Designing a Programming Contract Library for Java

Neha Rajkumar
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation
DOI: https://doi.org/10.31979/etd.urbd-323n
https://scholarworks.sjsu.edu/etd_projects/426

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.
Designing a Programming Contract Library for Java

A Project
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Neha Rajkumar
December 2015
The Designated Project Committee Approves the Project Titled

Designing a Programming Contract Library for Java

by

Neha Rajkumar

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Dr. Thomas Austin  Department of Computer Science
Dr. Chris Pollett  Department of Computer Science
Mr. Ron Mak  Department of Computer Science
ABSTRACT

Designing a Programming Contract Library for Java

by Neha Rajkumar

Programmers are now developing large and complex software systems, so it’s important to have software that is consistent, efficient, and robust. Programming contracts allow developers to specify preconditions, postconditions, and invariants in order to more easily identify programming errors. The design by contract principle [1] was first used in the Eiffel programming language [2], and has since been extended to libraries in many other languages.

The purpose of my project is to design a programming contract library for Java. The library supports a set of preconditions, postconditions, and invariants that are specified in Java annotations. It incorporates contract checking for objects of classes following the bean notation [3]. The library also supports checking for user-defined functions as contract conditions. This feature allows the user to check for complex contract conditions. In addition to these, the library supports contracts using lambdas in Java 8 [4], which to our knowledge has not been done in previous works on Java contracts. While the results show us that enabling contracts lowers the performance of the system, especially when lambda contracts are used, we also demonstrate how careful design can significantly reduce the overhead.
ACKNOWLEDGMENTS

I am very thankful to my advisor Dr. Thomas Austin for his continuous guidance and support throughout this project. His patience, answers to my questions, and that he was always there helped me complete this project.

I would also like to thank the committee members Dr. Chris Pollett and Mr. Ron Mak for their valuable time taken to monitor the progress of the project.

Finally, I would like to thank all of my family and friends, who in many ways helped me in my journey of Masters.
## Contents

### Chapter

1 **Introduction** .................................................. 1
   1.1 Blame Assignment .......................................... 2

2 **History** ...................................................... 4
   2.1 “Famous” Software Failures ............................... 5
   2.2 Example: Writing Contracts for a Function .......... 6
   2.3 Specifying Annotation Conditions ......................... 9

3 **Background Motivation** .................................... 11
   3.1 Java Assertion .............................................. 11
   3.2 Existing Contracts ......................................... 14
   3.3 Contracts in other Programming Languages ........... 15
     3.3.1 Contracts in Racket ................................. 16
     3.3.2 Contracts in C++ ...................................... 16
     3.3.3 Contracts in Python ................................. 17
   3.4 Contracts for Higher-Order Functions .................... 17

4 **Implementation of the Contract Library for Java** .......... 19
   4.1 Custom Annotations ....................................... 19
   4.2 Modularizing Cross-cutting Concerns using AspectJ .... 20
   4.3 Contract Checking for Objects ........................... 25
   4.4 Contract Checking using User-defined Functions ....... 27
   4.5 Contracts using Lambdas in Java 8 ....................... 29
5 Sample Contracts and Performance Results .......................... 31
  5.1 System Configuration ........................................... 31
  5.2 Contracts in File System Access Permission ..................... 31
  5.3 Contracts in an Account Application ............................ 34
  5.4 Performance Results ............................................ 36

6 Conclusion ......................................................... 39
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Contract between a Car Rental Company and a Customer.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Writing Contracts for a Method in Eiffel Programming Language.</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Contracts for Ariane 5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Contracts for a method which calculates the sum of 1..n</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Output of the function when the input is 8</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Output of the function when the input is -5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>“before” Advice</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>“after” Advice</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>Function using Assert</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>Function using Contracts</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>Output</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>Contracts in Racket</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>Contract Annotation</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>Run time Execution using AspectJ</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>Contracts for the method : methodproduct which returns product of numbers</td>
<td>23</td>
</tr>
<tr>
<td>16</td>
<td>Output for methodproduct</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>Program Flow</td>
<td>25</td>
</tr>
<tr>
<td>18</td>
<td>Contracts for a Library Application</td>
<td>26</td>
</tr>
<tr>
<td>19</td>
<td>Contracts for Quick sort</td>
<td>28</td>
</tr>
<tr>
<td>20</td>
<td>Example for Lambda Contracts</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Contracts for File Permission</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>--------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>22</td>
<td>Output for Contracts in File Permission</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Output for contracts in File Permission where file does not exist</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Contracts for Account Application</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Output for Contracts in an Account Application</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Quick sort Recursive</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Quick sort Recursive with Wrapper class</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Performance Results</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Software is getting larger. For example in 1985, Windows 1 had 1 to 3 million lines of code whereas now in 2015, Windows 10 have more than 60 million lines of code. As new software development tools and methods are built, more importance is given to productivity.

Reliability is a major component of the quality of a software: a system’s ability to execute the software according to the given specifications and to handle abnormal situations [34]. Software must work correctly without giving erroneous outputs, it should also be able to restore to a consistent phase even when an error occurs while informing the programmer about the cause of the error [5]. A programmer must be able to specify what actions each software element is supposed to do and determine the cause of a fault in order to improve software reliability. This can be achieved by programming contracts [35]. Design by contract has applications throughout the process of building software, from analysis and design to implementation, documentation, debugging, and even project management. As mentioned before contracts are specified by preconditions, postconditions, and invariants.

A contract denotes a relationship between a client and a supplier. The contract is said to be broken if the client has not met the preconditions of the supplier or the supplier has not met the postconditions. Software is correct when all of its preconditions, postconditions, and the invariants are true. According to R. Mitchell and J. McKim [1] design by contract helps to build bug-free software leading to safe exception handling.
To be more specific:

- A precondition is a condition on a method that must be true prior to the execution of the method.

- A postcondition is a condition on a method that will become true after the successful completion of the method.

- Invariant is a condition that must hold true in all cases.

1.1 Blame Assignment

Design by contracts are based on blame assignment [24]. If the precondition is violated it is the caller’s fault; if the postcondition is violated it is the fault of the called function. So with the help of the contract library we know who to blame, whether it’s the caller or the called method. Java assertions are another technique to check for the correctness of the program as given in [15]. Using assertions along with exceptions, users are able to develop robust programs. Assertions are enabled using the “assert” keyword. More on assert will be covered in the next chapter.

The table in Figure 1 illustrates a simple example of a contract between a car rental company and a customer [6].
This example gives a rough description of the mutual agreement between the client (customer) and the supplier (car rental company). The purpose of my project is to implement a programming contract library for Java that supports a set of preconditions, postconditions, and invariants which helps in error checking.

Chapter 2 takes one through the history of design by contracts along with the examples of few famous software failures. Chapter 3 describes some background information on design by contracts and shows how contracts are better than `assert` statements. Chapter 4 describes the language definition for specifying contracts along with the implementation and how the support for Java 8 lambdas are added to the library. Chapter 5 takes one through the examples of design by contracts and Chapter 6 gives the conclusion along with future work.

Figure 1: Contract between a Car Rental Company and a Customer.

<table>
<thead>
<tr>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td><strong>(Must ensure precondition)</strong></td>
</tr>
<tr>
<td>Must have a valid driver license.</td>
<td>Give the warranty that the car is in good condition.</td>
</tr>
<tr>
<td>Customer must pay the security deposit and insurance fees.</td>
<td></td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td><strong>(Must ensure postcondition)</strong></td>
</tr>
<tr>
<td>Give warranty to customer.</td>
<td>Not obligated to give the car to a customer who does not have a valid driving license, or who has not paid the security deposit and insurance fees</td>
</tr>
</tbody>
</table>
CHAPTER 2

History

Bertrand Meyer came up with the term design by contract during his design of the Eiffel Programming Language [2]. The concepts behind design by contracts were stated in articles starting from 1986 [7] followed by later editions (1988, 1997) in the book Object Oriented Software Construction [8]. Eiffel Software got their trademark in December 2004 [10]. The current owner of this trademark is Eiffel Software.

The Eiffel programming language use the keywords requires to check for the preconditions, and ensures to check for the postconditions.

```
method_name (argument declarations) is
  require
    Precondition
  do
    method body
  ensure
    Postcondition
end
```

Figure 2: Writing Contracts for a Method in Eiffel Programming Language.

Nowadays contracts are used to document software elements. This helps the clients to get information about the interface properties of the class. The use of contracts in testing, debugging, and quality assurance is worth noting. Apart from the above mentioned cases, design by contracts are used in exception handling - handling abnormal conditions [9].
2.1 “Famous” Software Failures

Several studies have shown the emphasis of design by contract in the construction of reliable software. On June 4th, 1996 the European Ariane 5 launcher crashed about 40 seconds after takeoff [11]. Media reports state that there was a loss of a half-billion dollars as the rocket was not insured. The French Space Agency and the European Space Agency were appointed immediately for investigation. After a month they reported that the explosion was due to a software error. Particularly distressing is the fact that the explosion was caused due to an exception which was not caught.

The exception was due to a floating-point error where the flight’s horizontal bias, the horizontal velocity of the rocket which was represented by a 64-bit floating-point value was converted to a 16-bit signed integer [11]. So the value that was converted was greater than the 16-bit signed integer. This resulted in an uncaught exception followed by a software crash and mission failure. Reports state that this piece of code was directly reused from the Ariane 4 launcher.

The programmers had made an assumption that the horizontal velocity of Ariane 4 would never exceed the maximum speed limit that can cause trouble. But unfortunately, Ariane 5 was much more faster than Ariane 4 and was able to achieve five times more velocity and acceleration resulting in an overflow error. This example illustrates that the specification associated with the reused module was absent. As explained before, one of the key fundamentals of design by contract includes stating the fundamental constraints of the software elements explicitly.

Assuming a positive value for horizontal_bias, the most likely value for maximum_bias is 32767 which is the maximum value of a 16-bit signed integer. So we have the postcondition as horizontal_bias <= maximum_bias. The example in Fig-
Figure 3: Contracts for Ariane 5

Table 3 gives us an idea of how design by contracts plays an important role in checking the correctness of reusable code.

2.2 Example: Writing Contracts for a Function

To know more about contracts let us take a simple example of a function that takes a positive integer value, as input and returns the sum of 1 to that number as output. This program uses my contract library, which was implemented as part of the thesis project. Let’s assume that this function is part of a very large software system [1].

Assume: The input is “var” and output is “sum”

Invariant condition: var > 0

Precondition 1: var >= 1

Postcondition 1: sum > var

In my contract library, the conditions to be evaluated are defined in the custom annotation @Contract [12]. The invariant conditions are specified in invariant_cond, preconditions in pre_cond, and postconditions in post_cond.
As seen in Figure 4, the invariant condition is “var>0”, precondition is “var≥1”, and the postcondition is “sum>var”. When the method methodsum is called, the contracts are checked at run time and shows whom to blame, whether the caller; or the called method. The library shows whom is to blame if either of these conditions fail. The figure below shows the output of the program after the execution of the program.

**Testcase 1 : The input is 8**

As seen in Figure 5, the library checks for the contract conditions for the method methodsum. It checks for the invariants first, followed by the preconditions, the execution of the method, the postconditions and the invariants again to confirm that even after the execution of the method the invariants hold true, since invariants
Checking for Contracts for the method: methodsum

Checking for Invariants
The Invariant conditions are: [var$>$0]
The Total num of Invariant conditions are: 1
The Invariant condition: 1 is correct

Checking for Preconditions
The Pre conditions are: [var$\geq$1]
The Total num of Pre conditions are: 1

Executing the function:
The Precondition: 1 is correct
The results are:
Sum of 1 to ..n is 36

Checking for Postconditions
After executing the function:
The Postconditions are: [sum$>$var]
The Total num of Postconditions are: 1
The Postcondition: 1 is correct

Checking for Invariant conditions
After executing the function:
The Invariant conditions are: [var$>$0]
The Total num of Invariant conditions are: 1
The Invariant condition: 1 is correct

Figure 5: Output of the function when the input is 8

must be true at all times. In this example the invariants, preconditions and the postconditions are true and there is no Contract Violation.

Testcase 2: The input is -5
In this example the invariant condition fails as the input is not greater than 0. Hence the program exits with the “Invariant violation” error. The example above gives us the first taste of design by contract.

2.3 Specifying Annotation Conditions

The main challenge involved in implementing the library was to check for the preconditions and the postconditions. The preconditions, postconditions, and the invariants are functionalities that are mixed with the application code. AspectJ [13] is used to separate these functionalities and to check the conditions at run time. AspectJ modularizes “cross-cutting” [14] concerns, where code is scattered across many files; logging is the canonical example of such a cross-cutting concern. The dynamic parts of AspectJ include join points, pointcuts, and advice. The contracts are specified through annotations and the conditions are checked at run time through the “before” and the “after” advice in AspectJ. The figure below shows the “before” and the “after” advice.
As seen in Figure 8, the Object \texttt{objret} gives the return value after the execution of the method which helps in checking for the conditions. More on the implementation using AspectJ will be covered in chapter 4.
CHAPTER 3

Background Motivation

This chapter explores existing approaches for guaranteeing the correctness of a program, followed by the approach used in my project.

3.1 Java Assertion

Java assertion is another technique to check for the correctness of the program as given in [15]. Using assertions along with exceptions enables the users to develop robust programs. Assertions are enabled using the `assert` keyword. The `assert` keyword can be used in two different ways as given below:

- `assert booleanExp;`
- `assert booleanExp : errorMessage;`

In both the cases the booleanExpression is checked at run time. If it evaluates to `false`, Java throws an `AssertionError` and the program terminates. Consider a function that takes a positive parameter as input and returns the square root of that number, shown in Figure 9.

The `assert` keyword checks whether the return value is greater than zero. In this case, a negative value was given as input giving an “Assertion error”, since `findsqraroot` returns `NaN` in this case.
```java
// The class to compute the square root of a number using assert
public class Report_squareroot {

    /* The main function
     * function name : main
     * input : args string
     */

    public static void main(String[] args) {
        Report_squareroot obj = new Report_squareroot();

        // The input is -999
        int var = -999;
        int result = obj.findsquareroot(var);

        assert(result>=0) : "Result is invalid" + result;
    }

    // close of main function

    /* The function to calculate the square root of the given input number
     * function name : findsquareroot
     * input : var of type int
     */

    public static int findsquareroot(int var) {
        assert(var>=0) : "Input is negative and cannot find square root";

        double result;
        result = Math.sqrt(var);
        System.out.println("Square root is :" + result);
        return (int) result;
    }
}
```

Figure 9: Function using Assert
public class Report_squareroot { 
    
    /* The main function 
    * function name : main 
    * input : args string 
    */
    
    public static void main(String[] args) {
        Report_squareroot obj = new Report_squareroot();
        // The input is -999
        int var = -999;
        int result = obj.findsquareroot(var);
    }
}

@Contract(invariant_cond="",
    pre_cond="\[var>0\]",
    post_cond="\[result>0\],
    Description="Check Contracts")

public static int findsquareroot(int var)
{
    double result;
    result = Math.sqrt(var);
    System.out.println("Square root is : "+result);
    return (int) result;
}

Figure 10: Function using Contracts

Checking for Contracts for the method : findsquareroot
Checking for Invariants
The Invariant conditions are : []

Checking for Preconditions
The Pre conditions are : \[var>0\]
The Total num of Pre conditions are:1

Executing the function :
CONTRACT VIOLATION
PRECONDITION : 1 IS WRONG

Figure 11: Output
Figure 10 shows the same function using my custom annotations to capture the same requirements/guarantees as the Java assertions. Figure 11 shows the result of executing this program.

The two examples illustrates that assert statements do not explicitly specify whether it’s an invariant, precondition, or a postcondition. Asserts do not provide contract checking. As seen in the example for method findsquareroot in Figure 9, asserts are included in the actual code, which can cause a serious issue when a programmer accidentally removes it without knowing the design requirements. Such errors can be avoided by contracts since they are written outside of the particular method that needs to be tested. Enabling contracts as in Figure 10 helps the programmers in the design phase as each module is specified. Asserts do not explicitly map contract requirements to parameters. So here the developers is expected to have to manually maintain the JavaDoc comments within the code and make it clear under what cases the assert will fail.

3.2 Existing Contracts

There exists several design by contract libraries for Java programming language. One such library is iContract [16]. In iContract the contracts are added as JavaDoc comments. So the library uses a preprocessor to generate the Java code with contracts. The library converts the comments into assertion check codes. Since the contracts are written as comments, the entire processing can be done using Java, but these do not allow switching the contract checking dynamically. The library implemented as part of my thesis project is quite different from the approach given in iContract.

Another library is jContractor [17], which is a pure Java implementation of design by contract. The contracts are written as methods that follow a naming convention
and provides run time contract checking by adapting the bytecode of classes where contracts are enabled. The library uses the Java reflection API [18] for dynamic checking. We have a similar approach as mentioned in jContractor, as the contract library supports runtime checking. Cofoja [19], which is short for Contracts for Java, is another library which was developed by an intern working at Google. Cofoja is a programming framework for Java which uses bytecode and annotation processing for run-time checking.

The Cofoja model specifies the contracts based on the Eiffel model and contracts are written as strings in annotations. The contracts for Java are annotated with its own contracts, which can be compiled, tested, and bundled into the resulting JAR file so it checks its own contracts when compiling [19]. An additional @ThrowEnsures annotation is included that handles the exceptions thrown from the method where contracts are enabled if the postcondition refers to a result that does not exist. My library uses a similar approach since the contracts are written as strings in the form of custom annotations which are checked at runtime.

Adbc [20] is another library that supports design by contract for the AspectJ programming language [13]. Here the contracts are written as JavaScript expressions within the annotations. The contracts are checked at run-time with AspectJ. When a contract is broken, an exception is thrown that shows whether to blame the supplier or the client. A similar approach is used in my library where the dynamic parts of AspectJ are used for contract checking.

3.3 Contracts in other Programming Languages

Design by contract is a major research topic and they are extensively used in other programming languages as well.
3.3.1 Contracts in Racket

The higher order contracts were first introduced in the Racket’s contract system [23]. Here contracts are enforced at module boundaries. Programmers specify the behavior of a module through the `provide` clause as in `provide (contract-out ....)` and the constraints are enforced by the `require` clause as in `(require racket/-contract)`. The figure in 12 gives an example of contracts in Racket. The example

```
#lang racket
(provide (contract-out [amount positive?] ))
(define amount ...)
```

Figure 12: Contracts in Racket

in Figure 12 promises the client that the value of `amount` will always be positive.

3.3.2 Contracts in C++

There exists several design by contract libraries for C++ programming language. One such library was developed by P. Guerreiro [21] where assertion conditions were expressed as comments. The preprocessor converts the comments into executable code and the compiler checks the syntax. All the assertion functions are placed in the `Assertions class` and are inherited by classes where its functions perform assert checking. The mechanism allows the user to comment out the assertions as and when required.
3.3.3 Contracts in Python

The principles of design by contract is integrated into Python [22] similar to the approach given in iContract [16]. The assertion statements are added through comments and the dynamic type checking for method parameters and instance variables are added by using the contract principles.

3.4 Contracts for Higher-Order Functions

Contracts for higher-order functions have a strong practical potential as they allow the programmers to write complex conditions. Findler and Felleisen [25] incorporated the support for higher-order function contracts using $\lambda^CON$, which is a typed lambda calculus that supports higher-order functions. The higher-order contract checker must be able to track down from the point the contract is established to the point of contract violation. The code shows an example of contracts for higher-order functions.

CheckEven : (integer -> integer) -> boolean

(define/contract save
  ((bigger-than-zero? -> bigger-than-zero? ) -> isEven?)
  (\( f \) (... )))

The function called CheckEven denotes the type specifications. Like Racket, the define/contract is used to define contracts on methods. Here the contract

((bigger-than-zero?  -> bigger-than-zero?  )  -> isEven?)

describes the functions that accept other functions as input to check for contracts.

The code shows another example of contracts for higher-order functions.
• **function1** is a procedure that takes an integer and returns a function.

• **GeneratePrime** is a function that takes an integer and returns the prime number.

Contract : (function1 -> GeneratePrime)

function1 : takes input of type integer?

GeneratePrime : (-> integer? integers_prime?)

(define/contract :

positiveinteger? (-> positiveinteger? integers_prime?)

....

The example shows a contract applied to a curried function. The procedure function1 takes an input of type integer and returns another function GeneratePrime that accepts the second argument and returns a prime number. So if another software component calls function1, and if it returns a value instead of a function, then function1 is to blame; if the function GeneratePrime returns a number which is not prime, then the function GeneratePrime is to blame.
CHAPTER 4

Implementation of the Contract Library for Java

This reviews the implementation details of the contract library which was developed as part of my thesis along with the support of Java 8 lambdas.

4.1 Custom Annotations

Contracts are written as Java code within strings in custom annotations [36]. Annotations are metadata that can be incorporated into the code. This feature was added to Java in version 5 [26]. Annotations can be processed in two ways; at compile time by pre—compiler tools or at run time by Java reflection [27].

```java
package annotations;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(value = ElementType.METHOD)
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Contract {
    String Description();
    String[] invariant_cond() default " ";
    String[] pre_cond() default " ";
    String[] post_cond() default " ";
}
```

Figure 13: Contract Annotation

As shown in Figure 13 in line #10, the “@” in front of interface denotes that it is an annotation. In the library, the invariant conditions are specified in the String
array, invariant_cond, preconditions in pre_cond, postconditions in post_cond, and Description specifies the comments(line #12,#13,#14,#15). The conditions have a default value a String, since there can be situations where the user does not specify the conditions. In such cases, a default value of "", denotes an empty condition. Multiple conditions are specified with comma separated strings. For example, pre_cond=\{"low>-1","high>low"\}, means that the two preconditions should be satisfied for a valid precondition check.

Directives in annotation definition:

- @Retention(value = RetentionPolicy.RUNTIME) means that the annotations can be accessed by reflection during run time.

- @Target(value = ElementType.METHOD) means that annotations can be used on classes and interfaces.

4.2 Modularizing Cross-cutting Concerns using AspectJ

Aspect-oriented programming is used for contract enforcement since contracts are scattered all over the code. The annotation conditions in @Contract are checked at run time through the dynamic parts of AspectJ. A join point defines a point in the program flow. A pointcut selects certain join points and values at these points. Advice is a block of code that is executed when it reaches a join point. In this library we use the before and the after advice during the execution of preconditions and postconditions.
import java.lang.reflect.Method;
import java.lang.annotation.*;
import annotations.Contract;

public aspect asp
{
    pointcut function() : execution(* *(..));

    before() : function()
    {
        Signature sig = thisJoinPoint.getSignature();
        Method method = ((MethodSignature)sig).getMethod();

        //Annotations for @Contract
        Annotation[] annost = method.getDeclaredAnnotations();

        Contract annos = (Contract) annotation;

        String[] invariant_cond = annos.invariant_cond();
        String[] pre_cond = annos.pre_cond();
    }

    after() returning(Object objret): function()
    {
        Signature sig = thisJoinPoint.getSignature();
        Method method = ((MethodSignature)sig).getMethod();

        //Annotations for @Contract
        Annotation[] annost = method.getDeclaredAnnotations();

        Contract annospost = (Contract) annotation;
        String[] post_cond = annospost.post_cond();
        String[] invariant_cond = annospost.invariant_cond();
    }
}

Figure 14: Run time Execution using AspectJ

In Figure 14, line #5 shows the aspect asp which encapsulates the pointcut function, the before advice, and the after advice. Line #7 denotes the pointcut
function that uses wildcards; which means that the pointcut picks the point during the execution of any method regardless of its parameter types, name, or return type. The before advice (line #9 to #20) will be executed before all method executions that are matched by the function() join point, and the after advice (line #24 to #35) will be called after the method execution.

In line #11, Signature sig = thisJoinPoint.getSignature() returns the signature object where the join point function() is matched. Method method = ((MethodSignature)sig).getMethod() in line 12, gives the method object from the join point. The an nost of type Annotation[] mentioned in line #15, contains all the annotations that are present in the method. If no annotations are present, an array of length null is returned.

Lines #19,#20 specifies that the string array invariant_cond will have the invariant conditions. So invariant_cond[0] will have the first condition mentioned in the contract annotation. Similarly the string array pre_cond will have the preconditions and post_cond will have the postconditions as specified in line #32 in Figure 14. The example in Figure 17 gives us an outline of the program flow during execution of the program specified in Figure 15. Here the block of code in before advice is executed first followed by the after advice. The output of the program is specified in Figure 16.

In before advice, the invariant_cond is executed first followed by the execution of pre_cond. If either of these conditions fail, the program exits with the ContractViolation message. The method is executed and the after advice is called. Here the post_cond and the invariant_cond are executed.
public class Program_Product
{
    @Contract(invariant_cond="var>-1"),
    pre_cond="var>1","var<100"),
    post_cond="product_val>1",
    Description="Check Contracts for method product")
    /**
     * Method to compute the product of 1 to n numbers
     * @param var : var is the input to the function
     * @return product : returns the product
     */
    public static int method_product(int var)
    {
        int product_val=1,i;
        for(i=1;i<=var;i++)
        {
            product_val = product_val * i;
        }
        System.out.println("Product of 1 to ..n is "+product_val);
        return product_val;
    }
}

Figure 15: Contracts for the method : method_product which returns product of numbers
Checking for Contracts for the method : methodproduct

Checking for Invariants
The Invariant conditions are : [var>-1]
The Total num of Invariant conditions are:1
The Invariant condition:1 is correct

Checking for Preconditions
The Pre conditions are : [var>1, var<100]
The Total num of Pre conditions are:2

Executing the function :
The Precondition:1 is correct
The Precondition:2 is correct
Product of 1 to .n is 2

Checking for Postconditions
After executing the function :
The Postconditions are : [product_val>1]
The Total num of Postconditions are:1
The Postcondition: 1 is correct

Checking for Invariant conditions
After executing the function :
The Invariant conditions are : [var>-1]
The Total num of Invariant conditions are:1
The Invariant condition: 1 is correct
4.3 Contract Checking for Objects

Let us look at an example of contract checking for objects. The example in Figure 18 shows a library application. The pre-cond is "libraryobj.membership == 'Student membership'". The libraryobj is the object of the class LibraryApplication. When an object is given as a condition, the corresponding method is called using Java reflection and the condition is checked at run time. In Java reflection class.getDeclaredMethods() returns all the methods declared for the particular class using reflection API.
```java
public class LibraryApplication {
    public static int age=14;
    public static String getMembership() {
        if (age <= 14) {
            return "Student membership";
        } else if (age > 14 && age <= 50) {
            return "Adult membership";
        }
        return "Senior membership";
    }

    @Contract(invariant_cond={"age">=6"},
        pre_cond={"libraryobj.membership="
            'Student membership'"},
        post_cond={"age">10"},
        Description="Check Contracts")

    public static void process_library(LibraryApplication libraryobj, int age) {
        System.out.println("Processing applicants");
    }

    public static void main(String[] args) {
        LibraryApplication libraryobj = new LibraryApplication();
        process_library(libraryobj, age);
    }
}
```

Figure 18: Contracts for a Library Application
4.4 Contract Checking using User-defined Functions

Writing complicated conditions helps the user in error checking during the initial stages. While I was working on the contract library, I came across several examples which had complex conditions. One such example was the quick sort function. Quick sort is a sorting algorithm that follows a divide and conquer approach. If \( \text{low} \) is the lower bound and \( \text{high} \) is the upper bound of an array to be sorted, the conditions are as follows as shown in Figure 19.

- \( \text{invariant\_cond} = \"\text{low} \geq 0\" \)
- \( \text{pre\_cond} = \"\text{low} > -1\", \text{high} > \text{low}\"
- \( \text{Array.isSorted} == \text{true}, \) is a complex condition to handle

To handle complex conditions like checking whether an array is sorted, I took the approach where user could specify functions as a contract condition. So user-defined functions were checked as contract conditions allowing the user to handle complex error conditions. The functions are defined as \texttt{FUNCTIONCHECK:}. So if the condition is of the form,

\[
\text{post\_cond} = \text{"FUNCTIONCHECK:obj.isSort"}
\]

then the function \texttt{isSort()} is called with the class object \texttt{obj} and executed using the Java reflection API. The function checked with the \texttt{FUNCTIONCHECK:} should always have the same number of arguments as the method which has contracts enabled.

Consider the quick sort example in Figure 19. Lines #6 to #9 shows the contract conditions. Lines #11 to #33 gives the quick sort code. The postcondition for the method is ‘’\text{FUNCTIONCHECK:obj.isSort}’’. So here the function \texttt{isSort()} is executed using the Java reflection API, where \texttt{obj} is the object of the main class.
public static void sort(int[] arr)
{
    quickSort(0, arr.length - 1, arr, obj);
}

@Contract(invariant_cond={"low>=0"},
          pre_cond={"low>-1","high>low"},
          post_cond={"FUNCTIONCHECK:obj.isSort"},
          Description="Check Contracts")

public static void quickSort(int low, int high, int arr[], Sort_Quicksort obj)
{
    int i = low, j = high;
    int tempval;
    int pivot = arr[(low + high) / 2];
    while (i <= j)
    {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j){
            tempval = arr[i];
            arr[i] = arr[j];
            arr[j] = tempval;
            i++;
            j--;}
        if (low < j)
            quickSort(low, j, arr, obj);
        if (i < high)
            quickSort(i, high, arr, obj);
    }
} //FUNCTIONCHECK(Description="Include functions in conditions")

public static boolean isSort(int low, int high, int arr[], Sort_Quicksort obj)
{
    for (int i=1;i<arr.length;i++)
    {
        if (!arr[i-1] <= arr[i])
        {
            return false;
        }
    }
    return true;
}

Figure 19: Contracts for Quick sort
4.5 Contracts using Lambdas in Java 8

Lambdas are a new feature added in Java SE 8 [29]. Lambdas in Java 8 provide an easier way to convert to functional interfaces. Using Java 8 lambdas to write contracts has not been done in previous work. Since the lambda contracts are given as string expressions, my first concern was to convert a lambda expression to a lambda object. There were two possible ways:

- Using the Nashorn JavaScript engine. Java 8 supports the Nashorn engine where a lambda expression can be evaluated with the JavaScript function [32].

- Use a library that converts a String expression to a lambda expression and execute it on the fly.

I used the second approach, as it allows a user to stay within the Java world. I used Pawel Chorazyk’s library [33] to add support for java 8 lambdas. The library LambdaFromString written by Pawel did not work for complicated operations on objects.

The library compiles the new class using the Java Compiler API and compiles the lambda expression on the fly. Currently the library supports only the standard library functions. Let us see a simple example using lambda contracts. Lines #1 to #5 in Figure 20, denotes the lambda contracts. The invariant condition is "input_val -> input_val >= 0" and the postcondition is "sum -> sum % 2 == 0". The method print_even finds the sum of all even numbers from 1 to input_val. As seen in Figure 20, the return value sum, should be an even number. The postcondition checks whether sum % 2 == 0.
@ContractLambda(invariant_cond_lambda=
    {"input_val -> input_val >= 0"},
    pre_cond_lambda={" "},
    post_cond_lambda="sum -> sum%2 == 0"),
    Description="Check Lambda Expression Contracts")

    /*
    * The method print_even calculates the sum of even numbers
    * from 2 to input_val
    * input : input_val is the user input
    * output : sum, which is sum of even numbers till input_val
    */
    public static int print_even(int input_val)
    {
        int i,sum=0;
        for(i=2;i<=input_val;i++)
        {
            if(i%2==0)
            {
                sum = sum + i;
            }
        }
        System.out.println("The results are: ");
        System.out.println("Sum of even numbers to ..n is " +sum);
        return sum;
    }

Figure 20: Example for Lambda Contracts
CHAPTER 5

Sample Contracts and Performance Results

This chapter reviews the sample contracts along with the performance results.

5.1 System Configuration

All my sample contracts were run on a Windows 7 Enterprise system. The system is running with 2.67GHz Intel Core i7 CPU with 2 cores and 4 GB of memory.

5.2 Contracts in File System Access Permission

File permissions are important as they keep the data secure and prevents unauthorized read/write operations. Let us take an example of applying design by contracts to a file permission system. The file given in the file path should exist and only the authorized user should have access to the file. The conditions for the method Access_level_operation can be given as follows:

- invariant_cond = "FUNCTIONCHECK:obj.isFileExists"
- pre_cond = "access_obj.accesslevel=='W/write'"
- post_cond = "count>0"

The invariant condition states that the file should exist for the user to access it. The precondition checks for the accesslevel of the user, the postcondition performs the file operation, and increments a counter by 1.
@Contract(invariant_cond="FUNCTIONCHECK:access_obj.isFileExists"),
    pre_cond="access_obj.accesslevel=='W/write'",
    post_cond="count>0",
    Description="Check Contracts")

    /*
     * The function : Access_level_operation checks for the
     * access level
     * Input : Object access_obj, the object of class
     * Contracts_grantingAccess and the file path
     * return : returns a count if file operation is success
     */

    public static int
    Access_level_operation(Contracts_grantingAccess
        access_obj,String filepath)
    {
        System.out.println("File testfileinput.txt has been
        modified");
        count = 1;
        return count;
    }

    /*
     * The function : isFileExists checks for the invariant
     * Input : String filepath, the file path
     * return : returns a boolean
     */

    @FUNCTIONCHECK(Description="Include functions in
    conditions")
    public static boolean isFileExists(Contracts_grantingAccess
        access_obj,String filepath)
    {
        File access_file = new File(filepath);

        if(access_file.exists() == false)
            {return false;
            }
        return true;
    }

Figure 21: Contracts for File Permission
Lines #1 to #4, shows the contract conditions. The `access_obj.accesslevel`, returns the permission access for the particular user. The method `Access_level_operation` is defined in lines #12 to #17. The function `isFileExists` is checked as invariant condition to check whether the file exists before performing a write operation.

```
Checking for Contracts for the method : Access_level_operation
Checking for Invariants
The Invariant conditions are : [FUNCTIONCHECK:obj.isFileExists]
The Total num of Invariant conditions are:1
The Invariant condition : 1 is correct

Checking for Preconditions
The Pre conditions are : [access_obj.accesslevel==’W/write’]
The Total num of Pre conditions are:1

Executing the function :
The Precondition : 1 is correct
File testfileinput.txt has been modified

Checking for Postconditions
After executing the function :
The Postconditions are : [count>0]
The Total num of Postconditions are:1
The Postcondition: 1 is correct

Checking for Invariant conditions
After executing the function :
The Invariant conditions are : [FUNCTIONCHECK:obj.isFileExists]
The Total num of Invariant conditions are:1
The Invariant condition : 1 is correct
```

Figure 22: Output for Contracts in File Permission

```
Checking for Contracts for the method : Access_level_operation
Checking for Invariants
The Invariant conditions are : [FUNCTIONCHECK:obj.isFileExists]
The Total num of Invariant conditions are:1
INARIANT VIOLATION
INARIANT CONDITION : 1 IS WRONG
```

Figure 23: Output for contracts in File Permission where file does not exist
As shown in Figure 22, the contract conditions are correct and the counter is updated which indicates that the file operation is a success. Figure 23, shows that the invariant condition is wrong and the file does not exist. So from this example we know whom to blame making error checking easier.

5.3 Contracts in an Account Application

Let us take an example of applying design by contracts to an account application using Java 8 lambdas. The method `create_user` as given in Figure 24 creates a new account for the user. The user should have a user name that starts with an uppercase letter followed by lower case letters and a maximum length of 12. The conditions for the method `create_user` can be given as follows:

- `invariant_cond_lambda = "username -> username != null"
- `pre_cond_lambda = "username -> Character.toUpperCase(username.charAt(0)) == true","username -> username.length() <= 12"
- `post_cond_lambda = "minbalance -> minbalance <= 100"

The invariant condition states that the username given by the user should not be null. The precondition checks whether the username starts with an uppercase letter followed by lower case letters and the maximum length of the username is 12, and postcondition checks whether the minimum balance is updated.
Figure 24: Contracts for Account Application

Lines # 1 to # 5, shows the contract conditions. The method create_user is defined in lines # 16 to # 19. The function create_user is executed when the conditions are true and the minimum balance is updated.

Figure 25: Output for Contracts in an Account Application
As shown in Figure 25, the contract condition is wrong. Here we see that that one of the preconditions are wrong. Precondition 1 is wrong and it can be concluded that the given user input did not follow the specifications.

5.4 Performance Results

To understand the trade-offs and performance in incorporating contracts, sample contracts were run and the time taken for execution was noted down. In chapter 4 Figure 19, we saw the quick sort example where the contract conditions were enabled on the recursive function. Let us modify the quick sort example such that contracts are applied on the wrapper function instead of the recursive function as shown in Figure 27.

```java
public class Sort_Quicksort {
    public static void sort(int[] arr) {
        quickSort(0, arr.length - 1, arr, obj);
    }

    @Contract(invariant_cond="low>=0",
              pre_cond="low<=arr.length","high>low"),
    post_cond="FUNCTIONCHECK:obj.isSort")
    public static void quickSort(int low, int high, int arr[], Sort_Quicksort obj) {
        ......
    }
}
```

Figure 26: Quick sort Recursive

```java
public class Wrapper_Quicksort {

    public static void sort(int low, int high, int arr[], Wrapper_Quicksort obj) {
        return quickSort(0, arr.length - 1, arr, obj);
    }

    private static int[] quickSort(int low, int high, int arr[], Wrapper_Quicksort obj) {
        ......
        return arr;
    }
}
```

Figure 27: Quick sort Recursive with Wrapper class

Table 28 shows the time taken for execution when contracts are enabled and when they are disabled.
<table>
<thead>
<tr>
<th>Test case</th>
<th>Runtime (in nanoseconds)</th>
<th>Overhead of adding contracts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contracts enabled</td>
<td>Contracts disabled</td>
</tr>
<tr>
<td>Contracts for file permission</td>
<td>25,441,570</td>
<td>3,069,541.60</td>
</tr>
<tr>
<td>Contracts for account application (using lambda contracts)</td>
<td>945,511,326.80</td>
<td>3,499,843.8</td>
</tr>
<tr>
<td>Quick sort on array of 50 numbers (using recursion)</td>
<td>5,572,116,637</td>
<td>3,180,479,513</td>
</tr>
<tr>
<td>Quick sort on array of 50 numbers (using a wrapper function)</td>
<td>4,237,491,114</td>
<td>3,259,894,164</td>
</tr>
<tr>
<td>Contracts for library application</td>
<td>27,956,183.2</td>
<td>2,908,351.4</td>
</tr>
</tbody>
</table>

Figure 28: Performance Results

From the table 28, it is seen that contracts for account application had a higher slowdown factor of 270. This is because the lambda contracts are compiled on the fly using the Java compiler API before evaluating. The quick sort example using recursion had a slowdown factor of 1.8 and the quick sort example with the wrapper function had a factor of 1.2. From the quick sort example it can be seen that contracts should be applied in an efficient way as given in Figure 27 to reduce the overhead.

The contracts for file permission in Figure 18 and the library application in Figure
21 had slow down factors 8.3 and 9.6 respectively. These had user-defined functions of the form \texttt{FUNCTIONCHECK}: as a contract condition. As mentioned in chapter 4, the contract conditions with \texttt{FUNCTIONCHECK}: are executed using the Java reflection API. So from Figure 28, we can see an increase in overhead for the file permission and the library application examples. Performance results states that enabling contracts takes more time in execution, but it guarantees error free code.
CHAPTER 6

Conclusion

Design by contracts can be implemented by different approaches, but it is still an ongoing field of research. Enabling contracts makes it easier for a user to fix errors in the initial stages. Adding contracts for Java 8 lambdas to the existing research on design by contracts helps the user to provide complicated conditions. From the performance results, it is seen that contracts lowers the performance of the system, but the user can be sure that his software works under all conditions. The contract library designed as part of the thesis does not require the user to learn other programming languages as the syntax for conditions are closely related to Java.

The topics in future work include extending the library to support contract checking for objects using Java 8 lambdas. This will help programmers to write complicated conditions by using the features of Java 8.
LIST OF REFERENCES


