Contract-Java: Design by Contract in Java with Safe Error Handling

Miguel Oliveira e Silva[1] and Pedro G. Francisco[2]

1 **University of Aveiro, IEETA, DETI**
Campus Universitário de Santiago, Aveiro, Portugal
mos@ua.pt

2 **University of Aveiro, IEETA**
Campus Universitário de Santiago, Aveiro, Portugal
goucha@ua.pt

Abstract

Design by Contract (DbC) is a programming methodology in which the meaning of program entities, such as methods and classes, is made explicit by the use of programming predicates named assertions. A false assertion is always a manifestation of an incorrect program.

This simple founding idea, when properly applied, give programmers a tool able to specify, test, debug, document programs, as well as a mechanism to construct a simple, safe and sane error handling mechanism. Nevertheless, although well adapted to object-oriented programming (and other popular techniques such as unit testing), DbC still has a very low practical acceptance and application. We believe that one of the main reasons for such is the lack of a proper support for it in many programming languages currently in use (such as Java). A complete support for DbC requires not only the ability to specify assertions; but also the necessity to distinguish different kinds of assertions, depending of what is being asserted; a proper integration in object-oriented programming; and, finally, a coherent connection with error handling mechanisms.

It is in this last requirement that existing tools that extend Java with DbC mechanisms completely fail to properly, and coherently, integrate DbC within Java programming. The dominant practices for systematically handling failures in programming languages are not DbC based, using instead a defensive programming approach, either by using normal languages mechanisms (as in programming language `C`) or by the use of typed exceptions in `try/catch` based exception mechanisms.

In this article, we will present and justify the requirements posed on programming languages for a complete support for DbC; On the context of the last presented requirement – error handling – defensive programming will be discussed and criticized; It will be showed that, unlike Eiffel's original DbC error handling, existing typed exceptions in `try/catch` based exception mechanisms are not well adapted to algorithmic abstraction provided by methods; Finally, a new DbC Java extension named *Contract-Java* will be presented and it will be showed that it is coherently integrated both with Java existing mechanisms and DbC. It will be presented an innovative *Contract-Java* extension to DbC that automatically generates debugging information for (non-rescued) contract failures, that we believe further enhances the DbC debugging capabilities.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, D.3.4 Processors

Keywords and phrases design by contract, defensive programming, exceptions, Java, contract-Java

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.111

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 111–126

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Design-by-Contract programming, or officially [21] *Design by Contract*^{TM} (DbC)[1] is a development methodology inspired both by studies on formal programming and also, by the way contracts work in the "real world", in particular, in a clear distribution of responsibilities whenever a failure occurs in a program. It aims for a substantial improvement of a program correctness and robustness. It was born in 1986 [19, 21, 18] and first implemented within the Eiffel language (1988) [20].

The concepts in which Design by Contract is based are present in the works of Turing [27], Floyd [5], Hoare [9], Dijkstra [4], Gries [8], Jones [11, 10] and also Goguen [7].

Several approaches exist to extend Java with DbC: Jass [2], Modern Jass [26], JML[2] [13], Cofoja [12], ezContract [3], and DbC4J [1]. However, all have achieved just a portion of the features required by the methodology. All but one (Jass) fail to implement fault tolerance with a disciplined exception mechanism and automatic documentation generation; furthermore, all treat the DbC approach as an optional add-on to the language (using annotations or aspects), failing to fulfill the requirements for the full implementation of DbC. A full integration of contracts as core language syntactical constructs enhances the possibility to fulfill all the requirements including a disciplined exception mechanism (in which the knowledge of contracts is essential). Although in this approach it is not possible to directly use of native Java compilers when the new syntax is involved (as happens with other approaches) it not only makes contracts non-optional language constructs (impossible to ignore), but is also allows the direct integration and use of native Java code (allowing the direct reuse of existing Java code and libraries). Table 1 summarizes the support for DbC of all these approaches.

Regarding error handling, defensive programming [14] remains the dominant approach to systematically handle internal program's failures. This dominance has also "contaminated" some DbC approaches (e.g. JML's `exceptional_behavior`) in which the launch of exceptions can also be made part of a contract specification blurring the simple DbC view of methods (that either succeed, meeting its postcondition and the object's invariant, or fail with a contract failure). If a method terminates launching a contractualized exception, did it really fail, or is it simply meeting its contract?

The way DbC handles errors is much simpler, coherent and safe. It even gives the ability for a program to unambiguous know when it is (or it is not) failing, without the necessity for an external error arbitration, or a deep knowledge of the way programmer implement their code. Current use of exceptions disallow such possibility, because there are many types of exceptions, and some of them are, sometimes, used as normal program's flux control, to the point of such usage is promoted as a good programming practice [14].

It should be clarified that we are following a very pragmatic view of DbC in the line of original Eiffel's proposal and implementation, and not aiming a more formal assurance of contracts. Also, we are not stating that in practice contracts express the full semantics of modules (e.g. a formally complete postcondition), but simply assert *some* of those semantics.

This article is structured as follows. In Section 2 we present our contributions. In Section 3 we identify an justify the requirements posed for a complete DbC language implementation. Section 4 the problems and solutions for systematic error handling are discussed. Section 5 presents Contract-Java approach. Finally, Section 6 present some concluding remarks and future evolutions of the language.

[1] Trademarked by Eiffel Software in the United States
[2] Java Modeling Language.

2 Contributions

The major contributions of this article are the following:

- The presentation and justification of the necessary requirements posed to a programming language for a complete pragmatic support for DbC;
- A critical comparison between defensive programming and DbC approaches to error handling, and, in particular, the algorithm abstraction problems posed by typed exceptions and `try/catch` based exceptions mechanisms;
- A new complete DbC extension for Java named *Contract-Java* (able to accept and compile existing Java code);
- A complete support for a disciplined exception mechanism in *Contract-Java* without negative and undesirable interferences with native's Java exception mechanism;
- An innovative, and safe, integration of Java's native exception mechanism within DbC (allowing the application of the powerful DbC disciplined exception mechanism to any Java exception);
- A DbC enhanced debugging mechanism, by automatic generation of semantic information in the presence of an assertion failure (freeing the programmer from that burden).

3 Requirements

To achieve a complete pragmatic support for DbC within a programming language, we must clearly identify the necessary requirements to be fulfilled and justify the rationale behind them. That is the goal of this section.

3.1 Different Assertions

▶ **Requirement 1** (different assertions). *Different kinds of contracts (preconditions, postconditions, invariants and others) should be represented by different assertions. These assertions assume different roles depending on their kind, carefully assigning responsibility to different parts of the program.*

Although one can attach the (total or partial) meaning of a software element by an assertion, different responsibility chains are involved depending on where the assertion resides. A clear identification of such responsibility is required for a proper software element understanding.

In general one may identify assertions (preconditions) that must be observed before a subprogram execution (method or block), and the ones that must be ensured afterwards (postconditions). The former, are the responsibility of the client of the subprogram execution (caller or the code before the block), and the latter are the subprogram's responsibility. Hence, in the presence of an error (false assertion), depending on the type of the assertion, different program parts are to be blamed (this distinction is essential not only for debugging but also for the implementation of an appropriate error handling mechanism).

Structured programming gives a special abstraction role to methods, from which a separation of method assertions and internal algorithm assertions is desirable. Hence the terms *precondition* and *postcondition* are usually applied to methods, and other internal assertions are named *assert* (or *check* in Eiffel).

From object-oriented programming also results the necessity for a new type of assertion: *invariant*. It is needed to properly attach meaning to abstract data types implementations (based on classes and objects). In particular, it asserts the conditions that are required to be

true whenever objects are in an observable state (stable time) [21]. Invariants also clearly identifies who's responsible for it (the object).

A DbC realization that does not syntactically differentiates all these different assertions may lead to a wrongly attributed responsibility chain, compromising its proper specification and rectification.

3.2 Locality of Contracts

▶ **Requirement 2** (locality). *Contracts should be defined near to the entities they specify. The meaning (specification) of a software entity should be defined near the classes they contractualize; they are integral part of the code.*

This requirement simply states that the programmer should not be mislead to assume a different contract of a software entity than the one that was really defined. Also, no doubt should ever exist on the complete contract that applies to it.

Such possibility arises when one allows that the definition of a contract to reside elsewhere in the program text other than near to the software element if contractualizes. By definition, AOP approaches to DbC suffer from this problem.

3.3 Contracts are Part of the Interface

▶ **Requirement 3** (interface). *Contracts are part of the interface*[3] *(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.*

An *Abstract Data Type* (ADT) [15, 21] defines a class of abstract objects which is completely characterized by the (public) operations available to those objects. A *class* is a (possible partial) implementation of an ADT [22]. An object-oriented program is a structured collection of ADT implementations [22]. 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 must be always true when the class's instances (objects) are in an observable state (named stable time [22]).

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

If we take a broader view of these concepts – 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, and method contracts implement the method semantics.

Contracts must, as such, be integral to the class interface, as is the name of the methods and its arguments. They define the ADT by means of a specification and, as such, contracts

[3] in terms of defining an ADT, not in terms of the Java's interface mechanism

are independent of the implementation. Furthermore, when we extend the class, using subtype polymorphism, the ADT must be kept consistent. The only way this is possible is if contracts belong to the class interface and not to its implementation.

3.4 Contracts are Inherited

▶ **Requirement 4** (inheritance). *Contracts are inherited: a descendant 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.*

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

> 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 [16].

In the context of DbC, this implies that class contracts must be inherited. As Meyer states [21] it is possible to redefine contracts on descendant classes as long as certain conditions are met. The precondition of the descendant class must be equal or weaker than that of the parent class and, in the case of invariants and postconditions, the descendant 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 descendant 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.

3.5 Documentation

▶ **Requirement 5** (documentation). *The documentation must not only be 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.*

Design by Contract in its full form allows for the "single product principle": the product is the software. All specification and documentation is in, or extracted automatically from, the software [23].

In order to be useful, the documentation of a class must present the full overview of this 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 superclasses. The flat documentation allows for a complete documentation of the class expressed semantics in one place.

3.6 DbC Exceptions

▶ **Requirement 6** (DbC exceptions). *An error handling mechanism should be provided in order to ensure that a method can only succeed, by observing all of its attached assertions, or*

Table 1 Support for DbC in some of existing Java extensions.

DbC Java	R1-different assertions	R2-locality	R3-interface	R4-inheritance	R5-documentation	R6-DbC exceptions
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

fail signaling its failure to the caller with an exception; no other outcome should be allowed. Also, a disciplined exception mechanism should be provide to support a clean termination or for fault tolerance purposes, without ever compromising the method execution semantics.

This last requirement will be discussed in the next section.

4 Systematic Approaches for Error Handling

The dominant practice for handling errors in use today, is based on *defensive programming* [14]. In this methodology, partial procedures are considered a bad idea, and should be replaced with methods that, accepting everything, protect themselves by using normal language constructs – such as conditionals, results, error variables or exceptions – to identify the error and notify the caller. Lacking an exception mechanism, programming language `C`, uses function results (given two different meanings to the result), or global variables (as `errno`). On the other hand, Java and other more recent languages, use also the exception mechanism for the same purpose. As an example, consider the following code excerpt:

```
static double sqrt(double x) {
  if (x < 0)
    throw new IllegalArgumentException();
  ...
}
```

It is assumed that all clients of `sqrt` should track `IllegalArgumentException` to ensure complete robustness. Even if a client it confident that the argument will never be negative – by simply performing the logical test *before* the call – one cannot deactivate internal method's defensive code (because it is a total procedure).

Hence, since methods are implemented as total procedures, in defensive programming no special burden is putted on a method's client *before* its execution. It is assumed that the client will take the proper measures to handle possible failures *after* the method's execution. Such practice, however, is not only seriously flawed when exceptions are not used (because it is easy to forget such post-execution verifications), but it may also be flawed when they are used. Not only it is easy in a `try/catch` instruction to ignore the exception (all that is required is an empty `catch` block), but also, in this example, it completely misses the real source of the error: *a precondition failure*. To support this argument, it is of little importance to verify that such precondition is not explicit in the code. No sane programmer

implements a real number `sqrt` method for negative arguments, only for non-negative ones. The precondition is simply implicit in the code, implemented defensively with a conditional and an `IllegalArgumentException`, but it is, nevertheless, there. Hence, the guilty part for this failure is not the `sqrt` method (as its post-execution error handling might suggests it was), but its caller. The fault *precedes* the call. Also, it should be absolutely clear that a call to `sqrt` with a negative argument should be considered a program error (and not some ambiguous execution state of the program).

DbC takes the opposite approach to this problem: methods should be specified for what they are meant to do. If such goal only makes sense for some of the possible states of its arguments (or its object's state) so be it. A partial method implementation should be the choice, expressing clearly the necessary preconditions for its use (and the postcondition for its effect and result). It is important to note, that nothing is lost in terms of detecting errors and protecting methods and classes. To that goal, in DbC, all that is required is active executable assertions. In this respect, the difference to defensive programming is that we may deactivate particular assertions if we are confident the asserted code is correct.

However, taking now a broader perspective on systematic error handling methodology, DbC gives us much more than simply a way to detect and act on errors.

First, as already stated, in DbC a program *knows* when and if it has failed: simply when a false assertion (any one) was executed. In defensive programming, no such simple criteria exists to assert the program correctness (while executing). Not even the existence of an active exception is a similar criteria because, due to defensive programming practices, exceptions are used also as a simple flux control instruction, and sometimes promoted as perfectly acceptable practices [14].

Secondly, DbC clearly and unambiguously distributes the responsibility of the fault: a precondition failure is the responsibility of the method's caller, any other false assertion is of the responsibility of the method and/or class it belongs to.

Finally, it is perfectly adapted to method's algorithmic abstraction, the meaning of a failure adapts itself automatically as we climb up the method execution stack. For example, a precondition failure is the responsibility of the method's direct caller, but if this failure propagates in the execution stack, it no longer is a precondition failure to the caller of the caller (which would not make sense, because the precondition failure was in another method).

To better understand this last point we must take a more detail view of the dominant exception mechanism in use today.

4.1 Typed Exceptions and `try/catch` Instruction

In modern programming languages, faults are handled through exception handling mechanisms. The rationale is to attach a particular exception type to each fault source, and then allow the programmer to explicitly handle them elsewhere in the program. However, methods and classes can be very powerful abstraction mechanisms. A particular type of exception might be meaningful near to where it is generated, but soon enough it loses its meaning as one travels up in the method call stack. Such problem has been identified more than 30 years ago [17], and to cope with it an "explicit propagation of exceptions" requirement was defined for an alleged ideal exception mechanism [6].

Take, as an example, a possible function to solve the quadratic equation (using the previous listed `sqrt` method).

```
/**
* ax^2 + bx + c = 0
* the 2 root returned as a 4 length array ({r1.re, r1.im, r2.re, r2.im}.
*/
static double[] quadraticEquationSolver(double a, double b, double c) {
  if (a == 0)
    throw new IllegalArgumentException();
  ... sqrt(delta) ...
}
```

For the sake of the argument, lets suppose that this method is incorrectly implemented and the programmer did not take enough care to ensure a call to `sqrt` with a non-negative argument. Obviously, a `IllegalArgumentException` will be throwed by `sqrt`. However, if automatically propagated to `quadraticEquationSolver` client, as the exception mechanism does by default, it will fool it to believe that it as passed a wrong argument ($a = 0$).

To cope with this serious problem, some authors [14] suggest that the exception should be handled in places in which its meaning is correct, and eventually a different type of exception should be launched. To put this wise advise in practice, however, not only it could be an overwhelming amount of extra work on the part of programmers (and easy to miss, creating hard to track errors), but also questions one of the main goals of the exception mechanism: to separate normal code from exceptional one. Our methods would be "contaminated" with lots of `try/catch/throw` instructions. A new defensive exception mechanism could be devised, in which the automatic propagation of exceptions in the call stack, would be always wrapped in new `MethodFailure` alike exceptions. But such a mechanism seems clumsy, inefficient, and most of all, not necessary because DbC provides a much simpler solution.

In Java, it is also suggested that the method signature should always list the exceptions it may throw (and its meaning), which would also "contaminate" our code with lots of `throws` declarations, making it a possible maintenance nightmare.

But even if we assume that such a practice was the only way to handle exceptions (which is is not), it would still break algorithmic abstraction of methods: When a method is constructed, an abstraction barrier is created between the client (caller) of the method, and its implementation. The client care only for the meaning of the method (its postcondition). The implementer job, is to use an algorithm than fulfills such meaning. However, the possible exceptions that might be launched *depends* on the algorithm chosen by the implementer. Take, for example, the case of a "days of a month" method:

```
public static int daysOfMonth(int month, int year) {
  final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
  int result = days[month-1]; // Possible ArrayIndexOutOfBoundsException
  if (month == 2 && leapYear(year))
    result++;
  return result;
}
```

Should `ArrayIndexOutOfBoundsException` exception be part of the method's signature? It might, in this particular implementation, but different algorithms can be devised in which no such exception will ever be launched:

```
public static int daysOfMonth(int month, int year) {
  int result = 0;
  switch(month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      result = 31;
      break;
    ...
  }
  return result;
}
```

Please note that, as already mentioned, the alternative usage of other exception types, such as `IllegalArgumentException`, would still be problematic because its scope (meaning) only reached the method's direct caller (hence a `try/catch` would be required to handle the exception).

This clearly shows that attaching such exceptions to the method's signature would be an overspecification, hence: a break in the algorithm abstraction of methods. Thus, any exception type linked only to an algorithm's implementation, should never be part of the method's specification.

4.2 DbC Error Handling

In a DbC approach, running a method can only result in two possible outcomes (no compromise here): either the method succeeds, observing all its attached assertions (postcondition, possible object invariants, and other internal assertions); or it fails raising a DbC exception. Furthermore, the meaning of the exception signaled by methods is not immutable (forever binded to its original fault). As the exception propagates in the method invocation stack, its abstract meaning changes, going from the original assertion failure to the failure of each method in which it is propagated. What this means is that if the programmer desires to build a fault tolerant program, he can adapt it (automatically) to the abstraction level of the redundant method, regardless of the primeval fault origin. The rationale is as simple as it is powerful: a method need only to ensure its attached assertions, hence in a DbC fault tolerant program we only need a disciplined exception mechanism [20] containing a rescue execution block, in which either the method execution is retried (possibly selecting a different execution path within the method), or it fails (with an exception propagation). The possible alternative method execution, need only to be concerned with ensuring the method postconditions (and object invariant), not with any possible internal assertion failure that might have occurred in a previous failed execution (hence, for fault tolerance goals, the specific type of the original exception loses much of its importance, as long as the objects involved were properly cleaned-up).

In a disciplined exception mechanism it is structurally very hard to ignore an exception either by distraction, laziness, or bad coding practices: A DbC exception is only recovered iff due to the interaction of the rescue code and the methods body, the method is able to fulfill its postcondition (and invariant). To the clients of such a method, the program proceeds as if nothing wrong had happened, thus achieving a simple, safe, and powerful fault tolerance mechanism.

To stress even further the importance of requirement 1, the rescue code should only be applied to faults of the responsibility of the method. Hence, if this method's precondition has failed, its eventual rescue clause should not be executed (it is impossible to ensure postconditions, in the presence of a false precondition).

Serving as an introduction to the next section, *Contract-Java*'s implementation of the quadratic equation solver methods follows:

```
static double sqrt(double x)
  requires x >= 0;
{ ... }
  ensures Math.abs(result*result - x) <= NEAR_ZERO;

static double[] quadraticEquationSolver(double a, double b, double c)
  requires a != 0;
{ ... }
  ensures areRoots(a, b, c, result);
```

Listing 1 Example of a Contract-Java array implementation.

```
public class Array<T>
{
  public invariant (isEmpty() && size() == 0) || // object's ADT
                   (!isEmpty() && size() > 0);   // invariant

  public Array(int size)
    requires size >= 0;                          // Constructor
  {                                              // precondition
    array = (T[]) new Object[size];
  }

  public int size()
  {
    return array.length;
  }
    ensures result >= 0;                         // size postcondition

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

  public void set(int idx, T elem)
    requires idx >= 0 && idx < size();           // set precondition
  {
    array[idx] = elem;
  }
    ensures get(idx) == elem;                    // set postcondition

  protected T[] array;

  protected invariant array != null;             // internal representation
}                                                // invariant
```

5 *Contract-Java*

Contract-Java language was developed to ease and to take full advantage of DbC programming within Java, while attempting to retain usual syntactical and semantic choices in the language. A special care was taken both to disallow undesirable side-effects and to promote synergic behaviors with existing mechanisms (exceptions, for instance).

Unlike most existing approaches, *Contract-Java* makes DbC constructs normal language entities and fully implements all six requirements for DbC support as specified in section 3. As such, *Contract-Java* is defined as a *superset* of the Java language aiming the support of DbC[4]. All Java code is a valid *Contract-Java* code (obviously, the reverse is not true).

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

Listing 1 shows an example of an array module's ADT in *Contract-Java*.

The syntax diagrams of the *Contract-Java* extensions to Java are showed in appendix 6.1.

5.1 Method Contracts

Contract-Java allows methods to be attached with a *precondition* and a *postcondition*. Such assertions must be defined near to the method declaration, and as close as possible as to where

[4] Minor incompatibilities may arise due to the new language keywords, although a proper compiler implementation might eliminate or reduce them to a minimum (`check`).

(when) they apply: precondition before the methods body, and postcondition afterwards (following Eiffel's approach). A contract might be applied to abstract methods (and even to an interface method declaration). All method interface contracts are inherited as described by requirement 4.

In the method's postcondition two new keyword are recognized in order to allow to express assertions using the method's result (if any), and the values of expressions when the method started its execution. Respectively: `result` and `old`.

No support exists yet for frame rules within method assertions to express what should *not change* during its execution.

5.2 Class Contracts

One or more invariant declarations might be declared in a *Contract-Java* class. Syntactically they are similar to method's preconditions and postconditions, except that its scope is within the class and its visibility can be specified (as happens with other class members).

The visibility definition of an invariant in *Contract-Java* is an interesting feature of the language because it allows the definition of different invariants applied to different abstraction levels (public, package, protected and private). Public invariants are the ADT's invariants. One the other hand, protected (or other) invariants are useful to express less abstract representation invariants[5]. These differences should be taken into consideration by *Contract-Java*'s automatic documentation tools. Listing 1 exemplifies the usage of both an ADT and a representation invariant.

When a class is a descendant of another class (or one or more interfaces) it inherits its invariants (as explained in requirement 4).

5.3 Java Interfaces

Interfaces specify an ADT without providing its implementation. As such, to completely specify the ADT, 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.

5.4 DbC Exceptions

The sixth requirement of our DbC requirements (section 3) is support for DbC exceptions.

To that goal, *Contract-Java* implements a set of DbC exceptions (descendant of `Error` type), for each type of assertion (although, as showed, such detail in not very important in DbC error handling, in which all that matter is if the method succeeds or has failed, and, if so, whose to blame for that). So, as part of the language specification, assertion errors will in fact be handled by Java's exception mechanism. However, it is ensure by the language semantics that it is impossible for a `try/catch/throw/throws` to use such exceptions[6]. These special exceptions can only be handled by the language's disciplined exception mechanism.

[5] Consistently, a private invariant will be hidden from descendant classes.
[6] Even catching `Throwable` does not catch a *Contract-Java* exception.

Listing 2 Example of real code of how to define a rescue clause on a class.

```
public class AFaultTolerantMethod
  // Does a path exist in labyrinth from src to dst?
  public boolean findPath(Labyrinth labyrinth, Location src, Locality dst)
    requires labyrinth != null; // this precondition is not rescued by
             src != null;       // this method's rescue clause.
             dst != null;
  local
    int attempt = 1;
  {
    boolean result = false;
    switch(attempt)
    {
      case 1:
        result=findPathAlg1(labyrinth,src,dst); // a method that tries
        break;                                  // to find the path
      case 2:
        result=findPathAlg2(labyrinth,src,dst); // another method that tries
        break;                                  // to find the path
    }
    return result;
  }
  rescue(RuntimeException e) // rescues both DbC exceptions and
  {                          // runtime exceptions.
    if (attempt < 2)
    {
      attempt++;
      retry;
    }
    // exception propagated to caller!
  }
```

This mechanism, whose syntax is showed in appendix 6.1, works in a similar way as Eiffel's original mechanism [21], but fully adapted to Java's mechanisms, in particular, exception handling.

Syntactically, methods were extended with an optional *rescue* clause able catch and handle contract failures within the execution of the method, and also (if desired) an optional *rescue* clause in which variables can be declared whose scope includes method's *body* and *rescue* clauses. This *local* clause allows the construction of rescue code which may depend on previous retried executions of the method.

A disciplined exception mechanism works as follows. With the important exception of the method's preconditions, all contract failures that occur during the method execution (including its postcondition, the invariant, and contract failures of called methods) are cached by the method's rescue clause. However, unlike catch blocks in usual exceptions mechanisms, rescue clauses are only allowed to retry the methods execution (command: `retry`), or re-propagate the failure to the caller method (if the rescue clause finishes without a retry command). Hence, they serve the purpose of an eventual object's cleanup, or to support a fault tolerant method.

Since some contract failures are sometimes implicit, and rely on the native exception mechanism (e.g. `NullPointerException`), *Contract-Java* extends rescue clause with an optional argument, quite similar to Java's 7 *catch* syntax, enabling the possibility to rescue non-DbC exceptions. This functionality could be extremely useful as it also eases the integration of legacy code within *Contract-Java*.

All these semantics is possible, because *Contract-Java* compiler is able to unambiguously distinguish *Contract-Java* classes and native Java's classes (both classes and object can coexist peacefully in a *Contract-Java* program).

Listing 2 exemplifies a fault tolerant method (allegedly it tries two algorithms for searching a path within a labyrinth).

The rescue clause must be associated to a non-abstract method. If the failure is of the method's responsibility (i.e., not on a precondition) then the execution jumps to the rescue clause 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.

5.5 Enhanced Debugging in *Contract-Java*

When checking for an assertion the programmer can define an appropriate error message to be used when that assertion fails. For example, the program:

```
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() : "message";
  }
}
```

would yield the result:

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

However, in *Contract-Java* the error messages associated with the assertion failures are automatically enhanced with relevant debugging information, such as the boolean expression that failed, and an expansion of the various expression values, thus reducing the need of manual definition of the assertions' associated text and providing a clearer view of why the assertion failed. If the following example program is executed:

```
public class TestBooleanExpansion {

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

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

it could be created the following output:

```
$ 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;
```

5.5.1 Fine-tuning

Contract-Java allows the possibility to fine-tune the activation and deactivation of assertions (by assertion kind, to the whole program, to packages, or class by class). However, unlike Java's native `assert`, contracts are defined at compile time.

5.6 Other Assertions

We support the equivalent to the `assert` keyword from Java, namely `check`. This instruction allows for the verification of a boolean expression triggering a failure when it is not true, of type `checkFailure`.

5.7 *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[7] being defined.

5.8 Documentation

The support for full automatic documentation generation is the fifth requirement of our DbC requirements (Section 3). 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). Such documentation would be extracted using another tool, which would generate javadoc-like documentation with the addition of contract information.

6 Conclusion

We have presented and justified the requirements needed to completely implement Design by Contract in an Object-Oriented programming language. Systematic error handling approaches, such as defensive programming, were critically analyzed and compared with DbC alternative. It was showed that typed exceptions and `try/catch` instructions break algorithmic abstraction of methods; hence becoming a questionable alternative to handle errors in a program. A better approach, based on DbC was presented and justified.

A new DbC language extension to Java – named *Contract-Java* – was presented. This new language accepts all existing Java code, and was implemented in order to avoid undesirable side-effects with Java's native mechanisms. Is was given a special care to error handling, thus, *Contract-Java* implements a disciplined exception mechanism, preventing exceptions from being ignored, and easing the development of fault-tolerant programs. To better integrate with existing Java code, this new exception mechanism is able to integrate any desired

[7] With a different ADT, since Java native classes follow defensive programming and have no regard for command-query separation.

non-DbC exceptions within the DbC error handling mechanism. Finally, a new enhanced debugging mechanism was implemented in which relevant information is automatically generated and printed in the presence of a contract failure.

6.1 Future Developments

We expect to enhance *Contract-Java* with mechanisms for pure query detection (assertions should use only expressions without side-effects to the program's state).

Some work on frame rules (predicates expressing what did not change), is also one of our objectives. Also, the implementation of other kinds of assertions like loop invariants

Finally, work in on the way for a concurrent versions of *Contract-Java* (*Concurrent Contract-Java*) adapting one of the authors PhD work [24, 25] to Java.

Acknowledgment. This work was partially supported by IEETA (Instituto de Engenharia Electrónica e Telemática de Aveiro), funded by FCT's project PEst-OE/EEI/UI0127/2014.

Contract-Java new Syntax

```
classDeclaration: modifiers "class" name ( typeParameters )?
   "{" ( ";" | staticBlock | interfaceDeclaration |
       classDeclaration | field | method | invariant )+ "}"

method: modifiers type name "(" ( arguments )? ")"
    ( precondition )?
    ( ( local )? "{" body "}" )?
    ( postcondition )?
    ( rescue )?

invariant: ("public" | "protected" | "" | "private" )
    "invariant" ( assertionClause )+

precondition: "requires" ( assertionClause )+

postcondition: "ensures" ( assertionClause )+

assertionClause: conditionalExpression
    ( ":" expression )? ";"

rescue: "rescue" ( excpDecl )? "{" blockStatement "}"

excpDecl: "(" excpTypeList name ")"

excpTypeList: excpType ( "|" excpType )*
```

References

1 Sérgio Agostinho. An aspect-oriented infrastructure for design by contract in java. Master's thesis, Universidade Nova de Lisboa, 2008.

2 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.

3 Chien-Tsun Chen, Yu Chin Cheng, and Chin-Yun Hsieh. Contract specification in java: Classification, characterization, and a new marker method. *IEICE – Trans. Inf. Syst.*, E91-D(11):2685–2692, November 2008.

4 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

5 Robert Floyd and J. T. Schwartz. Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, volume 19, pages 19–31, 1967.

6 Alessandro F. Garcia, Cecília M. F. Rubira, Alexander Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, November 2001.

7 J. A. Goguen, J. W. Thatcher, E. G. 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*, volume 4, pages 80–149. Prentice–Hall, 1978.

8 David Gries. *The Science of Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

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

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

11 Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980.

12 Nhat Minh Lê. Contracts for java: A practical framework for contract programming. Technical report, Google Switzerland GmbH, 2011.

13 Gary T. Leavens. The java modeling language (jml) online page. `http://www.eecs.ucf.edu/~leavens/JML/download.shtml`, August 2013.

14 Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

15 Barbara Liskov and Stephen Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, March 1974.

16 Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

17 B. H. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, 5(6):546–558, 1979.

18 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.

19 Bertrand Meyer. Technical report tr-ei-12/co. Technical report, Interactive Software Engineering Inc., 1986.

20 Bertrand Meyer. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software*, 1988.

21 Bertrand Meyer. *Object-oriented software construction.* Prentice-Hall, New York, 1988.

22 Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, 2nd edition, 1997.

23 Bertrand Meyer. Software architecture: Lecture 4: Design by contract. `http://se.inf.ethz.ch/old/teaching/ss2007/0050/slides/04_softarch_contract_6up.pdf`, ETHZ, March-July 2007.

24 Miguel Oliveira e Silva. Concurrent object-oriented programming: The mp-eiffel approach. *Journal of Object Technology*, 3(4):97–124, April 2004. Proceedings of the TOOLS USA 2003 Conference, September 30 to October 1, 2003 – Santa Monica, CA.

25 Miguel Oliveira e Silva. Automatic realizations of statically safe intra-object synchronization schemes in MP-Eiffel. In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 91–118. University of York – Department of Computer Science, July 2006. Available at `http://www.ieeta.pt/~mos/pubs`.

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

27 A. Turing. Checking a large routine. In Martin Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.