulfill the requirements for Design by Contract we defined in section 2.7, as well as if they are actively developed. Unless otherwise stated, the projects have a publicly accessible source repository.

3.2.1 Jass

Jass[3] [Bartetzko 01] has no releases since 2005; source repository is not available. Contracts are implemented using comments, as we show on listing 3.6. On its webpage Jass recommends interested users to look at Modern Jass, since it does not work since Java 1.4.

Jass features contracts with preconditions, postconditions and invariants. Contracts are defined using special keywords embedded in comments. Contracts are defined near the methods or classes they apply to, but in the case of methods they are defined inside the method's body. Inheritance is not supported by default, though it may be enabled using special constructs and annotations (`jass.runtime.Refinement`).

Jass presents a mechanism for fault tolerance which allows for catching exceptions derived from contracts failure. Unfortunately, the example shown in listing 3.7, from Jass' website, presents the possibility of catching a precondition failure, which makes no sense in terms of the blame assignment: the fault of a precondition failure is of the caller of the method. Furthermore, nothing in the specification defines the lack of possibility in catching contract assertions using a `try/catch` mechanism, thus allowing the handling of assertion failures outside the scope of contracts.

Jass presents documentation support, generating additional javadoc code in the classes it processes, which allow for documenting the existing contracts automatically, thus avoiding the need of asking the programmer to manually retype all the contracts in order for them to be available in the documentation. It does not, however, generate documentation in 'flat' form.

[3] `http://csd.informatik.uni-oldenburg.de/~jass/`

Listing 3.6: Jass example code (extracted from [Shanley 03])

```
public class XOBoard {
    ...
    public XOBoard(int xdim, int ydim) {
        /** require [dimensions_ok] xdim > 2 && xdim < 10
                                  && ydim > 2 && ydim < 10; **/
        this.xdim = xdim;
        this.ydim = ydim;
        board = new char[xdim][ydim];
        clearBoard();
        /** ensure board != null; **/
}
    ...

    /** invariant xdim > 2 && xdim < 10 && ydim > 2 && ydim < 10; **/
}
```

Listing 3.7: Jass rescue example code (extracted from [Jas 12])

```
public void add (Object obj){
/** require[obj null] obj!=null; !isFull() **/
(...)
/** rescue catch(PreconditionException e){
    if(e.label.equals("obj null")) {
        o=newDefaultObject();
        retry;
    } else
        throw e;
    }
}
```

3.2.2 Modern Jass

Modern Jass[4] has no releases since 2007 and its source repository indicates it is not developed since then. It was the work of Johannes Rieken for his master's thesis [Rieken 07]. Example code on listing 3.8. The assertions implementing the various type of contracts are implemented using annotations and contracts are considered to be part of the interface, being defined near the entities they apply to. Contracts are inherited without the need of special constructs. However, unlike jass, Modern Jass doesn't feature a DbC exception mechanism, reusing normal Java exceptions to express Contract violations. Additionally, Modern Jass features no documentation support.

3.2.3 Java Modeling Language

Java Modeling Language[5] (JML) is a specification language with several implementations and its page recommends implementations other than the main one if use of Java 1.5, 1.6 and 1.7 is intended [The 12]. Listing 3.9 shows a code snippet of language annotated with JML [Pestana 09]. Assertions for the various contracts are implemented using keywords inside

[4]`http://modernjass.sourceforge.net/`

[5]`http://www.eecs.ucf.edu/~leavens/JML/`

Listing 3.8: Modern Jass example code

```
@ModelDefinitions({
   @Model(name="mTheBuffer", type=Object[].class),
   @Model(name="mStoreIndex", type=Integer.class),
   @Model(name="mExtractIndex", type=Integer.class)})
@InvariantDefinitions({
   @Invariant("size() >= 0"),
   @Invariant("(0 <= mStoreIndex - mExtractIndex) && " +
      "(mStoreIndex - mExtractIndex <= mTheBuffer.length)")})
public interface BufferSpec<T> {
   @Pure
   @Post("@Result == mTheBuffer.length")
   public int size();

   @Pure
   @Post("@Result == (mStoreIndex==mExtractIndex)")
   public boolean empty();

   @Post("@Result == (mStoreIndex - mExtractIndex == mTheBuffer.length)")
   @Pure public boolean full();

   @Also({
      @SpecCase(
         pre="o != null && !full()",
         post="@Old(mStoreIndex) == mStoreIndex - 1"),
      @SpecCase(
         pre="o == null", signals = NullPointerException.class)})
   public void add(@Name("o")T o);

   @Pre("!empty()")
   public T getNext();

   @SpecCase(
      pre="!empty()",post="@Old(mExtractIndex) == mExtractIndex - 1")
   public T remove();

   @Pure
   @Post("@Result == @Exists(Object tmp: mTheBuffer;
      tmp != null && o.equals(tmp))")
   public boolean contains(@NonNull @Name("o") T o);

   @Post("@Result != null")
   public BufferSpec<T> copy();
}
```

Listing 3.9: JML example code

```
public interface Stack {
  //@ public model instance JMLObjectSequence stack;


  /*@  public normal_behavior
    @    requires !stack.isEmpty();
    @    assignable size, stack;
    @    ensures stack.equals(\old(stack.trailer()));
    @  also
    @  public exceptional_behavior
    @    requires stack.isEmpty();
    @    assignable \nothing;
    @    signals(java.lang.Exception e) true;
    @*/
  public void pop( ) throws java.lang.Exception;
}
```

comments and they are considered as being part of the interface. Inheritance is supported with classes inheriting from superclasses and interfaces.

Regarding fault tolerance, JML implements `exceptional_behavior` which permits certain exceptions under well defined circumstances. However such behavior encourages usage of Java's exceptions rather than presenting a disciplined mechanism for dealing with faults. Finally, JML has no support for any kind of automatic documentation.

3.2.4 Cofoja

Cofoja, or Contracts for Java[6] [Lê 11], is the Design by Contract implementation made by a team of Google engineers. It's heavily influenced by Modern Jass and the contracts are, as listing 3.10 shows [cof 12], defined using annotations. Cofoja has different assertions for the different contracts and they are defined near the entities they apply to. Inheritance is fully supported, with contracts being inherited from all the class parents (superclasses and implemented interfaces). Cofoja does not present a disciplined mechanism for dealing with exceptions and also does not support automatically generating documentation from the defined contracts.

3.2.5 DbC4J

Design by Contract for Java[7], also known as DbC4J, is the prototype for Sérgio Agostinho's master thesis [Agostinho 08a]. It uses aspects to implement Design by Contract in Java while trying to avoid common pitfalls of AOP while applied to defining contracts, as detailed by Meyer's "Can Aspects Implement Contracts?" [Balzer 06], namely using AspectJ and reflection as infrastructure and implementing contracts as methods following a naming convention (listing 3.11 [Agostinho 08b]). It has seen no developments since 2008, and while the source is published, no source repository is available.

[6]`http://code.google.com/p/cofoja/`

[7]`http://aosd.di.fct.unl.pt/sergioag/prototype/`

Listing 3.10: cofoja example code

```
interface Time {
  ...

  @Ensures({
    "result >= 0",
    "result <= 23"
  })
  int getHour();

  @Requires({
    "h >= 0",
    "h <= 23"
  })
  @Ensures("getHour() == h")
  void setHour(int h);

  ...
}
```

It features different assertions for the various contracts, which are implemented using various methods with a predefined naming convention. Contracts are defined on the same file as the original methods but do not need to be near them. It supports inheritance but does not present any kind of support for DbC exceptions nor automatic documentation generation.

3.2.6 ezContract

ezContract[8] was a new way to implement contract without breaking source code compatibility and maintaining compatibility with Java IDE by using a "new marker method" [Chen 08] using various static classes as markers and using bytecode manipulation to apply contracts. Its source repository is not available. It had one binary and source release on 2008 only. We present some example code in listing 3.12 [Chen 08], where we can see that there is no clear separation in terms of language constructs of contracts and code logic: contracts are considered to be part of the implementation and not of the interface. They present different assertions for different contracts but have no support for inheritance and provide no support for any kind of disciplined exception mechanism. ezContract also does not feature any kind of support for automatically generating documentation.

3.2.7 Summary of features

Table 3.2 summarizes the features of some of the most known Design-by-Contract implementations for Java.

3.2.8 Other implementations

Many other implementations of Design by Contract are available. For a question of completeness we will try to refer all of them along a short summary next.

[8]`http://sourceforge.net/projects/ezcontract/`

Listing 3.11: DbC4J example code

```
/* original method */
public int foo() {
        // ...
}

/* precondition method */
public boolean preFoo() {
        //...
}

/* original method */
public int bar() {
        // ...
}

/* postcondition method */
public boolean postBar() {
        //...
}


/* invariant method */
public boolean invariant() {
        //...
}

/* invariant method - alternative */
public Boolean invariant() {
        //...
}
```

Table 3.2: DbC for Java features

PpC Java	R1-different assertions	R2-locality	R3-interface	R4-inheritance	R5-DbC exceptions	R6-documentation
Native Java	no	yes	no	no	no	no
jass	yes	yes	no	no	partial	partial
Modern Jass	yes	yes	yes	yes	no	no
JML	yes	yes	yes	yes	no	no
Cofoja	yes	yes	yes	yes	no	no
ezContract	yes	yes	no	no	no	no
DbC4J	yes	yes	yes	no	no	no

Listing 3.12: ezContract example code

```
public class Stack<T> implements IContract {

   (...)
   public T pop() {

      Require.begin();
         assert size() > 0 : "stack size should be > 0";
      Require.end();


      Ensure.begin();
         assert Result.value == Old.value(top()) :
            "item has been saved on top location";
         assert size() == Old.value(size()) - 1 :
            "stack size should be decreased by 1";
      Ensure.end();

      return  (...)

      public void classInvariant() {
         assert size() >= 0 : "stack size should be >= 0";
      }

      (...)
}
```

Biscotti no longer available in any form, Biscotti was a modification of JDK 1.2 compiler which added assertions to the language [Meyer 07].

C4J also known as Contracts for Java (`http://c4j.sourceforge.net/`) allows for specifying contract in Java using annotations. Contracts are specified on a different file.

chex4j (`http://chex4j.sourceforge.net/`) uses annotations to specify contracts and was inspired by contract4j. It is still being developed, with its last release done on 2011.

Contract4J (`http://www.polyglotprogramming.com/contract4j`) uses aspects to implement contracts and annotations to bind them to methods.

Contract Java (without the hyphen) is a language extension focused only on formalizing the base for correct subtyping with contracts (including interfaces and multiple inheritance) [Findler 01a]. It has not seen any further use and is not publicly available. We refer it here due to its name similarity to our work.

Custos is not longer available. It used AspectJ to implement contracts.

iContract was developed by Reliable Systems and is no longer available (its website was `http://www.reliable-systems.com/tools/iContract/iContract.htm`). It was a pre-processor which embedded assertions in special comment tags, to preserve Java source compatibility.

iContract2 was the result of an attempt to recreate iContract using a decompiler. It was a “preprocessor/code-generator” and it used annotations in comments to specify contracts. It is no longer available (its website was `http://www.icontract2.org/`); a copy is available at the JcontractS project page, referred below.

JavaDbC (`http://code.google.com/p/javadbc/`) uses AOP and annotations to define contracts. It isn’t developed since 2006.

java-on-contracts (`http://code.google.com/p/java-on-contracts/`) uses annotations to bind contracts defined on separate files. Its development has been suspended in early 2012 to allow work on a successor to C4J.

Jcontract (`http://www.parasoft.com/ibm/products.jsp`) was a commercial product which used annotations to define Contracts. It seems to be no longer available.

JcontractS (`http://jcontracts.sourceforge.net/`) is based on iContract2, which is also available on its page in its original form. It is not developed since 2008, both in terms of releases as well as source repository activity.

jcontractor (`http://jcontractor.sourceforge.net/`) is abandoned since 2004. Contracts are implemented using naming convention.

JMSAssert (`http://www.mmsindia.com/JMSAssert.html`) is a commercial product available only for Windows as a DLL. Contracts are written using JMScript. Although it is available free of charge, no source is available and its development seems to be abandoned. It is reported not to work with Java 1.4 (`http://www1.idc.ac.il/oosc/jmsassert.htm`).

OVal (`http://oval.sourceforge.net`) uses AOP for implementing contracts and annotations to define them. It is still being developed.

SpringContracts (`http://springcontracts.sourceforge.net`) uses annotations to implement contracts and has no releases since 2007.

STclass (`http://www-valoria.univ-ubs.fr/stclass`) is a "A Contract Based Built-in Testing Framework (CBBT) for Java" which allows "runtime evaluable contracts". It has no new releases since 2006 and repository activity since 2007.

Kopi is no longer available (its website was `http://www.dms.at/kopi/`). It allowed to define contracts using additional keywords.

Chapter 4

The language Contract-Java

Contract-Java language was developed to ease and to take full advantage of DbC programming within Java. As seen in the previous chapter, most implementations for Design-by-Contract for Java work as extensions to the language, usually in the form of annotations (sections 3.2.2 and 3.2.3) or by using aspects (section 3.2.4). Unlike existing approaches Contract-Java makes DbC constructs normal language entities and also fully implements[1] all six requirements for DbC support as specified in section 2.7. As such, Contract-Java is defined as a superset of the Java language (hence, "Contract-Java") with the intent of supporting DbC. All Java language is valid Contract-Java language.

To achieve a full implementation of all six requirements, Contract-Java extends Java with 9 new keywords: `invariant`, `requires`, `ensures`, `rescue`, `retry`, `check`, `local`, `old` and `result`.

Listing 4.1: Contract-Java snippet

```
public class Array<T>
{
   public Array(int size)
   requires size >= 0; ...

   public T get(int idx)
   requires idx >= 0 && idx < size();
   { ... }

   public void set(int idx, T elem)
   requires idx >= 0 && idx < size();
   {
      ...
   }
   ensures get(idx) == elem;

   ...

   public invariant (isEmpty() && size() == 0) || (!isEmpty() || size() != 0);
}
```

[1]our current prototype, however, does not

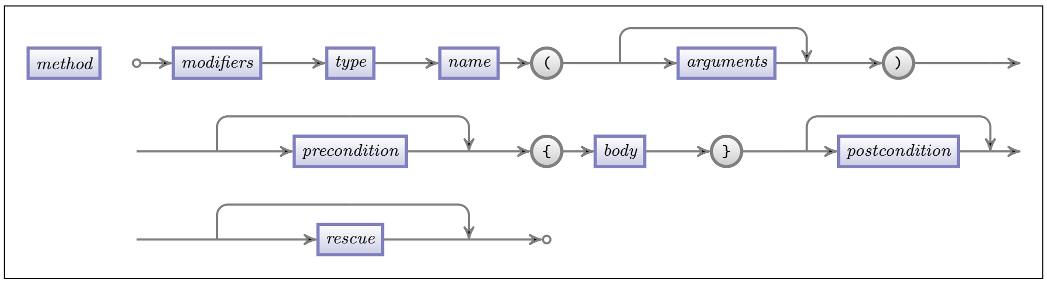


Figure 4.1: Contract-Java's method syntax diagram

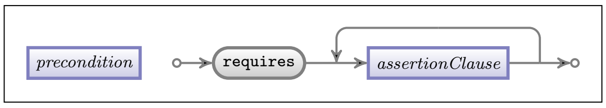


Figure 4.2: Contract-Java's precondition syntax diagram

4.1 Method Contracts

In Contract-Java the method assertions must be defined together with the method. Three new kind of assertions are possible: preconditions (using `requires`), postconditions (using `ensures`) and the DbC fault tolerant exception mechanism (using `rescue`). Contract-Java fulfills requirements R1 and R2 of DbC requirements listed in section 2.7).

The precondition and the postcondition must be placed immediately before or after, respectively, of the method's body. The rescue clause, if present, must be put after the postconditions clauses. A method may also have a `local` clause, which declares variables whose scope embraces the main body and rescue clauses, and is defined before the precondition. Methods in abstract classes and interfaces cannot have `local` nor `rescue`. Figure 4.1 describes the method definition.

Preconditions and postconditions have similar definitions, as we can see on figure 4.2 and 4.3.

We will proceed with various examples. On listing 4.2 we define contracts for a normal class and on listing 4.3 we define them for an abstract class.

The assertion clause being in a precondition, postcondition or invariant will further define the internal of a contract clause failure, but its definition is common (figure 4.4).

The assertion clause form is common to all contracts and consists in a boolean expression and optionally, any other object.

In summary, the rescue (figure 4.5) clause usually includes some code and, possibly, the

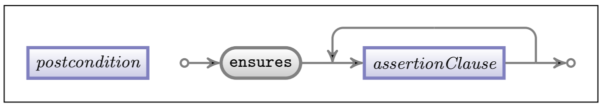


Figure 4.3: Contract-Java's postcondition syntax diagram

Listing 4.2: Example of real code of how to implement contracts on a class

```
public class Array<T>
{
   public Array(int size)
   requires size >= 0;
   {
      array = (T[]) new Object[size];
   }

   public int size()
   {
      return array.length;
   }

   public T get(int idx)
   requires idx >= 0 && idx < size();
   {
      return array[idx];
   }

   public void set(int idx, T elem)
   requires idx >= 0 && idx < size();
   {

      array[idx] = elem;
   }

   public String toString()
   {
      String result = "";
      for(int i = 0; i < size(); i++)
         result = result + " " + get(i);
      return result;
   }

   protected T[] array;

   public invariant (isEmpty() && size() == 0) || (!isEmpty() || size() != 0);
}
```

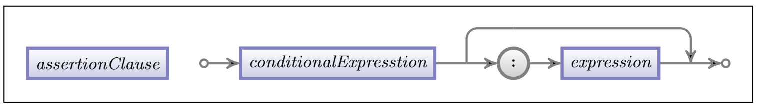


Figure 4.4: Contract-Java's assertion clause syntax diagram

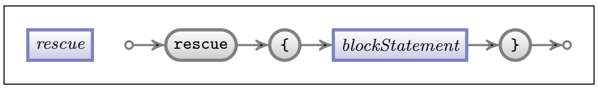


Figure 4.5: Contract-Java's rescue syntax diagram

Listing 4.3: Example of real code of how to implement contracts on an abstract class

```
abstract public class Stack<T> extends Indexable<T>
{
   public invariant isEmpty() || (top() == itemAt(0));


   abstract public Stack<T> deepClone();

   abstract public void push(T e);
   requires !isFull();
   ensures !isEmpty() && top() == e;
   ensures size() == old(size()) + 1;
   ensures tailList(1).equals(old(tailList(0)));

   abstract public void pop();
   requires !isEmpty();
   ensures !isFull();
   ensures size() == old(size()) - 1;
   ensures tailList(0).equals(old(tailList(1)));

   public T top()
   requires !isEmpty();
   {
      return itemAt(0);
   }

   abstract public void clear();
   ensures isEmpty();

   invariant (isEmpty() && size() == 0) || (!isEmpty() || size() != 0);
}
```

`retry` keyword.

Finally, contracts are inherited and must not be possible to ignore, thus fulfilling the fourth requirement of our DbC requirements (section 2.7).

Abstract classes An abstract class has the same constructs as a normal class (see listing 4.3).

4.2 Class Contracts

On figure 4.6 we present the diagram representation of class structure. The invariant is defined as in figure 4.7.

The definition of a class invariant can have the same levels of visibility of methods: public,

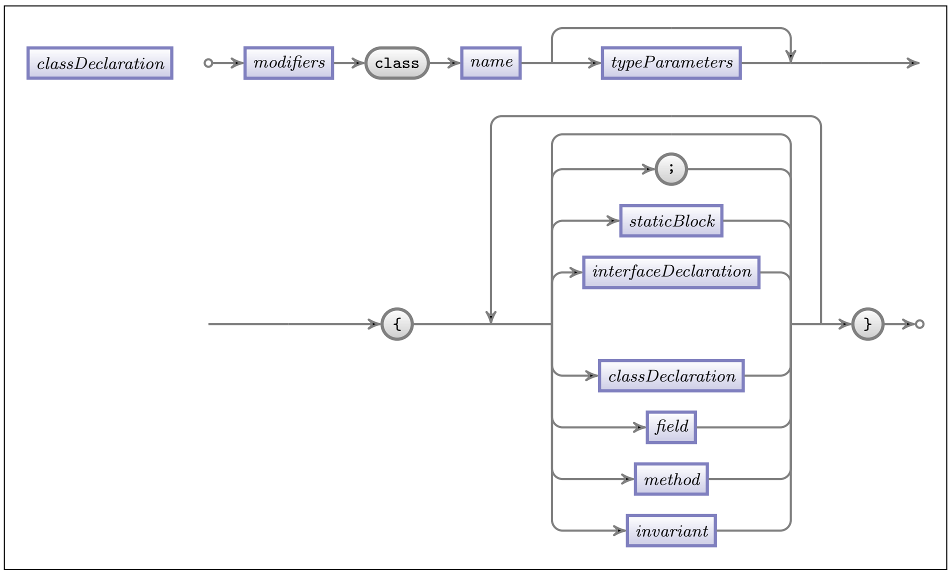


Figure 4.6: Contract-Java's class syntax diagram

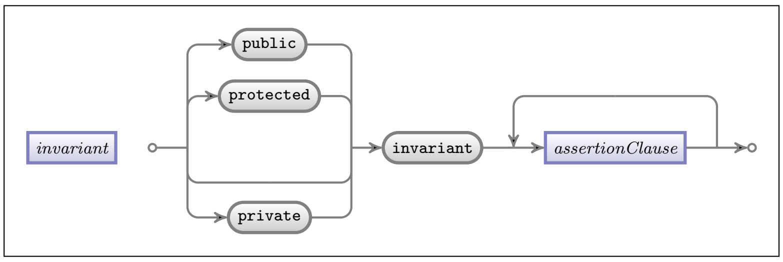


Figure 4.7: Contract-Java's invariant syntax diagram

package, protected and private, being the default one, when not specified, package, similarly to Java.

A public invariant can define the ADT properties (representation invariant), while a non-publicly accessible invariant will ensure the class internals are working properly, without adding further relevant information to the ADT specification.

When generating documentation for the ADT, the non-public invariants are omitted.

4.3 Java interfaces

Interfaces specify an ADT while not providing any kind of implementation. As such, for the ADT to be completely specified, contracts need to be supported on interface classes. In the same way Native Java's throws are considered part of the class interface, contracts also belong to it. By implementing an interface, a class inherits the interface contracts. The same rules apply to inheritance on interfaces as to normal classes: interfaces are also ADT definitions and as such have the same treatment.

Listing 4.4: Defining contracts on an interface

```
interface InterfaceStack<T> extends InterfaceIndexable<T>
{
   public invariant isEmpty() || (top() == itemAt(0));

   public void push(T e);
   requires !isFull();
   ensures !isEmpty() && top() == e;
   ensures size() == old(size()) + 1;
   ensures tailList(1).equals(old(tailList(0)));

   public void pop();
   requires !isEmpty();
   ensures !isFull();
   ensures size() == old(size()) - 1;
   ensures tailList(0).equals(old(tailList(1)));

   public T top();
   requires !isEmpty();

   public void clear();
   ensures isEmpty();

   public boolean isEmpty();

   public boolean isFull();

   public int size();
}
```

4.4 DbC exceptions

The fifth requirement of our DbC requirements (section 2.7) is support for DbC exceptions. A method either succeeds or fails, reporting the fault to the client.

We reuse the exception handling mechanism of Java, while keeping the independence between normal Java exceptions and DbC exceptions. One mechanism cannot interfere in any way with the other: they are totally orthogonal. To achieve so, we forbid the usage of `Contract_JavaAssertion` or any of its derived classes on our compiler. Furthermore, any failure which is not a Contract Java contract failure does not trigger rescue execution.

In order to keep both exception mechanism working, we need to enforce the separation between both kinds of exceptions (Contract-Java exceptions versus "normal" ones). It is thus possible to use `try`/`catch` program exceptions except for Contract-Java exceptions, as a way to allow interfacing with modules written using defensive programming. Even when catching Throwable does not result in Contract-Java exceptions being caught.

It is important to emphasize the `rescue` clause and the `try`/`catch` are two different and independent methods of dealing with exceptions. The `try`/`catch` mechanism cannot be used to catch contract failure exceptions. The Contract-Java language guarantees this, meaning the `try`/`catch` mechanism is orthogonal to contracts' handling using DbC exceptions (2.7); full support for DbC exceptions is thus available. So we have the legacy handling of exceptional behaviour, using the `try`/`catch` mechanism, in which the failure conditions are part

Listing 4.5: Example of real code of how to define a rescue clause on a class

```
public class Array<T>
{
   (...)

   public boolean equals(Array<T> other)
   requires other != null;
   {

      boolean result = (size() == other.size());

      for(int i = 0; result && i < size(); i++)
         result = get(i).equals(other.get(i));

      return result;
   }
   rescue {
      // various code
   }
}
```

of a method implementation and thus not visible to the outside, and the support for DbC exceptions, in which the failure conditions are visible and part of a class' interface.

The rescue clause is associated to a real, non-abstract, method. If the failure is of the method's responsability (i.e., not on a precondition) then the execution jumps to the rescue close where the failure is attempted to be dealt with. If the rescue clause reaches its end without a retry, or there is no rescue clause, the Contract-Java exception is rethrown, in order for the upper level of execution to be able to decide what to do with the error: either recover for it, or throw it again to its caller. In case a retry is done, the execution restarts on the beginning of the method; only local variables will keep its state.

4.5 Debugging in Contract-Java

4.5.1 Error

When checking for an assertion the programmer can define an appropriate error message to be used when that assertion fails. For example, the program displayed on listing 4.6 would yield the result in listing 4.7.

On Contract-Java the error messages associated with the assertion failures are automatically filled with the boolean expression that failed [2], in addition to expanding the various expression values and printing on the screen (and, optionally, normal text), thus reducing the need of manual definition of the assertions' associated text and providing a clearer view of why the assertion failed. We present an example program in listing 4.8 and a possible output in listing 4.9.

[2]this feature hasn't been implemented

Listing 4.6: Simple assert usage

```
public class TestAssert {

    static boolean boolFunc01() { return false; }
    static boolean boolFunc02() { return true; }
    static boolean boolFunc03() { return false; }

    public static void main(String[] args) {
        assert (boolFunc01() && boolFunc02()) ||
                boolFunc03() : "unhelpful assertion";
    }

}
```

Listing 4.7: Output example of Native Java's assert

```
$ java -ea TestAssert
Exception in thread "main" java.lang.AssertionError: unhelpful assertion
    at TestAssert.main(TestAssert.java:16)
```

Listing 4.8: Simple precondition usage

```
public class TestPrecondition {

    static boolean boolFunc01() { return false; }
    static boolean boolFunc02() { return true; }
    static boolean boolFunc03() { return false; }

    public void doSomething()
    requires (boolFunc01() && boolFunc02()) || boolFunc03();
    {
      ...
    }

}
```

Listing 4.9: Example of a possible output of Contract Java's boolean expansion

```
$ java TestBooleanExpansion
Exception in thread "main" Contract_JavaPreconditionFailure
    at TestBooleanExpansion.main(TestBooleanExpansion.java:16)
Precondition failed: boolFunc01() && boolFunc02() || boolFunc03()
   boolFunc01() && boolFunc02() || boolFunc03() => false;
   boolFunc01() && boolFunc02() => false;
   boolFunc01() => false;
   boolFunc02() => true;
   boolFunc03() => false;
```

4.5.2 Fine-tuning

There should be the possibility of fine-tuning on enabling and disabling contracts[3]. The language should present an option of enabling and disabling contracts at the global, package or class level, at compile time (in opposition to Native Java's enabling of assertions at runtime), for example, using a configuration file. In addition to fully enable or disable contracts in selected locations, we need a way to disable only some kind of contracts. Typically in development, all contract-checking is enabled, while in production only preconditions are checked.

4.6 Pure queries detection

Contract-Java should only allow pure queries to be used in contract definitions. In order to do so, Contract-Java should be able to find out if the queries used in the contracts are pure and prevent them from being used inappropriately. This could be done on compile time, taking an extra step to determine if a certain method is pure and marking it internally as such. There is also the possibility of defining a new keyword to do such, however that would be an extra burden for the programmer and it would be another source of errors if a previously pure method would change and the keyword were not to be removed; furthermore, it would be impossible to use methods which were not Contract-Java, as they would not be properly annotated.

However, neither of these strategies has been implemented and as such, on Contract-Java, the responsibility of using appropriate queries in contracts is left to the programmer.

4.7 Other assertions

We support the equivalent to the `assert` keyword from Java, namely `check` (shown on listing 4.10). `check` allows for the verification of a boolean expression triggering a failure when it is not true, of type `checkFailure`.

It is worth noting that we do not support loop variants and loop invariants. Though important, through their validation of algorithms used inside a program, they do not directly contribute towards the DbC requirements (section 2.7).

4.8 Contract-Java native library

Contract-Java does not support contracting preexisting class files. The alternative is to create wrapper classes in order to encapsulate access to a preexisting class file, adding the desired contracts on the wrapper class.

Since Java uses defensive programming, Contract-Java would probably benefit from an effort to contractualize Java libraries providing new libraries which wrap and contractualize native libraries, which would lead to new classes[4] being defined.

[3]not implemented

[4]with a different ADT, since Java native classes follow defensive programming and have no regard for command-query separation; also, consider LinkedHashMap, in which a boolean in the constructor changes the class behaviour – access to the HashMap may then lead to a structural modification

Listing 4.10: Example of real code of how to define a check clause on a method

```
public int pictureMosaic(ArrayList<String> paths) {

   ...
   PictureMosaic album = new PictureMosaic();

   while (...) {
      ...

      album.tryAdd(paths.get(i));

      check album.isPresent(paths.get(i));

      ...

   }

   ...
}
```

4.9 Documentation

The support for full automatic documentation generation is the sixth requirement of our DbC requirements (section 2.7). All contracts (with the exception of non-public invariants) must be extracted from the definition and automatically added to the documentation. In the cases where inheritance is involved, each class documentation should allow a full listing of the ADT definition, namely make documentation available in “flat” form (which includes all available public methods, their contracts and the class public invariant). We intend to have another tool, “cjavadoc”, to extract the documentation.

Chapter 5

Implementing Contract-Java

One of the main ideas of the implementation of Contract-Java compiler prototype was to reduce the "distance" between the generated Java source code and the original Contract-Java file. Doing such allows for easier debugging of the compiler, faster implementation and facilitates feedback from users, being the best strategy for the first phase of developing our prototype. As a consequence, we preserve as much as possible from the original input, such as white-space, comments and javadoc.

Implementing the prototype for our Contract-Java compiler involved several phases. We will detail them next.

5.1 Work strategy

In order to ease the creation of the first version of the tool we decided to implement a Contract-Java to Java compiler, allowing us to focus on the new constructs while ignoring low-level details, which we wouldn't be able to do if we had decided to translate to a lower-level language such as JVM bytecode.

$$\begin{tikzcd}
\boxed{\text{Contract-Java}} \arrow[r, "\texttt{cjavac}"] & \boxed{\text{Java}} \arrow[r, "\texttt{javac}"] & \boxed{\text{Java bytecode}}
\end{tikzcd}$$

In order to attain that objective, it was necessary to research available parser generators which would allow us to easily implement our compiler.

The two most widely known parser generators are bison/yacc and ANTLR [1]. We chose ANTLR over bison/yacc because ANTLR has additional features, such as built-in support for constructing Abstract Syntax Trees, is made in Java, avoiding dealing with another language such as C, and its webpage already had a full featured Java 1.6 grammar.

As a side note, although in the literature a top-down approach ("recursive descent"), such as ANTLR's is less powerful than a bottom-up one ("recursive ascent"), such as bison's, ANTLR is able to handle Java-based grammars with no issues.

[1]We used ANTLR 3.4; ANTLR 4 is yet under heavy development and thus not appropriate for general use

Syntactical Analysis
Semantic Analysis
Code Generation

Figure 5.1: Compiling phases

5.2 Implementation strategy

5.2.1 Choice of grammar

Several grammars for Java are available at ANTLR's website[2].

Having several grammars available for Java for use with ANTLR, we decided to use the most recent one. It has been developed by the "Compiler Grammar" project of the OpenJDK group with the intent of replacing the hand-written LALR parser[3] used by the javac compiler [Com 12].

We used the lexer and parser from the project while avoiding the tree grammar also developed by the project, since it implemented javac specific AST nodes which were deemed not necessary for the objectives we intended to attain.

Another reason which led us to choose this grammar was that after it was integrated with the javac compiler, the resulting tool was validated by the Java Compatibility Kit, furthermore supporting the validity of the grammar as a proper Java grammar.[4]

Despite such benefits, we must also note that the OpenJDK project "Compiler Grammar" has not accompanied the new Java 7 development since such the grammar has not been further updated. Since the JLS is publicly available such endeavor should be trivial. For the sake of completeness, however, we must note that none of grammars available on the Java Language Specification (JLS) totally conforms to javac compiler internal parser [Com 12].

5.2.2 Compiling phases

We use a three-step compiler (see figure 5.1), constructing a simple AST from the parser output. Implementation of an AST was not strictly required but allows for future implementation of semantical validation to the Java source code. The three phases are detailed on table 5.1 – it must be noted that the semantic check phase is very incomplete. Every of such phase is suppose to be totally completed before proceeding to the next one.

The Contract-Java source files are to be compiled are analyzed for dependencies and additional files deemed to be necessary for successful compilation are compiled as well, mimicking the standard javac compiler.

However, we do not currently do semantical validation, deferring such validation to the Java compiler. This means generated code may not be valid but does not, however, imply the lack of validity of our prototype.

[2]`http://www.antlr.org/grammar/list`

[3]the email archived at `http://mail.openjdk.java.net/pipermail/compiler-grammar-dev/2010-February/000034.html` mentions the "current version of javac is a hand-written recursive descent parser, augmented with operator precedence parsing for binary expressions"; however it is irrelevant for the current discussion

[4]simply put, the Java Compatibility Kit is the testsuite which Java implementations must pass to prove they support one or several JSR (Java Specification Requests) and thereby be able to claim to be Java compatible.

Table 5.1: List of phases Contract-Java compiler goes through

phase	description
syntactical analysis	in this phase we use ANTLR for parsing all the program files and appropriately tagging the required text in order to use it later
semantical analysis	in this phase we verify if the symbols used throughout the program exist and warn the user if necessary
code generation	in this phase we finally generate the appropriate code, to a temporary Java file, and compile it, deploying transparently on the same directory as cjava file

5.2.3 Code Generation

ANTLR uses and includes a tool called StringTemplate, by the same author. Being a "java template engine" [Str 12] it allows for generating source code, as well as other formatted text output.

Since it is already included with ANTLR it was the ideal candidate to generate the new code. We handle most list of arguments using StringTemplate; it allows more flexibility on handling exactly how to treat each of the list items, since we can easily apply a rule to any list item or print it as is, with no processing. StringTemplate coupled with ANTLR's `TokenRewriteStream` allows us to insert new code without needing to manually output the original code.

StringTemplate is thus responsible for generating all the text used in the output of our compiler and in the following sections we will show how those templates are used to construct the intended behaviour.

5.3 Method Contracts

In order to better deal with the fact that Java can have multiple return points, we decided to wrap the methods in another one, which would be responsible for calling the precondition statements, saving the return value and calling postconditions and rescue clauses. Thus, we implemented the calling of contracts by renaming the original methods (using the original name prefixed by `_cj_Wrapped_`) and creating new methods which dealt with calling the contracts and afterwards, called the actual method. This allows us to deal properly with the return value, which would otherwise require us to track all the returns and add the appropriate code several times.

The wrapping of functions in others which handle contract calling facilitates the implementation of contracts inheritance, the saving of the return value and handling of old values.

We decided to name those extra methods as the `_cjRequire` and `_cjEnsure` plus the name of method. The invariant is stored with the name `_cjInvariant` plus visibility, with `_cjInvariant` being responsible for calling all of them. In listing 5.1 we present a snippet of code generated by our compiler. All our methods verifying contracts are `void` and do nothing if the assertion is true. If it does not verify, a class derived from `Contract_JavaAssertion` is thrown.

A snippet of the code generated for the example in the previous chapter (listing 4.2) is

Listing 5.1: Example of generated code of how to implement contracts on a class methods

```
public class Array<T>
{
    ...

    public void _cjRequire_get(  int idx) {
        if (! (idx >= 0 && idx < size())) throw
            new Contract_JavaPreconditionFailure();
    }

    public void _cjEnsure_get(T _cjReturn,    int idx) {
    }

   /**
    * Gets array's element at {@code idx} position.
    */
   public T get(int idx)
   {
    T _cjReturn;
    _cjRequire_get(idx);
    _cjInvariant();

    _cjReturn=_cj_Wrapped_get(idx);

    _cjInvariant();
    _cjEnsure_get(_cjReturn, idx);
    return _cjReturn;
   }

}
```

presented in listing 5.1 and for the invariant on listing 5.2.

Contract-Java handles inheritance in the following way: if a class is derived from one which has contracts, those contracts will be called every time the method from the derived class is called. There is no inheritance for private methods, constructors and static methods and as such no contracts are inherited as well. The generated code is similar of abstract methods. The contracts redefinition is done using Meyer's approach, as referred on section 2.2.2.

5.4 Class Contracts

Invariants can be, in terms of visibility, public, protected, private and package-only. In listing 5.2 we show the result of a possible code generation by the Contract-Java compiler.

5.5 Java Interface

Interfaces imply a specific handling of the contracted code, since interfaces can't have code associated. Contract-Java supports setting contracts on interfaces and handles them internally accordingly. Definition of contracts applied to interfaces are created in a inner class inside the interface, thus preserving the one to one file generation we have on the other code generation.

Listing 5.2: Example of generated code of how to implement contracts on a class

```
public class Array<T>
{
    ...

    public void _cjInvariantPublic() {
        if (! ((isEmpty() && size() == 0) || (!isEmpty() || size() != 0)) throw
            new Contract_JavaInvariantFailure();
    }

    public void _cjInvariant() {
         _cjInvariantPublic();
    }

    ...
}
```

Listing 5.3: Example of a possible code generation to handle interface contracts

```
public interface InterfaceIndexable<T> {

   public static class contractJava() {
      static public void _cjRequire_get(InterfaceIndexable that, ...) {
         ...
      }

      static public void _cjEnsure_get(InterfaceIndexable that, ...) {
         ...
      }

      static public void _cjInvariant(InterfaceIndexable that) {
         ...
      }

      static public void _cjInvariantPublic(InterfaceIndexable that) {
         ...
      }

   }

   boolean get(Object o);
}
```

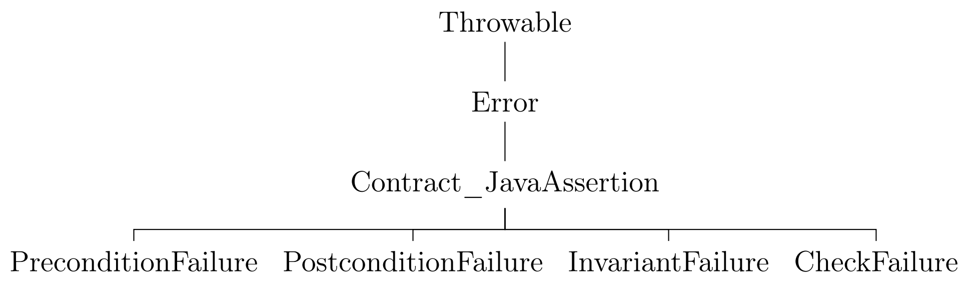
Listing 5.4: Example of a possible code generation to handle existing exceptions on a method

```java
public double _cj_Wrapped_somethingBeingDone(double parameter) {

   /**
    * this code is the normal code inserted by the programmer
    * we handle a non-contracted Stack
    */

   Stack stc = new Stack();
   try {
      stc.pop()
   // generated line follows
   } catch(Contract_JavaAssertion cja) {
      throw cja;
   // original catch
   } catch(EmptyStackException ese) {
      throw new Error("failure in Stack");
   }
}
```

5.6 DbC exceptions



The support for a disciplined exception mechanism (section 2.3) is one of the key features of Design by Contract. It is implemented recurring to the `rescue` clause. The existence of the `rescue` allows for decluttering the code from error handling. A failure of a method in the body of the method corresponds to a postcondition failure, and its recovery is done through rescue clauses.

Since we are implementing our language as a superset of Java, and although the ideal usage would be to not have to deal with exceptions in our method body, we need to handle existing exceptions in such a way as to not interfere with Contract-Java exceptions. On listing 5.4 we present the proposed solution, in which we always modify the `catch` clause to catch `Contract_JavaAssertion` and rethrow it, to ensure the original `catch` cannot ever catch it.

On listing 5.5 we define a possible rescue clause implementation.

5.7 Other assertions

The `assert` instruction will still work as in native Java and it's handling is not changed in any way, since it is considered a separate assertion mechanism of those implemented by

Listing 5.5: Example of a possible code generation to handle a rescue clause on a class

```
public int methodName(int parameter) {
    int _cjReturn;

    /* "local" */
    int example_attempts = 0;
    int example_limit = 2;
    boolean _cjRetry;


    do {

        invariant();
        precondition(); // caller responsability, out of rescue reach

        _cjRetry = false;
        try {
            _cjReturn=realmethod(int paramenter);
            postcondition(int _cjReturn, int paramenter);
            invariant();
        } catch(Contract_JavaAssertion ex) {
            example_attempts++;

            if (example_attempts == example_limit)
                // this is the text which "retry" generates
                { _cjRetry = true; continue; }

            //else fail:
            throw ex;
        }
    } while(retry)

    return _cjReturn;
}
```

Contract-Java. However, we define the `check` instruction which generates an exception which is dealt by our disciplined exception mechanism. On listing 5.6 we have the generated code for the code present in listing 4.10.

As other assertions, the `check` instruction can be deactivated at compile-time.

5.8 Contract-Java native library

We present a simple implementation of Stacks, Queue and AssociativeArray on appendix A, in order to provide a simple but complete example of the usage of contracts on full classes.

5.9 Documentation

`cjavadoc` hasn't been implemented yet, but it should extract all information needed from contracts and adds them to the class documentation; the usage of javadoc as a backend should be possible in order to reduce the complexity of `cjavadoc`.

Listing 5.6: Example of a possible code generation to handle a check assertion on a method

```
public int _cj_Wrapped_pictureMosaic(ArrayList<String> paths) {

   ...
   PictureMosaic album = new PictureMosaic();

   while (...) {
      ...

      album.tryAdd(paths.get(i));

      if (! (album.isPresent(paths.get(i)) != true)) throw new
         Contract_JavaCheckFailure("album.isPresent(paths.get(i)) != true",
         /* originalFile */ "pictureClass.cjava", /* original line */ "17"),
         true);

      ...

   }

   ...
}
```

5.10 Contract-Java notes/issues

5.10.1 Line numbers

Contract-Java strives to preserve as much input as possible, thus, when errors are present in generated source code, they are easy to track. Nevertheless, the fact that Java compiler errors do not correspond to the original file is a problem from the usability perspective of the tool. Since we don't yet have a full semantical validation, it would be desirable to have a clear proper handling of Javac's errors, since there will be errors which will only be caught by javac but not Contract-Java compiler.

On a C based language we could use the `pragma` operator to define which source file line was responsible for generating a certain line in the intermediary Java file would link. On Java such solution isn't possible, since there is no preprocessor.

If a complete semantic check proves unfeasible, it may be possible to workaround the issue using a "Diagnostic Listener"[5] which would translate errors' line numbers from Java files to the corresponding line number on the original CJava file.

5.10.2 Usage

Contract-Java provides it's own program to compile both Java programs as well as CJava, providing a drop-in replacement to calling javac (internally, it calls javac when appropriate).

Listing 5.7: Invoking the Contract-Java compiler

```
$ java -jar CJ2J.jar <list of Java or CJava files>
```

[5] `http://docs.oracle.com/javase/6/docs/api/javax/tools/DiagnosticListener.html`

All command-line options are properly passed on to javac, with the exception of “@” (to load options from a file), and “-d” (which is needed internally by the Contract-Java compiler and thus ignored with a warning when specified from the command-line).

5.11 ANTLR issues

Throughout the development of our prototype, we found some issues regarding the usage of ANTLR. Those issues have arisen mainly from documentation omissions, including the lack of documentation of known bugs in the code. We will detail them in the following subsections.

5.11.1 Matching several text tokens

After the parse phase ANTLR provides us with two alternatives: a text-based output using StringTemplate or an Abstract Syntax Tree (AST); from an AST we can always generate a text-based alternative, if we wish to.

What we did internally was to extract certain ranges of text, namely from functions calls, and reuse them on the several snippets of code we generated. Such operation is easier on the parser than on a tree parser phase, as happen when working with an AST[6]. To do it properly, we were required to enforce hierarchy on the various tree nodes, using imaginary nodes as helpers. However, when using ANTLR's syntax, we were unable to extract all text below such new nodes. We were required to use the ANTLR's internal API to get all the text below such nodes.

5.11.2 Documentation caveats

ANTLR is a free product, available under the BSD license since version 3 [7]. The author has, however, two non-free books available to buy. "The Definitive ANTLR guide"[8] and "Language Implementation Patterns[9].

Probably as a side effect of such strategy, the documentation available for the site is incomplete and loosely connected, and should be seen only as a complement to the books.

Furthermore, the API documentation is in some cases incomplete, not providing a flat approach to inheritance with the effect of this hiding some implementation details of ANTLR which should have found earlier.

Even so, most documentation available details simple transformations of code, which are usually achievable using an Abstract Syntax Tree (AST). However, in the case of complex translations (namely, involving saving and deletion of text and insertion of text on different areas) ANTLR's syntax presents no solution, having forced us to hook the ANTLR's lower level API.

[6]an 'upper' level rule only matches the leftmost token child instead of the whole tokens; e.g., the rule 'booleanExpression' matches only the first token

[7]`http://www.antlr.org/license.html`

[8]Parr, Terence, "The Definitive ANTLR guide", Pragmatic Programmers

[9]Parr, Terence, "Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages", Pragmatic Programmers

5.11.3 Invalid code generated

ANTLR 3.4.0 (the latest stable version) has an undocumented bug which leads it to generate invalid code when assigning labels to rules on a parser grammar (but not on a tree grammar).

Such problem was first solved using an older ANTLR version (3.3)[10] and afterwards using an unreleased version, available on ANTLR's source repository, as per instructions received on the support mailing list [11]; that version will eventually be released as ANTLR 3.5[12].

5.11.4 Composite grammars

ANTLR's primary mode is to work with a so called "combined grammar". Such grammar is a file consisting of both lexer and parser rules, which define the grammar in its entirety.

However ANTLR has a functionality called "composite grammars". Following the idea behind object-oriented programming, ANTLR allows for a grammar to extend another. Since Contract-Java extends Java, it would make sense to separate the Java grammar for the Contract-Java grammar, allowing for a clear separation of the languages and easier maintenance.

Despite being a good idea, composite grammars presented more issues. Besides finding out could not import a lexer from another lexer, we also triggered an infinite loop on the grammar processing and found out a lexer rule is always required on the main grammar file.

We eventually workarounded such issues but in the end the parser generated from a composite grammar didn't work properly with any of the tested versions of ANTLR (3.3, 3.4 and what will be 3.5[13]). As such, we decided not to follow such code organization.

This does not mean we cannot maintain a different grammar for Java and Contract-Java and eventually merge the changes from one to the other, but such bugs imply the approach we can take to get such advantages will probably be more error-prone (e.g., we can use a set of diffs between our grammar and the original Java.g and port the changes from a more recent Java.g to the older Java.g and from there to the Contract-Java grammar).

5.11.5 Backtracking and Error handling

Since any of the Java grammars available for ANTLR uses backtracking by default, unambiguously reporting errors is difficult. To be able to properly report errors, it would be required to refactor the grammar in order to remove left-recursion, which would render it more difficult to maintain and update. For the purpose of building a prototype, we decided to use the grammars already available, which require backtracking.

[10]it may be worth explaining in versions prior to 3.4 ANTLR was dependent on ANTLR 2 to build, which had a different license and led to projects such as Eclipse to not include ANTLR on their projects, as explained on `http://www.antlr.org/wiki/display/ANTLR3/ANTLR+3.4+Release+Notes`; version 3.4 was therefore a complete rewrite of ANTLR's built-in grammars and as such, the appearance of such bug is understandable

[11]http://thread.gmane.org/gmane.comp.parsers.antlr.general/34301/focus=35845

[12]http://markmail.org/thread/vqzeptcda65xpdzz

[13]we used a SNAPSHOT version

Chapter 6

Conclusions

We achieved a significant part of the objectives which were intended. After surveying existing options and defining the requirements for a new language which implemented contracts on Java, we defined a new language, Contract-Java, which treats contracts as normal languages entities, defined the support for inheritance and incentives the usage of Design by Contract approach as a program development methodology. We implemented a prototype which, while not supporting Design by Contract to all language constructs, demonstrates the feasibility of the language Contract-Java.

6.1 Future work

First of all, we intend to make the Contract-Java compiler a more usable prototype or, if possible, a complete product. Several choices were made during the this work which we consider useful to revisit sometime; on the other hand, some approaches of our prototype can be enhanced. A brief description of the different topics follows.

Framerules We should be able to specify frame rules on the language, as a complement to the specification of contracts.

Loop variants/invariants Loop variants and invariants ease the verification of the proper functioning of loop contructs. Although not required to implemented DbC, they represent a useful addition to the language.

Concurrency The usage of contracts within a language may be a way to implement more efficient concurrency within a language. Contract-Java could expand in that direction.

New operators The construction of contracts could be enhanced by supporting JML constructs such as equivalent ("<==>"), implies ("==>") and is implied ("<==").

Support for pure query detection Contract-Java should be able to automatically determine if a query is pure or not, and forbid accordingly its usage on a contract checking.

Debugging in Contract-Java A contract failure should present the user with not only the line of the contract which failed but an expansion of the boolean expression values, for an easier interpretation.

Documentation We should also generate flat documentation, extracting the contracts from the method's interface and adding them as Javadoc on the various methods and classes.

Native library Defining a new language on top of Java is an important step towards expanding the ease of use of Design by Contract programming but such step is more easily done with the support of a preexisting, contracted library. The idea will be, in the future, to implement a Contract-Java library which contractualizes preexisting Java classes, to allow for seamless use in contracted classes.

IDE support We are also considering the feasibility of implementing IDE support for Contract-Java on a later phase.

Semantical validation In the future we intend to add support for proper inheritance support and semantic validation. The Contract-Java translator should support automatic generation of contract documentation and present it in flat form.

The usage of semantic validation should allow us to:

- verify if an assertion is boolean;
- deny extending `Contract_JavaAssertion`;
- enforce usage of public methods by public contracts;
- forbid class attributes to be public;
- type consistency;
- an finally, prevent errors from being reported in a more confusing way by the regular `javac`.

Grammar choice Although the OpenJDK's grammar has been fully tested and integrated in the javac compiler, that came at the cost of having a more complicated grammar: while that cost was negligible as long as the OpenJDK grammar was maintainted by the OpenJDK project, once that maintenance is no longer kept the onus of updating it is probably bigger than using the simpler Terence Parr's grammar; revisiting the choice of grammar may be of use for future work.

Internal organization The way the conversion of the Contract-Java files is done internally is to handle strings of text which are copied from various places to another. Such approach was the easiest but in the long-term may prove to be unmaintainable and, as such, should probably be converted on a AST manipulation coupled with AST to text generation.

Appendix A

Example code

The following code serves as first approach to the implementation of a Contract-Java library.

Listing A.1: The implementation of an Array in Contract-Java

```
/**
 * Generic array module.
 */
public class Array<T>
{
   /**
    * Creates an array with {@code size} elements.
    */
   @SuppressWarnings(value = "unchecked")
   public Array(int size)
   requires size >= 0;
   {
      array = (T[]) new Object[size];
   }

   /**
    * The array's immutable size
    */
   public int size()
   {
      return array.length;
   }

   /**
    * Queries equality with {@code other} array.
    */
   public boolean equals(Array<T> other)
   requires other != null;
   {

      boolean result = (size() == other.size());

      for(int i = 0; result && i < size(); i++)
         result = get(i).equals(other.get(i));

      return result;
   }
```

```
    /**
     * Gets array's element at {@code idx} position.
     */
    public T get(int idx)
    requires idx >= 0 && idx < size();
    {
        return array[idx];
    }

    /**
     * Sets array's element at {@code idx} position.
     */
    public void set(int idx, T elem)
    requires idx >= 0 && idx < size();
    {

        array[idx] = elem;
    }

    /**
     * Returns a {@code String} with all array's elements separated with a space
     */
    public String toString()
    {
        String result = "";
        for(int i = 0; i < size(); i++)
            result = result + " " + get(i);
        return result;
    }

    public void resize(int newSize) {
        T[] newArray = (T[]) new Object[newSize];

        int oldSize = size();
        int minimum = (oldSize < newSize) ? oldSize : newSize;

        for (int i = 0; i < minimum; i++) {
            newArray[i] = array[i];
        }

        array = newArray;
    }

    protected T[] array;
}
```

Listing A.2: We define an abstract class Listable which will be reused by the following classes.

```
/**
 * Generic listable list.
 */
abstract public class Listable<T>
{
    public invariant size() >= 0;
    public invariant (isEmpty() && (size() == 0)) ||
        (!isEmpty() && !(size() == 0));

    /**
     * The list total number of elements.
     */
    abstract public int size();

    /**
     * Queries the list emptiness.
     */
    public boolean isEmpty()
    {
        return size() == 0;
    }

    /**
     * Queries the list fullness.
     */
    abstract public boolean isFull();

    /**
     * Checks if list is unbounded.
     */
    abstract public boolean isLimited();

    /**
     * The list maximum number of elements.
     */
    abstract public /*@ pure @*/ int maxSize();
    requires isLimited();

    /**
     * Extracts all the elements of the list.
     */
    abstract public Array<T> toArray();
    ensures result.size() == size();

    /**
     * Clears the list.
     */
    abstract public void clear();
    ensures size() == 0;
}
```

Listing A.3: A generic stack partially implemented.

```
/**
 * Generic stack module.
 */
abstract public class Stack<T> extends Listable<T>
{
    public invariant isEmpty() || (top() == toArray.get(0));
    protected boolean limited;

    /**
     * Adds an element to the top of the stack.
     */
    abstract public void push(T e);
    requires !isFull();
    ensures !isEmpty() && top() == e;
    ensures size() == old(size()) + 1;

    /**
     * Removes the top element from the stack.
     */
    abstract public void pop();
    requires !isEmpty();
    ensures !isFull();
    ensures size() == old(size()) - 1;

    /**
     * Without changing the stack, returns its top element.
     */
    abstract public T top();
    requires !isEmpty();

    /**
     * Empties the stack.
     */
    abstract public void clear();
    ensures isEmpty();

    public boolean isLimited()
    {
        return limited;
    }

}
```

Listing A.4: The full implementation of a Stack, using an Array as the internal representation.

```
public class ArrayStack<T> extends Stack<T> {

    ArrayStack() {
        my = new java.util.ArrayList<>();
        limited = false;
    }

    ArrayStack(int maxSizeIn)
    requires maxSizeIn >= 0;
    {
        my = new java.util.ArrayList<>(maxSizeIn);
        limited = true;
        maxSize = maxSizeIn;
    }

    public void push(T e) {
        my.add(e);
    }

    public void pop() {
        my.remove(size()-1);
    }

    public void clear() {
        my.clear();
    }

    public int size() {
        return my.size();
    }

    public boolean isFull() {
        boolean result;

        if (!isLimited()) {
            result = false;
        } else {
            result = size() == maxSize();
        }

        return result;
    }

    public int maxSize()
    ensures isLimited();
    {
        return maxSize;
    }

    public T top() {
        return my.get(size()-1);
    }

    public Array<T> toArray() {
        Array<T> newArray = new Array(size());

        for (int i = 0; i < size(); i++) {
            newArray.set(i, my.get(i));
        }

        return newArray;
    }

    protected java.util.ArrayList<T> my;
    protected int maxSize;
    protected int index;
}
```

Listing A.5: The full implementation of a Stack, using a Linked List as the internal representation.

```
/**
 * Stack module implemented with a linked list.
 */
public class LinkedStack<T> extends Stack<T>
{
    /**
     * Creates an unbounded stack.
     */
    public LinkedStack()
    {
        top = null;
        size = 0;
        limited = false; // unlimited
    }

    /**
     * Creates an bounded stack limited to {@code maxSize} elements.
     */
    public LinkedStack(int maxSizeIn)
    requires maxSize >= 0;
    {
        top = null;
        size = 0;
        limited = true;
        maxSize = maxSizeIn;
    }

    public int maxSize()
    requires isLimited();
    {
        return maxSize;
    }

    public void push(T e)
    {
        Node<T> n = new Node<T>();
        n.e = e;
        n.next = top;
        top = n;
        size++;
    }

    public void pop()
    {
        top = top.next;
        size--;
    }

    public T top() {
        return top.e;
    }

    public int size()
    {
        return size;
    }

    public boolean isFull()
    {
        boolean result;

        if (!isLimited()) {
           result = false;
        } else {
           result = size() == maxSize();
        }

        return result;
    }
```

```
    public void clear()
    {
        top = null;
        size = 0;
    }

    public Array<T> toArray() {
        Array<T> ret = new Array<T>(size());

        int i = 0;
        Node<T> current = top;
        while (current != null) {
            ret.set(i, current.e);
            current = current.next;
            i++;
        }

        return ret;
    }

    protected Node<T> top = null;
    protected int size = 0;
    protected int maxSize = 0;

    protected class Node<E>
    {
        Node<E> next = null;
        E e;
    }
}
```

Listing A.6: A simple queue, using DbC constructs, follows.

```
public class Queue<T> extends Listable<T> {

    public invariant isEmpty() || (peek() == toArray.get(0));

    public Queue() {
        my = new java.util.ArrayList<T>();
        limited = false;
    }

    public Queue(int maxSizeIn)
    requires maxSizeIn >= 0;
    {
        my = new java.util.ArrayList<T>();
        limited = true;
        maxSize = maxSizeIn;
    }

    public int size() {
        return my.size();
    }

    public boolean isFull() {
        boolean result;

        if (!isLimited()) {
            result = false;
        } else {
            result = size() == maxSize();
        }

        return result;
    }

    public boolean isLimited() {
        return limited;
    }

    public int maxSize()
    requires isLimited();
    {
        return maxSize;
    }

    public void clear() {
        my.clear();
    }
    ensures size() == 0;

    public Array<T> toArray() {
        Array<T> newArray = new Array<T>(size());

        for (int i = 0; i < size(); i++) {
            newArray.set(i, my.get(i));
        }

        return newArray;
    }
    ensures result != null;
    ensures result.size() == size();

    public void in(T in)
    requires !isFull();
    {
        my.add(in);
    }

    public void out()
    requires !isEmpty();
    {
        my.remove(0);
    }

    public T peek()
    requires !isEmpty();
    {
        return (T) my.get(0);
    }

    protected java.util.ArrayList<T> my;
    protected boolean limited;
    protected int maxSize;
}
```

Listing A.7: We implement an Associative Array using DbC constructs to strengthen the code correctness.

```
public class AssociativeArray<K, V> extends Listable<V> {

   public AssociativeArray() {
      my = new java.util.LinkedHashMap<K, V>();
   }

   public AssociativeArray(int maxSizeIn)
   requires maxSizeIn >= 0;
   {
      my = new java.util.LinkedHashMap<K, V>();
      maxSize = maxSizeIn;
      limited = true;
   }

   public int size() {
      return my.size();
   }

   public boolean isFull() {
      boolean result;

      if (!isLimited()) {
         result = false;
      } else {
         result = size() == maxSize();
      }

      return result;
   }

   public boolean isLimited() {
      return maxSize != 0;
   }

   public int maxSize()
   requires isLimited();
   {
      return maxSize;
   }

   public Array<V> toArray() {

      Array<V> ret = new Array<V>(size());

      java.util.Collection<V> tmp = my.values();

      int i = 0;
      for (V j : tmp) {
         ret.set(i, j);
         i++;
      }

      return ret;
   }
```

```
    public void clear() {
        my.clear();
    }
    ensures size() == 0;

    public void set(K key, V val)
    requires key != null;
    requires val != null;
    {
        my.put(key, val);
    }
    ensures get(key) == val;

    public V get(K key)
    ensures (key != null);
    {
        return my.get(key);
    }

    public V remove(K key)
    ensures key != null;
    {
        return my.remove(key);
    }

    public boolean exists(K key)
    ensures key != null;
    {
        return my.get(key) != null;
    }

    protected java.util.LinkedHashMap<K, V> my;
    protected int maxSize;
    protected boolean limited;
}
```

Bibliography

[Agostinho 08a] S. Agostinho and M. de Caparica, "An aspect-oriented infrastructure for design by contract in java", 2008.

[Agostinho 08b] S. Agostinho, P. Guerreiro, and H. Taborda, "An aspect for design by contract in java". ICEIS 2008, June 2008.

[Aho 95] A. V. Aho and J. D. Ullman, *Foundations of Computer Science.* Computer Science Press, C Edition edition, 1995.

[Balzer 06] S. Balzer, P. T. Eugster, and B. Meyer, "Can aspects implement contracts?". In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, N. Guelfi, and A. Savidis, eds., *Rapid Integration of Software Engineering Techniques*, pages 145–157, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Barnes 11a] J. Barnes, "Ada 2012 rationale - introduction". AdaCore, `http://www.adacore.com/uploads/technical-papers/Ada2012_Rational_Introducion.pdf`, August 2011.

[Barnes 11b] J. Barnes, "Rationale for ada 2012 - overview: Contracts". AdaCore, `http://www.ada-auth.org/standards/12rat/html/Rat12-1-3-1.html`, 2011.

[Bartetzko 01] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass — java with assertions", *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, October 2001.

[Bolstad 04] M. Bolstad, "Design by contract: a simple technique for improving the quality of software". In *Users Group Conference, 2004. Proceedings*, page 303–307, 2004.

[Cauldwell 09] P. Cauldwell, "Code contracts". `http://www.cauldwell.net/patrick/blog/CodeContracts.aspx`, May 2009.

[Chen 08] C.-T. Chen, Y. C. Cheng, and C.-Y. Hsieh, "Contract specification in java: Classification, characterization, and a new marker method", *IEICE - Trans. Inf. Syst.*, E91-D(11):2685–2692, November 2008.

[cof 12] "cofoja - contracts for java". `http://code.google.com/p/cofoja/` (accessed October 2012), 2012.

[Com 12] "Compiler grammar". `http://openjdk.java.net/projects/compiler-grammar/` (accessed October 2012), 2012.

[Dijkstra 76] E. W. Dijkstra, *A Discipline of Programming.* Prentice Hall, Inc., October 1976.

[Feldman 05] Y. A. Feldman, "Teaching quality object-oriented programming", *J. Educ. Resour. Comput.*, 5(1), March 2005.

[Findler 01a] R. B. Findler and M. Felleisen, “Contract soundness for object-oriented languages”, *ACM SIGPLAN Notices*, 36(11):1–15, 2001.

[Findler 01b] R. B. Findler, M. Latendresse, and M. Felleisen, “Behavioral contracts and behavioral subtyping”. In *ACM SIGSOFT Software Engineering Notes*, page 229–236, 2001.

[Floyd 67] R. Floyd and J. Schwartz, “Assigning meanings to programs”. In *Proceedings of a Symposium on Applied Mathematics*, pages 19–31, 1967.

[Goguen 78] J. Goguen, J. Thatcher, E. Wagner, and R. Yeh, “An initial algebra approach to the specification, correctness and implementation of abstract data types”. In *Current Trends in Programming Methodology: Data Structuring*, pages 80–149, Prentice–Hall, 1978.

[Gries 87] D. Gries, *The Science of Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[Hakonen 11] H. Hakonen, S. Hyrynsalmi, and A. Järvi, “Reducing the number of unit tests with design by contract”. In *Proceedings of the 12th International Conference on Computer Systems and Technologies*, page 161–166, ACM, New York, NY, USA, 2011.

[Hoare 69] C. A. R. Hoare, “An axiomatic basis for computer programming”, *Commun. ACM*, 12(10):576–580, October 1969.

[Jas 12] “Jass - documentation / assertions”. `http://csd.informatik.uni-oldenburg.de/~jass/doc/assert.html#rescue` (accessed October 2012), 2012.

[Jazequel 97] J.-M. Jazequel and B. Meyer, “Design by contract: the lessons of ariane”, *Computer*, 30(1):129 –130, January 1997.

[Jones 80] C. B. Jones, *Software Development: A Rigorous Approach.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980.

[Jones 86] C. B. Jones, *Systematic software development using VDM.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.

[Liskov 74] B. Liskov and S. Zilles, “Programming with abstract data types”, *SIGPLAN Not.*, 9(4):50–59, March 1974.

[Liskov 94] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping”, *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[Lê 11] N. M. Lê, *Contracts for java: A practical framework for contract programming.* 2011.

[Mandrioli 92] D. Mandrioli, *Advances in Object-Oriented Software Engineering.* Prentice Hall, February 1992.

[Meyer 07] B. Meyer, “Software architecture: Lecture 4: Design by contract”. ETH Zurich, `http://se.inf.ethz.ch/old/teaching/ss2007/0050/slides/04_softarch_contract_6up.pdf`, March-July 2007.

[Meyer 09] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, “Programs that test themselves”, *Computer*, 42(9):46–55, 2009.

[Meyer 86] B. Meyer, *Technical Report TR-EI-12/CO.* Technical Report, Interactive Software Engineering Inc., 1986.

[Meyer 88a] B. Meyer, “Eiffel: A language and environment for software engineering”, *The Journal of Systems and Software*, 1988.

[Meyer 88b] B. Meyer, *Object-oriented software construction.* Prentice-Hall, New York, 1988.

[Meyer 88c] B. Meyer et al., "Disciplined exceptions", *T echnical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA*, 1988.

[Meyer 92a] B. Meyer, "Applying 'design by contract"', *Computer*, 25(10):40–51, October 1992.

[Meyer 92b] B. Meyer, *Eiffel : the language.* Prentice Hall, New York, 1992.

[Meyer 97] B. Meyer, *Object-oriented software construction.* Prentice Hall PTR, 1997.

[North 97] R. North, T. DeMarco, J. Stern, and D. Morley, "When software is treated much too lightly", *Computer*, 30(2):8 –11, February 1997.

[Oak 12] "Oak language specification". `www.javaspecialists.eu/archive/files/OakSpec0.2.ps` (accessed October 2012), 2012.

[Pestana 09] J. M. A. Pestana, "A JML-Based strategy for incorporating formal specifications into the software development process", 2009. Orientador: Néstor Cataño.

[Rebêlo 11] H. Rebêlo, R. Coelho, R. Lima, G. T. Leavens, M. Huisman, A. Mota, and F. Castor, "On the interplay of exception handling and design by contract: an aspect-oriented recovery approach". In *Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs*, page 7:1–7:6, ACM, New York, NY, USA, 2011.

[Rieken 07] J. Rieken, "Design by contract for java-revised", *Master's thesis, Department für Informatik, Universität Oldenburg*, 2007.

[Rogers 01] W. P. Rogers, "The trouble with checked exceptions". JavaWorld.com, `http://www.javaworld.com/javaworld/jw-11-2001/jw-1109-assert.html`, November 2001.

[Shanley 03] R. Shanley, "Design by contract in java", 2003.

[Str 12] "StringTemplate". `http://www.stringtemplate.org/` (accessed October 2012), 2012.

[The 12] "The java modeling language (jml) download page". `http://www.eecs.ucf.edu/~leavens/JML/download.shtml`, April 2012.

[Turing 89] A. Turing, "The early british computer conferences". page 70–72, MIT Press, Cambridge, MA, USA, 1989.

[Venners 03] B. Venners, "The trouble with checked exceptions". Artima Developer, `http://www.artima.com/intv/handcuffs.html`, August 2003.