CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

master's thesis[1]

Design By Contract for Java - Revised

Johannes Rieken

April 24th, 2007

— *Correct System Design Group* —

Responsible Supervisor:	Prof. Dr. Ernst-Rüdiger Olderog
Second Supervisor:	Dipl.-Inform. André Platzer
Advisor:	Dipl.-Inform. Michael Möller

[1] In German: *Diplomarbeit*, Studiengang Diplom Informatik

Abstract

The software development method *Design by Contract* (DBC) bases on the idea of having contracts between two software modules. A client-module guarantees to fulfil certain conditions before calling a method from a supplier-module which in return guarantees certain properties to be true after the method call returns [Mey92]. Some programming languages like Eiffel [ECM06] or D [Wik07] support design by contract out of the box, while the Java programming language [GJSB05] has minimal support for design by contract only.

This thesis will present the design and implementation of a DBC-tool for Java, that utilises Java 5 Annotations, the Annotation Processing Infrastructure, the Java Compiler API, and the Instrumentation API [Sun04b, Dar06, Art06, Sune]. In contrast to existent DBC-tools, this implementation is distinguished by a seamless integration, a rich feature set, and by being easy to maintain. To provide a basis for the tool implementation, an analysis of existent DBC-tools for Java precedes. Its objective is to outline different techniques that have been used, and to have an overview of common features. Aside from the tool implementation, a set of Java 5 annotations, that can be used to specify the behaviour of Java programs, is defined and deployed separately.

To prove the achievements of this thesis being valuable, different case studies have been carried out. Despite the successful realisation of these case studies, some limitations exist. The origin and possibly workarounds for this limitations, as well as future work, are going to be addressed.

Acknowledgments

Many people have helped and accompanied me during this thesis and some have played an outstanding role in more than one way. First of all, I want to thank my supervisor Prof. Dr. Ernst-Rüdiger Olderog who let my freely chose the subject of this thesis. Furthermore, I would like to thank my advisor Michael Möller for encouraging and supporting me over the last six months. He thought me everything I now about behavioural specification languages and advanced concepts of design by contract. I also thank André Platzer for being my second supervisor and backup-advisor.

For a fruitful collaboration of our projects, I thank Prof. Dr. Gary T. Leavens and Kristina Boysen. I hope that both projects keep on collaborating, so that the world of Java programmers is going to enjoy first-class specification annotations.

My sincere gratitude goes to Daniel Schreck, Andreas Schäfer, and Michael Möller for proof-reading and helpful suggestions. Writing 90% of a thesis is only half the work that is to be done, thanks for helping me finishing the second half.

Further, I want to thank all my friends who accompanied me during the last five years. Writing this thesis is only the final part of my studies, and without their support I would have never come so far. Doing computer science is team sport, so thanks for collaborating on assignments and in project groups, for constant encouragement, soccer & beer, cycling, running, for blockbuster- and poker-nights.

Last, but not least, I want to thank my parents and my family for their mental and financial support during my studies.

Contents

List of Figures

Listings

Chapter 1

Introduction

Over the last decades complexity of software systems increased drastically and there are numerous examples of software projects which failed due to this complexity. Consequently, methods to deal with complexity of software systems have been developed. They help to understand and prove, what a large software system is doing, by specifying its behaviour. *Design by Contract* (DBC) is such a methodology as it treats two software components as client and supplier which have a contract specifying how they interact. Originally design by contract was part of the Eiffel programming language [ECM06], but nowadays other programming language implement design by contract, too. However, Java has no native support for design by contract, and developers must turn to third party tools.

1.1 Design by Contract for Java

In December 1990 at Sun Microsystems Inc., Patrick Naugthon, Mike Sheridan, and James Gosling started the *Green Project*. It was launched to figure out how computing might develop in future and what markets will grow [Byo]. As the members of the green project considered the consumer and mobile devices to be a quick growing and important market, the need for an appropriate programming language came up. Hence, a programming language named *Oak* was developed. *Oak* was a platform independent, object-oriented, general purpose programming language with support for design by contract [Fir94]. Later, *Oak* evolved to what is known as the Java programming language, and although most features from *Oak* have been adopted or improved by Java, design by contract got lost. Rumours say, this happened due to a tight deadline.

Subsequently, numerous requests for enhancement were submitted to Sun, making DBC one of the most requested enhancements. With the release of Java 1.4, Sun added a simple assertion facility to the Java programming language [Blo99]. However, this tiny step towards design by contract disappointed many developers and strengthened the perception that Java will never have native support for design by contract. The wish for a powerful design by contract implementation for Java still exists and dominates the top 25 RFE's (Request for Enhancements) [Sup01]. To overcome this, many third-party projects have been started, all bringing design by contract to Java. Still, no project could reach a bigger group of software developers and, thus, the application of design by contract in Java can only rarely be seen.

1.2 Goals of this Thesis

Currently, a lot of design by contract implementations for Java exist. Examples, to name only a few, are jContractor [KA05], Contract4J [Asp06], Jass [Bar99, BFMW01], and the Java Modelling Language (JML) [BCC+05]. All these tools implement different concepts to output Java programs enriched with assertions but only a few of them integrate seamlessly into the Java platform and today's development environments. A common approach to implement design by contract is to use a pre-processer or a specialised Java compiler in combination with contracts that are embedded in some kind of Java comment. Alternatively, a tool might ask for contracts that are implemented as Java methods sticking to some kind of naming pattern. Although most tools haven proven to be powerful and bring design by contract to Java, they are not wide spread. This might be due to tool chains which are hard to use or maintain, missing IDE integration, or complicated syntax additions. To give an example for a tool that is hard to maintain, the Common JML Tools can be named. They provide a custom compiler for the Java Modelling Language, and it took almost two years before this compiler was capable of handling Java 5 syntax.

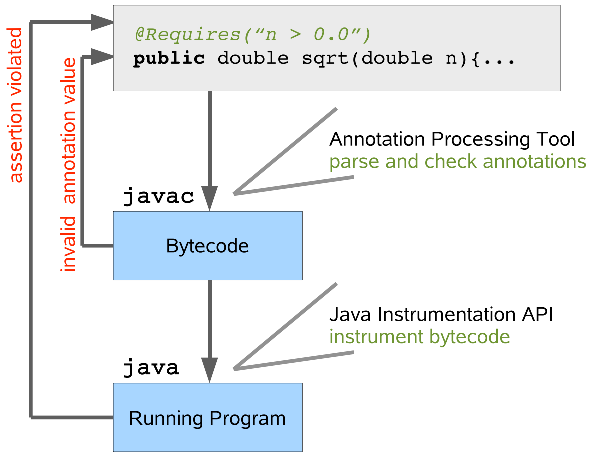
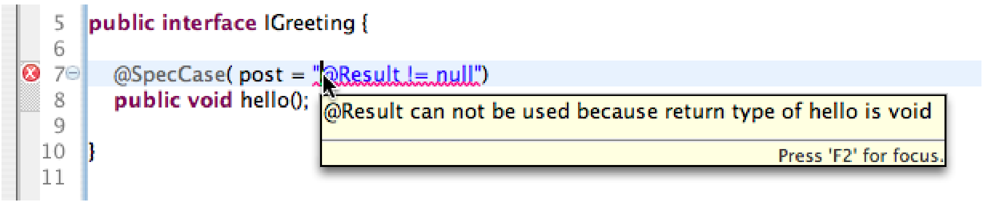


Figure 1.1: The proposed concept of a new DBC implementation for Java.

The goal of this thesis is to leverage support for design by contract in Java by designing and implementing a DBC-tool, that solely uses off-the-shelf Java constructs and focuses on functional, as well as, on non-functional features. Focusing on non-functional features, like ease of use, a seamless integration, and a low maintenance costs, makes this approach different from existent tools, which aimed at functional features only. The main contribution of this thesis will be a DBC implementation that offers a rich feature set at low maintenance costs, and that integrates seamlessly into development environments and build processes. In particular, seamless integration means, that contracts are validated as part of the compilation process, and that contracts are enforced when the program is executed without further user interaction.

Recently added language features, such as *Java 5 Annotations*, the *Annotation Processing Infrastructure*, the *Java Compiler API*, and the *Java Instrumentation API* are going to be utilised to achieve these goals [Sun04b, Dar06, Art06, Sune]. Figure 1.1 shows briefly how these technologies are combined to form a new design by contract tool. It can be seen that the annotation processing environment, which is a plug-in for the Java compiler, is going to be used to validate contract annotations, and that the bytecode instrumentation API is going to be used to enforce contracts. Both technologies, annotation processing and bytecode instrumentation, are part of the standard Java platform and enable DBC to be a first class citizen in the Java world. Figure 1.2 is an example of this seamless integration as it shows a compile error that results from an invalid annotation value. For the user this error is not distinguishable from other compiler errors and will be generated in all IDEs, or when invoking the Java compiler manually. This new DBC approach is not specific to a certain IDE but uses the extension mechanisms provided by the standard Java platform.



```
public interface IGreeting {

    @SpecCase( post = "@Result != null")
    public void hello();

}
```

@Result can not be used because return type of hello is void

Press 'F2' for focus.

Figure 1.2: Screenshot of the Eclipse IDE displaying a compilation error which has been created by an annotation processor.

The work of this thesis can be split up in several subtasks that are outlined in the following.

1. **Examination of Java's features and existing DBC tools.** In order to design a new tool for DBC in Java, existing tools and approaches must be examined. Further on, lately added features of the Java programming language must be evaluated to prove them applicable for realising DBC. These examinations are undertaken to expose the advantages and disadvantages of existing design by contract tools, and to identify the limitations of Java annotations and the annotation processing environment.

2. **Defining a set of specification annotations.** Contracts and behavioural specifications are going to be expressed as Java annotations. Consequently, a set of annotations, that allows to express various assertions in a Java program, must be

defined. These annotations shall not be bound to a certain tool implementation, so that third parties can use and process them as well.

3. **Designing & Implementing a new DBC-tool.** A DBC-tool implementation, that is capable of processing the specification annotations, is to be designed and implemented. It must satisfy the following two main goals: First, contracts in annotations must be validated at compile time so that a programmer is notified about invalid contracts as soon as possible. Second, contracts must be enforced at runtime so that a contract violation causes an error. On top, the tool must be designed to integrate seamlessly into the Java world, meaning that it integrates into developments environments and build processes.

4. **A converter for the Java Modelling Language.** The Java Modelling Language (JML) has a rich feature set and is relatively wide spread so that it is desirable to align with it. As part of the implementation work, a prototypic converter for the Java Modelling Language is developed. It translates JML specification into corresponding Java 5 annotations.

5. **Case studies.** The value of the new DBC-tool is going to be demonstrated in the realisation of different case studies. The goal of these case studies, is to prove the tool being as powerful as existent tools when looking at the raw feature set, and to prove that the implemented concept provides a superior set of non-functional features, like ease of use and seamless integration.

1.3 Outline

Chapter 2 introduces the fundamentals of the Java programming language that are required for the understanding of this thesis. Thereby, the focus is put on recently added Java features and Java bytecode. A basic understanding of the Java programming language is assumed.

In Chapter 3, design by contract in general and how it has been implemented for Java is getting introduced. At the end, a classification and characterisation of existing approaches and tools is presented.

Based on the results of the examinations, in Chapter 4, the creation of a new design by contract tool is reflected. The discussion of different approaches for contract validation and enforcement, as well as the identification of necessary functional requirements is presented.

A set of specification annotations is presented in Chapter 5. A specification for every single annotation describes what semantics it has, and how it is to be validated. Tool implementers shall be guided by this specification, when they implement a tool, that works with these specification annotations.

The main contribution of this thesis, a DBC-tool that implements the specification annotations, is introduced in Chapter 6. It implements the above concept to offer non-functional features, that existent DBC-tools do not provide, and implements a rich feature set. Further, a converter to transform assertions from the Java Modelling Language into equivalent Java 5 annotations is presented.

In Chapter 7 different case studies, that have been carried out, are presented. These case studies will prove the implemented DBC concept being valuable in two ways: First, it offers a rich feature set, matching up with or surpassing most existent DBC implementations, and, secondly, it offers non-functional features that have not been seen in other DBC-tools before. These non-functional features are a seamless integration into whatever tool or environment, and minimal maintenance costs.

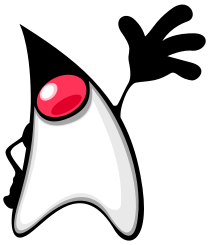
Final remarks about the results of this thesis, about future work and related work can be found in Chapter 8.

Chapter 2

Fundamentals

For the understanding of this thesis the Java programming language plays a prominent role and previous knowledge about Java is assumed. Still, in this chapter some important, but rather uncommon, aspects of Java and Java application programming interfaces are highlighted and explained. Starting with a gentle wrap-up of Java, this chapter focuses on annotating Java code (2.1.1 and 2.1.2) and processing annotations(2.1.3). The second part of this chapter is about Java Bytecode, mainly about ways how to create and manipulate bytecode (Section 2.2.2 and 2.2.3).

The Java Programming Language



The Java programming language is an object oriented platform independent programming language, which is not specialised in design or application. Thus, Java is wide-spread across different computing platforms ranging from consumer devices like cellphones to high-end server applications. Java is owned and maintained by Sun Microsystems and was designed under the lead of James Gosling. Development started in the early 90s as a project called *Oak* targeting embedded consumer-devices and evolved into the Java programming language, which has been released in 1995 [GJSB05].

Due to the rise of the world-wide web and Java's applet technology, it was initially perceived as the programming language for the internet and quickly gained popularity. Nowadays, Java is one of the most popular programming languages [BV06] and plays a prominent role for server, client, and consumer device applications. The sixth version of Java has been released in December 2006. Around that time, the Java platform has been made available under the General Public License version 2 (GPL) which makes most of the Java technology open source [Sun06a]. Still, long before making Java open source, Sun started the Java Community Process (JCP) which gives third parties the opportunity to participate in the Java development process by originating Java Specification Requests (JSR)[Jav98]. JSRs are standardised documents which get approved or declined in a transparent process by the JCP Executive Committee. Since its foundation in 1998, all new Java features and add-on libraries went through the JCP. The following JSRs are important for this work:

- JSR 41 - A simple assertion facility [Blo99].
- JSR 175 - A Metadata Facility for the JavaTM Programming Language [Blo02].
- JSR 269 - Pluggable annotation processing API [Dar06].
- JSR 199 - Compiler API [GvdA02].

The following section will focus on annotations and deals with JSR 175 and JSR 269. The assertion facility, JSR 41, is introduced in Chapter 3, whereas the Compiler API (JSR 199) is introduced at the end of this chapter.

2.1 Annotating Java Code

According to the Oxford American dictionaries an annotation is '*a note of explanation or comment added to a text or diagram*'. Since source code is text only, this definition covers what semantics annotations in Java code have. However, from a technical point of view its not defined how an annotation looks like, and how annotations are added to Java code. This sections will introduce annotation techniques for Java source code. First of all, when talking about Java and annotations, one has to differentiate between an era prior and subsequent to Java 5. Prior to Java 5, annotations could only be embedded in Java comments and needed to be processed by an external tool. A popular example is the Javadoc tool and doclets which will be introduced in Section 2.1.1. With Java 5 the annotation mechanism in Java was reviewed and a new 'general purpose annotation facility' [Sun04b] was added to Java (Section 2.1.2).

2.1.1 Embedded Annotations

The nature of a comment in a programming language is to embed annotations. Such an annotation might be a note for programmers, documentation, or it might be intended to be processed by external tools. For Java, the doclet API [Sunb] exists and enables programmers to write custom processors for comment-based annotations. A prominent example that uses the doclet API is the Javadoc tool, which creates HTML documentation from source code. In Listing 2.1 the return value of a method is documented with the text following the doclet-tag `@return`.

More generally speaking, a doclet may specify a set of tags which it is able to process. The output of a doclet is not limited to certain file formats or types making it a universal

```java
/**
 *
 * @return Returns a string.
 */
public String toString(){ ... }
```

Listing 2.1: Javadoc-documentation of a method return value.

```java
/**
 *
 * @pre obj != null
 */
public void add(Object obj){ ... }
```

Listing 2.2: A doclet-tag representing a pre-condition.

processor for Java source code. For instance, a doclet could have defined the pre-tag representing a pre-condition (see Listing 2.2). The content of the pre-tag could be used to create a type that actually implements the pre-condition.

Limitations

A consequence of the fact that annotations reside in comments only, is that they get lost during compilation. Hence, further processing e.g. with reflective programming (Section 2.1.3) or bytecode instrumentation (Section 2.2.3) is not possible. In addition, the content of doclet-tags are Strings in a doclet-specific format which might be subject to further processing like parsing. Parsing can be troublesome when different doclets use tags with the same name but different semantics. For instance, consider Listing 2.3 were two different semantics of doclet-tags collide.

```java
/**
 *
 * @throws NullPointerException In case the param <i>obj</i> is null.
 * @throws NullPointerException -> obj == null
 */
public void add(Object obj) throws NullPointerException { ... }
```

Listing 2.3: A doclet-tag, throws, with two semantics.

For the first, the throws-tag contains a fragment of the documentation created by the HTML-doclet. Secondly, throws is used to define a boolean expression which might be evaluated by a DBC-tool.

Other embedded annotation

Aside from doclet-tags and the doclet API, third parties are free to define their own comment-based annotations, and to process them in a propriety fashion. Usually, such tools include a parser for Java source code which is capable of locating and preparing comment-based annotations for further processing.

2.1.2 Java 5 Annotation

To overcome the shortcomings of comment-based annotations, a new annotation facility for Java was designed and shipped with Java 5 for the first time. In terms of the Java 5 annotation facility, an annotation is a language type, similar to interfaces, making it a first class Java citizen being easy to process and more flexible to apply. This section will introduce Java 5 annotations by outlining how annotation types are defined and how they are added to a set of annotation targets. Subsequently, Section 2.1.3 will show how Java 5 annotations can be processed.

Defining Java 5 annotations

Java is a strongly typed language which means that the type of a variable or expression can be determined at compile-time. Basicly, there are two fundamentally different kinds of types.

- Primitive Type: Primitive types are boolean and numerical values. They do not share data with other primitive types and are made up from the following primitive-types which differ in size and semantics: boolean, char, byte, short, int, float, long, and float.
- Reference Type: A reference type is either a Class type, an Interface type, or an Array.

In addition to the reference types above, there are some specialised types. Such are the enum-type and the annotation-type (chapter 4 in [GJSB05]). The latter one is a decent of the interface type and defined by the grammer in Listing 2.4.

```
AnnotationTypeDeclaration:
        InterfaceModifiersopt @ interface Identifier AnnotationTypeBody

    AnnotationTypeBody:
        { AnnotationTypeElementDeclarations_opt }

    AnnotationTypeElementDeclarations:
        AnnotationTypeElementDeclaration
        AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration

    AnnotationTypeElementDeclaration:
        AbstractMethodModifiers_opt Type Identifier ( ) DefaultValue_opt ;
        ConstantDeclaration
        ClassDeclaration
        InterfaceDeclaration
        EnumDeclaration
        AnnotationTypeDeclaration
        ;

    DefaultValue:
        default ElementValue
```

Listing 2.4: Grammer of the annotation type definiton.

Aside from nested types or constants, an annotation type definition consists of the @interface-keyword (line 2) and an optional list of method-like declarations (line 12). Similar to interfaces, methods declared in annotation types are abstract, which means that they do not allow every method modifier (e.g. private, static) and do not have a method body. However, method declarations in annotation types obey further restrictions making them distinct from usual method declarations.

1. The return type of a method in an annotation type must be either a primitive type, an enumeration, the types java.lang.String or java.lang.Class, an annotation different from the annotation itself, or an array of one of the previous types.

2. An annotation type may define a default return value for its methods using the default-keyword.

3. Methods in annotation types cannot define parameters and cannot have a throws-clause.

Due to the differences of methods in annotations and other reference types, and due to their informative nature, methods of annotation types are also called *attribute*. Besides, the annotation type cannot be generic and cannot extend other types. An example of an annotation type-definition is shown in Listing 2.5.

```
1  @interface PreCondition {
2
3    String value();
4    Visibility visibility() default Visibility.PUBLIC;
5
6    enum Visibility = { PUBLIC, PROTECTED, PRIVATE }
7  }
```

Listing 2.5: The type-definition of a pre-condition annotation.

Starting with the @interface-keyword in line 1, this annotation type declares two attributes and the nested enumeration-type Visibility (line 6). The value-attribute, line 3, returns instances of java.lang.String but does not declare a default value. A default value can be found at the visibility-attribute in line 4 which returns Visibility.PUBLIC if no other value gets assigned to it.

Annotation targets

The elements of a Java element, where an annotation can be placed at, is called annotation target. As defined in the Java language specification [GJSB05] an annotation is a modifier and, hence, can be placed at:

- Type definitions (E.g. classes, interfaces, enum-types, and annotations[1]),

- Fields (this includes enum constants),

[1] If the target is an annotation type itself, the term meta-annotation is used.

- Constructors,
- Methods,
- Parameters of methods and constructors,
- Local Variables,
- Packages.

By default, an annotation can be placed at all targets from the list above. However, there are some pre-defined annotations which restrict the application of an annotation. For instance, the meta-annotation java.lang.annotation.Target can be used to denote a subset of above targets for an annotation type. Listing 2.6 shows a modified version of the PreCondition-annotation which is restricted to be used with methods only (see line 3).

```
 1  import java.lang.annotation.Target;
 2
 3  @Target( ElementType.Method )
 4  public @interface PreCondition {
 5
 6    String value();
 7    Visibility visibility() default Visibility.PUBLIC;
 8
 9    enum Visibility = { PUBLIC, PROTECTED, PRIVATE }
10  }
```

Listing 2.6: An annotation which is applicable to methods only.

Application of Annotations

Above paragraphs have shown what a Java 5 annotation is and how annotation types can be defined. This section presents how annotations are applied. Listing 2.7, line 3, shows the application of the @PreCondition-annotation. The attribute value is assigned with the String 'obj != null' whereas the attribute visibility remains unchanged so that it is still assigned to its default value (Visibility.PUBLIC).

```
1  public class Buffer {
2
3    @PreCondition(value = "obj != null")
4    public void add(Object obj) { ... }
5  }
```

Listing 2.7: Using the PreCondition-annotation.

Note, since an annotation is understood as a modifier, public could appear before @PreCondition. However, coding-guidelines recommend to start with annotations.

Despite the attractiveness of annotations, their application is restricted by mainly two rules [GJSB05].

1. A certain annotation type can be placed at its target only once, otherwise a compile error is raised.

2. The values of annotation attributes must be compile time constants, otherwise a compile error is raised.

Listing 2.8 shows the violation of these rules, as the value which is going to be assigned to the attribute value is not a compile time constant (line 4) and, as the PreCondition-annotation is used twice (line 3 and 6).

```
 1  public class Buffer {
 2
 3    @PreCondition(
 4      value = foo(),
 5      priority = Visibility.PRIVATE)
 6    @PreCondition(value = "obj != null")
 7    public void add(Object obj){ ... }
 8
 9    public String foo(){
10      return "I'm not a compile-time constant!";
11    }
12  }
```

Listing 2.8: Violating the rules for annotation application.

If an annotation shall be used more than once for a single target, it needs to be wrapped into another annotation, defining a single attribute which is an array of the desired annotation type. An example of such a container-annotation is given in Listing 2.9. Listing 2.10 shows how the annotation is used.

```
@interface PreConditions {

  @PreCondition[] value();
}
```

Listing 2.9: A container-annotation for the @PreCondition-annotation.

```
@PreConditions(
 value = {
  @PreCondition(value = "obj != null"),
  @PreCondition(value = "!isFull()")
 })
public void add(Object obj){ ... }
```

Listing 2.10: Applying a container-annotation.

Inheritance

Although being modifiers, annotations are not inherited when an annotated type is getting subclassed. However, there is one exception of this rule. The Java Standard Platform defines the meta-annotation java.lang.annotation.Inherited which enables annotation inheritance. The semantics of the Inherited-annotation is that if

> *the user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, than the class's superclass will automatically be queried for the annotation type* [Sunc].

This query process will be repeated for all supertypes, but will skip interfaces. Because of this very limited support for inheritance, one might say that inheritance of Java 5 annotations is an open issue that is left to third party developers.

Shorthands

When using annotations, the Java language specification defines syntactical shorthands [GJSB05]. One can be put in for empty annotations, which are annotations without any attribute, another one eases the use of annotations with a single attribute. A third one simplifies the use of attributes that expect an array.

- If an annotation defines no attributes (a marker annotation) or all attributes do have default values, the round brackets are optional. Consider the annotation @NonNull which can be applied either way.

  ```
  public @interface NonNull{  /*empty, no attributes */ }
  ```

  One can type out the open and closing brackets or

  ```
  void add(@NonNull() Object obj){ ... }
  ```

  simply omit them.

  ```
  void add(@NonNull Object obj){ ... }
  ```

  An annotation which does not define attributes, like the @NonNull-annotation, is called marker annotation.

- If there is only one attribute without a default value and its name is value, or the only attribute is called value, the name of can be omitted. For instance, the @PreCondition-annotation (Listing 2.6, page 12) is suitable for this shorthand. It can be applied either way, typing out the name of the attribute and the '='-sign,

  ```
  @PreCondition(value = "obj != null")
  void add(Object obj){ ... }
  ```

  or omitting both, leaving the value only.

  ```
  @PreCondition("obj != null")
  void add(Object obj){ ... }
  ```

Annotations like the @PreCondition-annotation are called single element annotations. However, as soon as an attribute, different from value, is getting assigned, all attributes must be qualified with their names.

- When an attribute of an annotation is defined as an array of elements but only a single element exists, the array creation brackets can be omitted. To give an example, the @PreConditions container annotation (Listing 2.9) has a single attribute which data type is an array of the @PreCondition annotation. In case, the array as length one, the annotation can be written either way. First with the curly brackets,

  ```
  @PreConditions( value = { @PreCondition( value = ”obj != null”) })
  void add(Object obj){ ... }
  ```

  or the shortened way omitting the brackets and attribute names.

  ```
  @PreConditions( @PreCondition(”obj != null”) )
  void add(Object obj){ ... }
  ```

Outlook 1: JSR 308 - Annotations on Java Types

The Java specification request 308 proposes to allow annotations on other types than just class, method, field, and variable declarations [CE06]. An example is to annotate an object creation statement with an annotation expressing that the object is read-only:

```
...
Foo foo = new @ReadOnly Foo();
...
```

JSR 308 is targeted for Java 7 and currently new targets for annotations are discussed. At this point merely annotations on type related constructs like variable/field declaration, new, instanceof, extends, or throws are discussed, but annotations for loops or arbitrary blocks are also on the agenda.

Outlook 2: JSR 305 - Annotations for Software Defect Detection

The goal of JSR 305 is to assemble a set of annotations for ‘software defect detection’ which is to be included into the Java standard platform [Pug06]. Currently, different software vendors and third party developers use their own annotations for very similar purposes, e.g. a @NonNull annotation. The goal of JSR 305 is to ease the burden of using such an annotations by defining a standard set and, thus, promoting compatibility. Initially, the annotations of JSR 305 will bundled in a standalone package, but at the long run it is targeted for Java 7.

2.1.3 Processing Java 5 Annotations

In general, working with annotations and their values is called annotation processing. Java 5 annotations can be processed at runtime via Java’s reflection capabilities or they can be processed using an annotation processing tool. The first part of this section gives an idea of the use of reflective programming, and the second part emphasises the annotation processing environment shipped with the Java development kit.

Reflective Programming

Reflection, or sometimes called introspection, is a programming language feature which empowers a program to inspect and maybe even modify its own structure and variable values at runtime. The programming paradigm of reflective programming emphasises the use of reflections. Java supports reflection natively through classes in the package java.lang.reflect along with the java.lang.Class-class.

At runtime, every object in the virtual machine refers to an instance of java.lang.Class which can be retrieved by calling java.lang.Object.getClass(). It represents the type definition of an object, as defined in source code, and allows to explore and modify its structure. Figure 6.7 shows a selection of types and methods from the reflection API and how the types relate to each other.

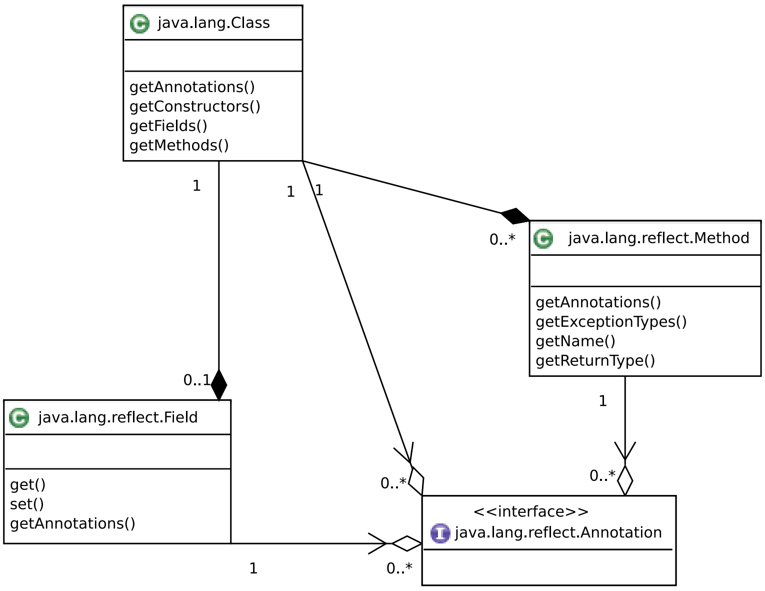


Figure 2.1: Reflection-capabilities in Java (selection).

The recommended use of reflection is to accommodate developer tools like debuggers, object inspectors, or graphical user-interface builders, but reflections may also be used to inspect annotations and their attribute values. Listing 2.11 gives an example of how to obtain the PreCondition-annotation (see Listing 2.6 and 2.7) and how to access its attribute values.

In line 1 and 2, a reference of the PreCondition-annotation is obtained and stored into a local variable. Line 5 demonstrates that annotation attributes can be accessed via

```
Method method = Buffer.class.getMethod("add");
PreCondition reference = method.getAnnotation(
  PreCondition.class);

assert reference.value().equals("obj != null");
```

Listing 2.11: Accessing annotation values at runtime via reflection.

standard method calls and asserts that the value of the attribute did not change.

Pluggable Annotation Processing API

In addition to reflective programming, annotations can be processed by an external tool or processor which can read annotations from source code and bytecode. The Java 5 software development kit delivered by Sun Microsystems contains such a tool - the annotation processing tool, shortend APT [Sun04a]. It consists of an API for which annotation processors are developed, and an executable which runs these annotation processors. However, the annotation processing tool (APT) is not part of the Java Standard Platform which means that other Java vendors are not obligated to provide such an annotation processing tool. Although Sun made APT open source, the Java specification request 269 (JSR 269) came to life. Its objective is to replace APT with a processing environment that is integrated into the Java compiler, and that is part of the official Java Standard Platform [Dar06]. Since version 6, the JSR 269 is part of the official Java release, coexisting with the annotation processing tool which will not be available in future Java versions. Due to that and the advantages of JSR 269, the outdated APT will be not considered in this thesis.

JSR 269 has been launched with the title '*pluggable annotation processing API*' which is good summary of its goals. Its objective is to provide an annotation processing facility which is integrated into the Java compiler and easy to manage. To do so, the program javac was extended with the ability to discover and run annotation processors. By default, every annotation processor, being present on the classpath, is executed when compilation is performed. In addition, a couple of program switches have been added to javac in order to get a fine-grained control over the annotation processors [Sun06b][2].

`-proc:{none, only}` Control whether annotation processing and/or compilation is done.

`-processor <class1>[,<class2>,<class3>...]` Names of the annotation processors to run; bypasses default discovery process.

`-processorpath <path>` Specify where to find annotation processors.

`-d <directory>` Specify where to place generated class files.

`-s <directory>` Specify where to place generated source files.

[2] Run javac -help to get this information.

In its essence an annotation processor is an implementation of the javax.annotation.processing.Processor-interface (see UML diagram in Figure 2.2). The Java compiler will call the process-method of this interface while providing a processing environment. This processing environment can be used to interact with the tool running the annotation processor, usually the Java compiler. The interface javax.annotation.processing.Messager, for instance, can be utilised by an annotation processor to report errors or informational messages.

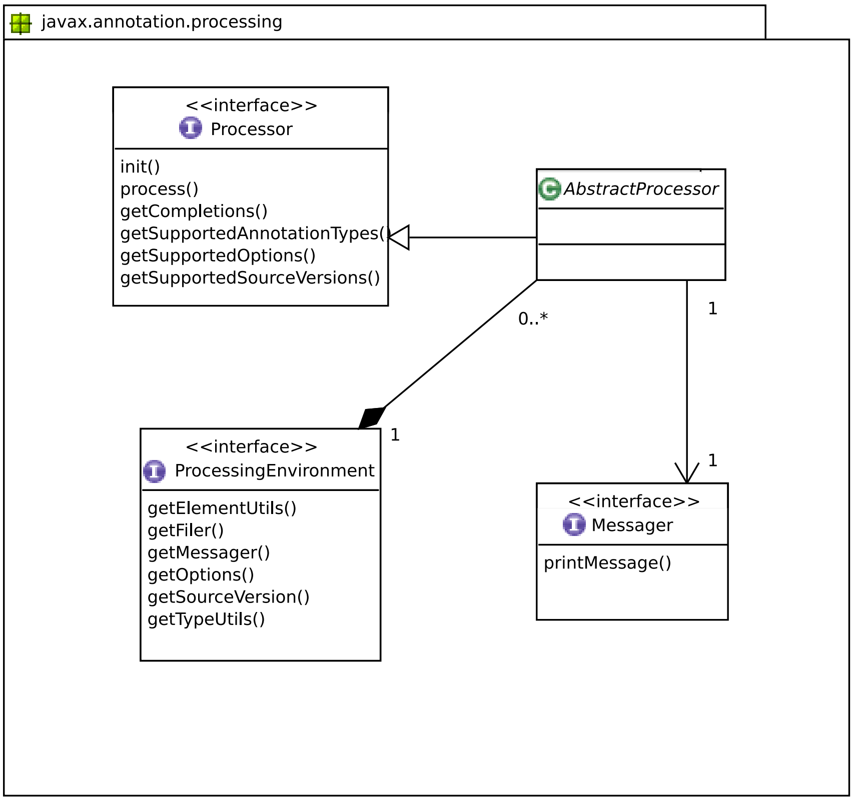


Figure 2.2: The javax.annotation.processing-package which hosts the API for JSR 269-based annotation processing.

In addition to the classes and interfaces above (Figure 2.2), the annotation processing environment provides programmatically access to the structure of the class currently processed. Therefore a Java model API is used to reflect the structure and type information of Java classes. This API is split up into two parts, an element API and a type API. The type API focuses on the different types like primitive types, array types, or reference types, whereas the element API models the elements of the Java programming language like classes, methods, variables and so on. Figure 2.3 shows an abridged view of the element API as it can be found in the package javax.lang.model.element. Besides the elements of the Java programming language, Figure 2.3 shows that an interface El-

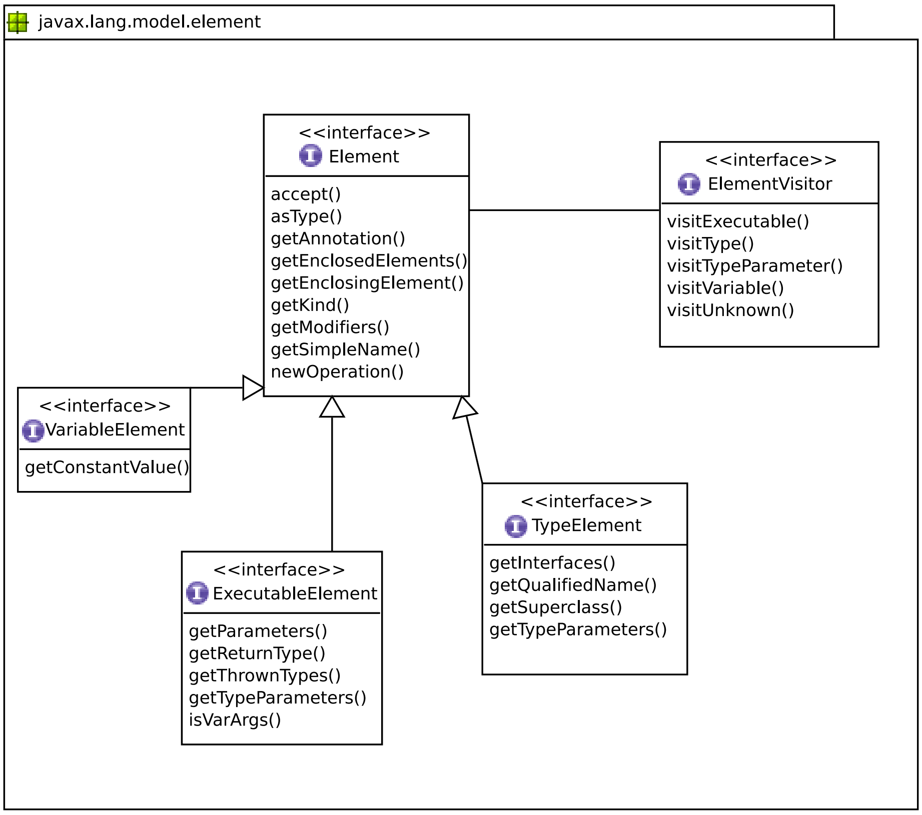


Figure 2.3: Abridged view of the package javax.lang.model.element.

ementVisitor exists. Utilising the visitor pattern is the proposed way to process models and one may benefit from default implementations of this interface provided by Sun.

Integrated Annotation Processors

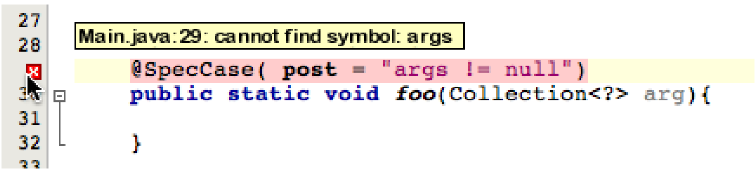
As mentioned above, the Java compiler is capable of running JSR 269 compliant annotation processors. Besides, the integrated development environment Eclipse[3] and NetBeans[4] do run and integrate annotation processors [Net06, Har06, HGH06]. Figure 2.4 shows a screenshot of the NetBeans IDE running an annotation processor and displaying an error message created by that processor.

Other Annotation Processing Tools

In addition to the pluggable annotation processing API (JSR 269), developers and software vendors are free to develop their own custom annotation processing tools. Since

[3] www.eclipse.org

[4] www.netbeans.org



Main.java:29: cannot find symbol: args

```
@SpecCase( post = "args != null")
public static void foo(Collection<?> arg){

}
```

Figure 2.4: Integration of an annotation processor in NetBeans.

annotations are present in source-code and, depending on their retention policy, byte-code, ISVs can have a custom toolkit to process annotations.

2.2 Java Bytecode

Java is an platform independent programming language which means that source code must be compiled only once and can be executed on nearly all computing platforms. To achieve this, Java source code is not compiled into machine code, but into Java bytecode which can be executed by a platform specific virtual machine. So to speak, Java bytecode is machine code for the Java virtual machine (JVM). This section is about Java bytecode, especially about bytecode creation and manipulation.

2.2.1 Classfile-Format

The Java virtual machine specification exactly defines the format for class files. At this place, the basic concepts are outlined only, so that for an in-depth introduction [LY96] should be consulted.

Although a single Java source file may define more than one class, at bytecode level each class definition corresponds to one class-file, all ending with the class-suffix. Basically, a class files consists of structural information, such as defined methods, fields, annotations, its type hierarchy. and the actual opcodes, representing the code of a method. The following explains the structure of class files, visualised in Figure 2.5 more deeply:

Header The header identifies the file as an Java bytecode file, and informs about the class file version.

Constant Pool The constant pool is a table storing constant values (e.g. String literals), the name and descriptors of fields and methods, and names of other classes. Every entry in the constant pool can be referenced by an index.

Access Rights Access rights are stored in a single bitmask containing information about the visibility of a class and whether its abstract, static, or final. For instance the modifiers public and final result in the bitmask `0x0011`.

Class Hierarchy The information stored in the class hierarchy field denote a supertype and all directly implemented interfaces. Java allows to have one supertype only, but an arbitrary number of interfaces.

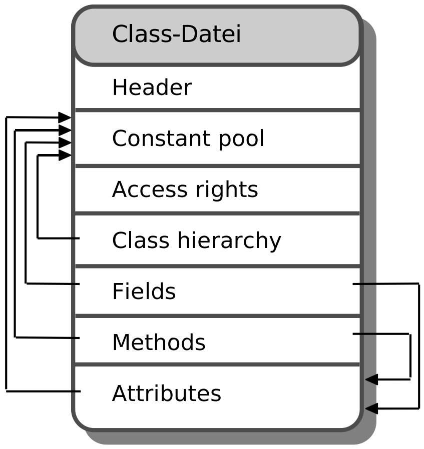


Figure 2.5: Structural view of a class file.

Fields This part of a class files lists all defined fields, whereas a single field is defined by its name, its type, access rights, and a number of attributes. An attribute can be used to represent annotations or extended debug information.

Methods Similar to *Field*-information, the methods-part of a class files is a listing of all defined methods. Methods consist of a descriptor, encoding the signature, its name, access rights, and a set of attributes. The actual code of a method is stored in an attribute and linked to the method.

Attributes Attributes in classfiles may contain all kinds of information. The virtual machine specification defines a number of them but ISVs are free to add propriety attributes to a classfiles. To give an example, pre-defined attributes are used to store the actual code of a method, and to store annotations.

2.2.2 Creating Java Bytecode

In almost every case, Java Bytecode is created by a compiler, such as javac, which is shipped with the Java platform, or by third party compilers, like the Eclipse JDT compiler. They read Java source files and create class files, the file representation of Java bytecode. A Java compiler might be invoked manually or automatically by an IDE, but usually three distinct steps can be identified. First, Java source code is written. Secondly, the source code is compiled into bytecode and, third, bytecode is executed by the Java virtual machine. To ease the integration of the Java compiler into an IDE, the Java Compiler API was introduced. It enables one to programmatically interact with the Java compiler. The Compiler API evolved from JSR 199 and is part of the standard Java platform since version 6 [GvdA02].

The Java Compiler API

The Java Compiler API can be used to start and control the Java compiler at runtime and, thus, dynamically create bytecode bytecode from source code. Moreover, the Java compiler API does not depend on a file-based representation of source files or bytecode files, but can process arbitrary streams of bytes. For instance, a program can dynamically create source code which is compiled and loaded into the Java virtual machine (JVM). In its essence, the Java Compiler API can be controlled with the following four interfaces:

javax.tools.JavaFileObject A JavaFileObject is an abstraction of Java files regardless being a source file (*.java) or a binary file (*.class). For instance, utilising this interface allows to create source files in memory, retrieve them through a network connection, or read them from the hard disc.

javax.tools.JavaFileManager During compilation, the JavaFileManager is used to manage JavaFileObjects. E.g. it may reuse source from a cache or load it from a network resource.

javax.tools.JavaCompiler.CompilationTask A compilation task consists of a set of JavaFileObjects which are to compile. The CompilationTask can issue warnings and errors that occurred during compilation.

javax.tools.JavaCompiler The interface to the Java Compiler which does the actual source code to bytecode transformation. The JavaCompiler is used to create a CompilationTask from a set of compilation units. The Java standard platform is delivered with an implementation of this interface but ISVs are free to provide their own implementation of the JavaCompiler interface. The Eclipse JDT project, for instance, provides a Java compiler which can be accessed via the JavaCompiler interface as well [Art06].

2.2.3 Manipulating Java Bytecode

Normally, bytecode is created through compilation of source code and remains unchanged unless further compilation is performed. However, bytecode can be created or changed manually without a compiler. This, so called bytecode manipulation or instrumentation, can be used for profiling, debugging, or to add new features to a program. Bytecode can be manipulated either statically by changing classfiles, or dynamically, when a class is loaded into the Java virtual machine. The following outlines one static and two dynamic approaches for bytecode instrumentation.

Static Bytecode Instrumentation

Static bytecode instrumentation is performed prior to runtime and changes the content of classes in the file system. Although it is an additional step it might appropriate when byte instrumentation is required only once, e.g. for bytecode obfuscation or encryption.

Custom Classloader

In Java, a classloader is used to locate and load the content of a class file into the virtual machine. Third party developers are free to implement custom classloaders and, thus, adding new class loading capabilities to the Java platform. For instance, such are loading classes from an encrypted file, via a network connection, or performing bytecode manipulation. Such an on-the-fly bytecode manipulation is useful whenever the instrumentation must be computed dynamically or when class files could not be manipulated statically (e.g. class-files are write protected). However, dynamic bytecode instrumentation always imposes runtime penalties as it slows down class loading.

Prior to Java 5 dynamic bytecode instrumenting could only be achieved by a wrapper program. Such a program will be a stand-in for the actual program, forwarding all program parameters, but installing a custom classloader that performs bytecode instrumentation. The Java instrumentation API, introduced in Java 5, makes such stand-in programs dispensable because now the JVM supports bytecode instrumentation natively.

The Java Instrumentation API

The instrumentation API is part of the Java standard platform and embraces dynamic bytecode instrumentation. It consists of a set of interfaces, a new parameter for the program java, and a new attribute for manifest-files in jar-archives.

In terms of the instrumentation API, a *Java agent* is a program which is loaded prior to the main-method of a program and has the ability to instrument all classes getting loaded for that particular program. A Java agent must define a premain-method, see Listing 2.12, which is called prior to the main-method [Sune].

```
public static void premain(String agentArgs, Instrumentation inst);
```

Listing 2.12: Method signature of a premain-method.

The passed java.lang.instrumentation.Instrumentation-instance is used register bytecode-transformers and, hence, the entry point for bytecode instrumentation.

To enable instrumentation at runtime, the program java needs to be called with the parameter javaagent followed by the name of an archive. That archive must contain a class with the premain-method, and point to that class via an entry in its manifest file. Listing 2.13 shows such a command.

```
/>java -javaagent:instrument.jar foo.bar.HelloWorld
```

Listing 2.13: The command used to enable dynamic bytecode instrumentation.

Chapter 3

Design by Contract in Java

Except for the simple assertion facility, Java does not support design by contract (DBC), and it is left to third party developers to implement DBC for Java. This chapter is about DBC in general, and about existent DBC implementations for Java. In Section 3.1 the basic ideas of DBC and frequently used terms are recapitulated, whereas in Section 3.2 tools and approaches, that have been used to implement DBC, are introduced. Section 3.3 gives an overview and review of these approaches, and clarifies some common concepts.

3.1 Design by Contract

With DBC Bertand Meyer developed a programming methodology to increase the reliability of software systems. Meyer defines reliability as the combination of correctness and robustness or, put in other words, the absence of bugs [Mey92]. In the style of the business world, design by contract uses the terms *client* and *supplier*. Usually, when a client and a supplier have a contract, obligations must be fulfilled and some benefit is expected. The same is true for design by contract and programming. For instance, calling a method involves two parties: a caller (*client*) and a method (*supplier*). Besides handing down words from the business world to programming, design by contract focuses on the *contract* which states:

- what conditions a caller must fulfil in order to be allowed to call a certain method, and

- what conditions a method guarantees, assuming the caller stuck to his part of the contract, once its execution is done.

In other words, a caller must fulfil certain pre-conditions to call a method and, in return, the method fulfils certain post-conditions. The concept of pre- and post-conditions dates back to the work of C.A.R Hoare and the Hoare triple [Hoa69]. A Hoare triple is defined as:

$$\{P\}\ C\ \{Q\} \tag{3.1}$$

where P and Q are logical formulae and C is a program or command. The reading of the Hoare triple is that whenever P holds true in the state before starting the program C, C will stop in a state where Q holds. Actually, P is called *pre-condition* and Q is called *post-condition*. Loosely speaking, DBC is the pragmatic application of the Hoare triple.

The following terms are frequently used when talking about design by contract:

Pre- & Post-Conditions A pre-condition is an assertion which must hold true before a method is executed but after the method execution has been initialised, meaning, local variables are loaded already. In return, a post-condition is an assertion which must hold true after the execution of a method has terminated but the before the program counter jumps back to the calling method.

Class-Invariants As pre- and post-condition are specified for a particular method only, class-invariants are valid for a whole class and, thus, for each method of that class. Class-invariants are checked whenever pre- or post-conditions are checked.

Pre-State The pre-state is the state of an object in which a pre-condition is checked. The Old-construct is used to capture the pre-state values of an object so that it can be used in the post-state.

Post-State The post-state is the state of an object in which a post-condition is checked. In the post-state, the pre-state values can be accessed via Old, and method return values, if existent, can be accessed.

Inheritance Pre- and post-conditions and class-invariants are inherited in terms of object-oriented programming. How inheritance works, depends on the DBC implementation one uses. In Section 3.2 different realisation are introduced.

Note that design by contract is not to be mistaken with *defensive programming* which aims for the direct opposite. Instead of specifying the behaviour of program and monitoring this specification for violation, defensive programming points out that a program should handle all possible kinds of input gracefully, thus, increasing complexity.

3.2 Existing DBC Tools & Specification Languages

This section introduces a wide range of DBC-implementations for Java. Starting with the simple assertion facility natively supported by Java (3.2.1), some elaborated tools like Jass (3.2.2), the Common JML Tools (3.2.3), jContractor (3.2.4), and Contract4J (3.2.5) are presented. Further, two rather technical concepts of how to implement DBC in Java are presented in 3.2.6 and 3.2.7.

3.2.1 Build-in Assertion Facility

Evolved from JSR 41 [Blo99], Java 1.4 and higher includes a simple keyword-based assertion facility. An assertion is defined as a boolean expression which is ought to be true at a certain state. In Java, assertions are expressed using the assert-keyword [GJSB05]. An assertion violation will raise an java.lang.AssertionError, a 'fail-hard' policy, which makes it a simple but powerful construct to improve code quality. Listing 3.1 shows how the assert-statement is used in Java code.

```
public void clear(){
  ...
  assert isEmpty() : "Buffer not empty after 'clear'";
}
```

Listing 3.1: Using the assert-keyword.

The assert-keyword is followed by a boolean expression and an optional object which will be treated as error message.

Technial realisation

To not impose runtime penalties, Java assertions are disabled by default. To enable assertions at runtime, one has to use the switches of the java-program, outlined below.

-ea $[:\langle \mathit{packagename}\rangle^{*}|:\langle \mathit{classname}\rangle^{*}]$ Without any parameters, the -ea switch enables assertions for all classes which make use of the assert-statement. Optionally, a colon-separated list of class or package names might be specified, which is treated as a filter and enables assertions for these classes only. The ea switch is an alias for *enableassertions* which can be used in the same manner.

-da $[:\langle \mathit{packagename}\rangle^{*}|:\langle \mathit{classname}\rangle^{*}]$ Disable all assertions or only those specified via the package and class-name filters. The switch *disableassertions* can be used synonymical.

-esa Enables assertions for all system classes, the same as *enablesystemassertions*. A package or class name based filter does not exist.

-dsa Analogue to esa, this switch disables all system class assertions. Shorthand for *disablesystemassertions*.

Technically speaking, assertions are very simple as the compiler transforms the assert-statement into an if-statement. Thereby, the boolean expression gets negated so that an error will be raised if the boolean expression does not hold. The enablement of assertions is realised with a seconded enclosing if-statement which checks for a boolean field. This boolean field is generated by the compiler and named $assertionsEnabled. It will be initialised by the system classloader depending on the switches introduced above. At bytecode level, Listing 3.1 is equivalent with Listing 3.2.

```
public void clear(){
  ...
  if($assertionsEnabled){
    if(!isEmpty()){
      throw new AssertionError("Buffer not empty after 'clear'");
    }
  }
}
```

Listing 3.2: The ‘decompiled’ assert-statement.

3.2.2 Jass - Java with Assertions

As part of his master’s thesis, Detlef Bartezko developed Jass which founds on former work of Clemens Fischer and the JaWa (Java with Assertions) project [Bar99][1]. Jass uses a comment-based syntax and a pre-compiler to implement design by contract. The following outlines the Jass feature-set, yet, a comprehensive introduction to Jass can be found on its project website [Jas05].

Class-Invariant

Jass allows the definition of class-invariants with the restriction that they have to be defined at the end of the class. They are checked in every visible state of the class where visible states are defined as all method entry and exit points, and the end of constructors. The invariant statement, as shown in Listing 3.3, consists of the invariant-keyword followed by a boolean expression. The boolean expression may extend the standard Java boolean expression by quantifiers which will be introduced later in this section.

```
public class Buffer implements java.lang.Cloneable {
  ...

  /** invariant fBufferSize >= 0 **/
}
```

Listing 3.3: Invariant in Jass.

Method Specifications

In Jass, pre- and post-conditions can be used to annotate methods and constructors. A pre-condition (see Listing 3.4) is written as multiline comment starting with the require-keyword and followed by a boolean expression. Similar to invariants, these boolean expressions might be extended with quantifiers. Pre-conditions must be the first line in a method body.

[1] Note, this section refers to Jass version 2.x and not to Jass version 3.x, which uses the Java Modelling Language (JML) as input language.

```
public void add(Object obj){
  /** require obj != null; !isFull() **/

  ...
}
```

Listing 3.4: Pre-condition in Jass.

Analogue to pre-conditions, post-conditons are multiline comments and have to be the very last statement of a method body. They start with the ensure-keyword followed by a boolean expression. As one can see in Listing 3.5, the special variables Return and Old can be used in Jass. As the sound of their names indicate, Return refers to the return value of the current method whilst Old refers to the state of the this-reference in the pre-state. The latter one, Old, imposes that Jass is able to create a copy which is to be achieved by implementing java.lang.Cloneable. Note, that in Section 3.3.3 the semantics of Old is discussed.

```
public Object removeLast(){
  ...

  /** ensure Return != null; Old.fBufferSize = fBufferSize − 1 **/
}
```

Listing 3.5: Post-condition in Jass.

Loop-(In)Variants

In addition to class-invariants and pre-/post-conditions, the Jass feature-set contains loop-variants and loop-invariants. A loop-invariant is a property which may not change as the loop executes, whereas a loop variant has to decrease in every loop iteration. Listing 3.6 shows how they are used in Jass.

```
for(int i=0; i < 100; i++){
  /** invariant B **/
  /** variant D **/

  ...
}
```

Listing 3.6: Loop-variant and loop-invariant in Jass.

Quantifiers

Quantifiers in Jass allow one to check a condition for a whole range of elements. Listing 3.7 shows an invariant which ensures that no element in a buffer is the null-reference.

```
/** invariant forall i : {0..fBufferSize} # get(i) != null **/
```

Listing 3.7: The forall-quantifier in Jass.

The exists-quantifier is similar to the forall-quantifier. It ensures that a condition is satisfied at least once for a number of elements.

Retry and Rescue

The retry and rescue construct can be used to handle contract violations programmatically and to react upon them. Listing 3.8, for example, shows how a retry and rescue strategy is implemented in the case the null-element is added to a buffer.

```
public void add(Object obj){
  /** require [obj_null] obj != null; !isFull() **/

  // ..code goes here...
  /** rescue catch (PreconditionException e) {
                  if (e.label.equals("obj_null")) {
                     o = new DefaultObject();
                     retry;
                  }
                  else throw e;
         }
}
```

Listing 3.8: The retry and rescue construct in Jass.

Check

The check-statement works just as the standard Java assert-statement (see Section 3.2.1). It can be placed anywhere in method bodies where a condition is ought to be true.

Inheritance

In Jass, contracts from a supertype are not inherited by a subtype. Further, contracts cannot be specified for abstract methods, either in interfaces or abstract classes, since they do not have a method body and, hence, inheritance for these types is not possible.

However, Jass supports a mechanism called refinement which makes it supertype aware and, thus, aware of contracts in supertypes. By default, refinement is disabled, but when a subtype implements the marker interface jass.runtime.Refinement, contracts of the supertype are checked as well, in particular, it is ensured that a subtype may weaken pre-conditions but strengthens post-conditions (see 3.1, page 25). In the case of a pre-condition, Jass evaluates the expression $pre_{supertype} \wedge \neg pre_{subtype}$. If the result is *true*, a subtype does not fulfil a pre-condition the supertype fulfils and a refinement error occurs. While this check is done automatically, a subtype must provide an instance of the supertype for these checks. Therefore, a method jassGetSuperState, that returns an instance of the supertype, must be implemented. Although the subtype itself is an instance of its supertype, it might not be a sufficient super state, because the subtype cannot access private members of the supertype and might even hide those members [Bar99, Jas05].

3.2.3 JML - Java Modelling Language & Common JML Tools

The Java Modelling Language (JML) is a behavioural interface specification language which has been developed by Gary T. Leavens, who is a Professor of Computer Science at the Iowa State University. JML and the Common JML Tools bring DBC to Java. Note, that JML is a specification language only, and that the Common JML Tools implement a JML parser and compiler, thus, enabling specification-checks at runtime. Aside from the Common JML Tools, there are some other tools using JML as their input language [BCC$^+$05]. Yet, in this thesis, JML and the Common JML Tools are examined only.

The following shows a small selection of the JML feature-set, mainly chosen because other tools have similar features or because they are of importance for this thesis. A comprehensive guide to JML is the JML Reference Manual [CCC$^+$06].

Class-Invariants

JML knows of class-invariants and a related construct called history constraints. Both define properties which hold true in all *visible states* of a program execution. The definition of visible states is made up from the following statements [CCC$^+$06]. Have an object o, so all visible states are:

- the end of any constructor initialising o,
- the beginning of the finaliser which removes o from the memory,
- the beginning or the end of all methods defined in o or inherited to o.

An exception from these rules are all methods (including constructors) which are marked as *helper* using a helper-annotation.

Class-invariants and history constraints can be differentiated by looking at the properties they define. A class-invarinat may define a boolean expression which is ought to be true in all visible states. In contrast, a history constraint targets post-states only but can access values from the pre-state. The example in Listing 3.9 shows an invariant, that asserts that a buffer is never less than zero, and a history-constraint, that uses the old-construct to ensure that the size of a buffer grows monotone. Further, the method skipMe is marked as helper which excludes it from invariant checks.

```
public class Buffer {

  //@ invariant fBufferSize >= 0;
  private int fBufferSize;

  //@ constraint data.length >= \old(data.length)
  private Object[] data;

  public /* helper */ void skipMe(){ ... }

  public void checkMe(){ ... }
}
```

Listing 3.9: Invariants and history constrains in JML.

Additionally, a visibility modifier may be added to invariants and history constraints. The visibility modifiers have the same semantics as those known from Java, and inheritance of contracts is controlled with them.

Method Specifications

In JML the specification of a method consists of a number of specification cases, each being a behavioural specification. Essentially, each method specification consists of:

- A visibility modifier expressing that a method specification is public, protected, package_private, or private. The semantics of these modifiers equal those known from Java.
- A pre-condition and two different types of post-conditions, namely, normal and exceptional post-conditions.
- An assignable clause which is used to denote members which value can be modified safely in the course of the method execution.
- Further, aspects like concurrency, timing-issues, and redundancy are optionally addressed by JMLs method specifications. See chapter 9.9 in [CCC$^+$06] for further information.

An example of a method specification, which defines normal and exceptional behaviour, is given is Listing 3.10. An exceptional post-condition is evaluated if the execution of a method is terminated due to an exception.

```
/*@ public normal_behaviour
  @   requires obj != null && !isFull();
  @   ensures size()-1 == \old(size()) && contains(obj);
  @ also
  @ public exceptional_behaviour
  @    requires obj == null;
  @    signales (NullPointerException) \old(size()) == size();
  @ */
public void add(Object obj){ ... }
```

Listing 3.10: A method specification in JML.

In Listing 3.10 both specification cases have the visibility modifier public. Note that there are two rules restricting the use of visibility modifiers in JML specifications.

- The visibility modifier cannot permit more access, than the method being specified does. Thus, a public method may have a private specification but it is not allowed to have a public specification with a private method.

- The visibility of a specification must also be validated against the elements referenced by that specification. For instance a public specification may not refer to a private member. Otherwise a specification is accessible from other classes (especially with inheritance) but the elements used in the specification are not.

Inheritance

In JML, specifications of supertypes are inherited to its subtypes no matter if the supertype is an interface or class type. The only restriction one has to obey is that method specifications that inherit specification cases from supertypes must start with the also-clause. Further, visibility of the supertype specifications matters, e.g. a private specification case will not be inherited to a subtype. In other words, the visibility-attribute of a JML specification has the same semantics as the visibility-modifier in Java.

```
public interface IBuffer {

  /*@ public normal_behaviour
    @     requires obj != null;
    @ */
  public void add(Object obj);
}

public class BufferImpl implements IBuffer {

  private int size;

  /*@ also
    @ private normal_behaviour
    @     ensures \old(size)+1 == size;
    @ */
  public void add(Object obj){ ... }
}
```

Listing 3.11: Inheritance and refinement of specifications in JML.

In Listing 3.11, a concrete class and the interface it implements is shown. The class inherits a specification case for the method add and extends the specification with a second specification case. Note that a third class, extending BufferImpl, will not inherit the second specification because it is private. In such a case, only the specification of the interface, which is public, will be inherited.

Model Variables

JML does not allow that a specification refers to elements with a visibility that is more restrictive than its own visibility. For instance, the pre-condition in Listing 3.12, is not a valid JML specification because the public post-condition refers to a private member.

```
public class Buffer {
  private int size = 0;

  /* @ public normal_behaviour
     @     ensures \old(size)+1 == size;
     @ */
  public void add(Object obj){
    size += 1;
    ...
  }
}
```

Listing 3.12: Invalid post-condition – a public specification may not refer to a private member.

This restriction enforces a clean decoupling of implementation details and specification. In order to fix the specification in Listing 3.12, the following approaches exist.

- The visibility modifiers of the members referenced by a specification get adjusted. So, the field size in Listing 3.12 would be made public. However, making such a helper variable public enables foreign changes which can easily break the specification. Further, in the case of interfaces, Java does not allow to define non-constant fields and, thus, helper variables cannot be used.

- In JML specification-only visibility modifiers as spec_public exist. In the example above private and spec_public can be used in combination so that the field is protected for foreign access but can be used in specifications. However, for interfaces, this approach is not very valuable, because fields in interfaces must be constant.

- A third approach is to use auxiliary methods, e.g. getSize() (see Listing 3.12). Such a helper method must guarantee to be side-effect-free. Still, using helper methods only, may not be enough to thoroughly specify method behaviour.

To address these problems, JML defines model variables and model methods [CLSE05]. In the course of this thesis model variables are of interest only, so the they are put in focus. Still, model methods are analogue to model variables. Basically, a model variable is a specification-only variable which can be used with specifications exclusively. On top of solving above problems, model variables can be used to avoid under-specification. What under-specification means and how model variables may solve that problem is shown once the syntax of model variables have been introduced.

A model variable is declared using standard Java field declarations plus the model-keyword [GJSB05, CLSE05] . Listing 3.13 shows a model variable declaration and how the model variable is referenced by an invariant.

```
public interface IBuffer {

  /*@ public model Object[] data; */

  /*@ invariant data.length >= 0 */

  public int getSize();
}
```

Listing 3.13: A model variable defined in JML.

In the next step a value representation is linked to the model variable. This is done with the represents-clause as shown in Listing 3.14. It shows a class implementing the IBuffer-interface and, thus, inheriting the model variable data. The model variable representation in line 5 shows that the model variable data is *represented by* the function fStorage.toArray. In particular, a model variables is not getting assigned with a value but relates to a model variable representation which evaluates to a value.

```
1 public class Buffer implements IBuffer {
2
3   private Collection fStorage;
4
5   /*@ represents data <- fStorage.toArray(); */
6
7   ...
8 }
```

Listing 3.14: Assigning a value to a model variable.

More about the syntax of represents-clauses and model variables/methods can be found in [CCC$^+$06], chapter 7 and 8.4.

Above section has already mentioned the problem of under-specification and that model variables might be a solution. For instance, using the model variable data (see Listing 3.13) makes it easy to define a post-condition for the getSize-method, Listing 3.15.

```
...
/*@ ensures \result == data.length */
public int getSize();
...
```

Listing 3.15: A post-condition using the model variable.

Without the knowledge about a specification-only representation of the data storage, such a specification could have not been expressed.

3.2.4 jContractor

The jContractor tool ushers design by contract in Java by a principle based on the separation of code and contracts, the adherence to naming conventions, and bytecode manipulation. The jContractor project was initially launched by Murat Karaorman, Urs Hölzle, and John Bruno in 1998 [KHB98]. These days the project is in the ownership of students of the University of California, Santa Barbara.

The essence of jContractor is to implement contracts as methods returning a boolean and, thus, expressing if a contract holds or not. The binding between contracts and code is based on naming patterns. The jContrator feature set includes class invariants, pre-, and post-conditions.

Class-invariants

JContractor relies on contract-methods and a naming-patterns. In the case of class-invariants, the name of such a contract-method needs to be _Invariant. As a consequence of this and due to the fixed method-signature, all invariants need to be defined in this single contract-method (see Listing 3.16).

```
protected boolean _Invariant(){
  return invariant_1 && invariant_2 && ... && invariant_N;
}
```

Listing 3.16: A method implementing an invariant.

Method assertions

The name of a method implementing a pre- or post-condition can be derived by adhering to the rules defined in table 3.1. Just as for invariants, all pre-conditions and post-conditions for a method need to implemented in a single method. In Listing 3.17 for instance, a method is shown which implements a post-condition ensuring that the return value is not the null reference.

Method	Method-signature
Target	<*modifier*> **returnType** name(**parameters**)
Pre-Condition	protected **boolean** name_**Precondition**(**parameters**)
Post-Condition	protected **boolean** name_**Postcondition**(**parameters**, **returnType RESULT**)
Exception-Handler	protected **returnType** name_**OnException**(**parameters, Exception**$_i$ **e**) throws **Exception**$_i$

Table 3.1: Signatures for contract-methods in jContractor.

```
protected boolean get_Postcondition(int i, Object RESULT){
  return RESULT != null;
}

/** Target for the post-condition. */
public void get(int i){
  return data[i];
}
```

Listing 3.17: A method explicitly implementing a post-condition.

Notes

The declaration of contract methods and ordinary methods must not be separated into different classes. Nevertheless, it is recommended to comply with a clean separation of contracts and code, additionally, contracts for interfaces cannot be specified differently.

There is a tool called C4J (Contract for Java) which implements design by contract in a similar fashion, namely by implementing contracts as methods and defined a special naming-scheme. An interesting facet of C4J is that, in addition to the naming patterns, binding relies on a Java 5 annotation [Ber06]. However, as there is less literature and information about C4J, jContractor has been chosen for this work.

3.2.5 Contract4J

Contract4J enables design by contract in Java with assertions written as Java 5 annotations. Assertions are evaluated using aspect oriented programming techniques and a dynamic Java evaluation engine [Wam06a, Wam06b]. This approaches contrasts with the previously introduced tools as assertions are not validated or compiled but interpreted at runtime. Contract4J is featured by the aspect research associates and can be spilt up into Contract4J5 and Contract4J-Beans [Asp06]. Since the latter one is merely a prototype and not covered in this work, the terms Contract4J5 and Contract4J are used likewise. Contract4J offers three annotation types: pre-, post-conditions, and class-invariants.

Class-invariant

A class-invariant is expressed using the @Invar-annotation which defines two elements. Both elements are Strings and they are used to set a boolean expression and a message. The boolean expression has to evaluate to a boolean value, whereas the latter one is a custom message in case this invariant is violated. The @Invar-annotation can be placed at almost any available annotation-target, so that invariants can be placed at type-definitions, constructors, methods, fields, and annotation-types. However, due to the annotation design (see 2.1.2), every target can be annotated only once with the @Invar-annotation. An example of class-invariants in Contract4J is given in Listing 3.18.

```
@Invar("fBackupBuffer != null")
public class Buffer {

  private Object[] fBackupBuffer = new Object[]{};

  @Invar(value="fBufferSize >= 0", message="Illegal buffer-size.")
  private int fBufferSize = 0;
}
```

Listing 3.18: Invariant in Contract4J.

Method Specifications

The @Pre-annotation implements the pre-condition feature in Contract4J and can be applied to constructors and methods. Its default value is a string which is meant to be

Java code evaluating to a boolean. The example code in Listing 3.19 shows the pre-condition which uses the special keyword $this. It is equivalent to the Java this-keyword but must be used because of the evaluation engine which is not capable of handling this.

```
@Pre("item != null && $this.isFull() == false")
public void add(Object item) { ... }
```

Listing 3.19: Pre-condition in Contract4J.

The post-condition feature resembles the pre-condition except for the additional $result-keyword and the $old()-function. The latter one is used to access the state of the passed type whilst $result references the return value. The post-condition and both constructs are shown in Listing 3.20.

```
@Post("$result != null && $old(fBufferSize) - 1 == fBufferSize")
public Object removeLast(){ ... }
```

Listing 3.20: Post-condition in Contract4J.

Inheritance

Contract4J has limited support for inheritance of specification from annotated supertypes to subtypes [Asp06]. In particular, Contract4J has to cope with the fact that Java 5 annotations, and thereby its specifications, are not inherited (see 2.1.2, page 14). The exception from this restriction, class-type annotations and the Inherited-meta-annotation, make Contract4J inherit the invariant which possibly has been placed at the type definition.

Apart from this difficulties, inheritance in Contract4J bases on the Liskov Substitution Principle (LSP) which states that a subtype has to be substitutable for his parent type without breaking the program. Put in other words, if a certain property holds true for a type T than it holds true for a type S which is a subtype of T [LW93]. As a consequence, Contract4J must obey the contracts of a parent type when subtypes exist. In the case of pre-conditions, this means that an overwriting method may weaken the pre-condition so that in any case the overwritten pre-condition is satisfied by the subtype's pre-condition. In opposition, post-conditions may be strengthened as the overwritting method can narrow down the set of possible results. Back to Contract4J, inheritance is implemented with the following rules and restrictions:

- The only invariant that possibly gets inherited is the one placed at the type-definition. However, it is meant to be immutable so that new invariants in subtypes can be placed at fields, constructors, and methods only.
- For methods, inheritance must be enabled manually by adding the annotations @Pre and @Post to the overwriting method. However, if an overwriting method wants to refine the specification it inherits, it is obligated to repeat the inherited contract and to connect it with the logical and (post-condition), or or (pre-condition)

respectively. This restriction makes refinement and inheritance of specifications antagonists and one might say that Contract4J supports inheritance of specifications only if they are not getting refined.

Listing 3.21 shows how inheritance in Contract4J works and how refining specifications thwarts inheritance.

```
 1  public interface IBuffer {
 2
 3    @Pre("obj != null")
 4    public void add(Object obj);
 5  }
 6
 7  public class EndlessBuffer implements IBuffer {
 8
 9    @Pre
10    public void add(Object obj){ ... }
11  }
12
13  public class RestrictedBuffer implements IBuffer {
14
15    @Pre("!this.isFull() || obj != null")
16    public void add(Object obj){ ... }
17  }
```

Listing 3.21: Inheritance and refinement in Contract4J.

The pre-condition defined in line 3 is inherited to both implementation of the interface IBuffer. Since the EndlessBuffer does not refine it, adding the empty @Pre-annotation is sufficient for contract inheritance (line 9). In contrast, the RestrictedBuffer refines the pre-condition and, thus, needs to re-type the parent specification. Line 15 shows the refined pre-condition.

Notes

Contract4J uses Java 5 annotations to specify Java expressions, each representing a boolean value, which will be evaluated at runtime. Evaluation happens dynamically at runtime and, thus, compilation is not required. However, the dynamic contract evaluation brings two drawbacks. First, specification errors, misspelled variable names for instance, are detected at runtime only and, secondly, the evaluation engine used in Contract4J does not offer full Java syntax support.

3.2.6 Using Aspect-Oriented Programming

This section does not introduce a concrete nor existent design by contract implementation but sketches up how a DBC-tool for Java could look like if implemented with aspect-oriented programming techniques. The basic ideas of this approach are borrowed form Filippo Diotalev who has shown how design by contract and aspect-oriented programming for Java could be paired up [Dio04]. Before going into more detail, aspect-oriented programming will be introduced briefly.

Aspect-oriented programming

Aspect-oriented programming (AOP) was introduced in the late 90s by a group of researchers at the Xero Palo Alto Research Center [KLM+97]. It is a programming paradigm which emphases the separation of *cross-cutting* concerns. Cross-cutting concerns are a specialisation of the basic idea of separation of concerns which is used to aid programmers to define well modularised programs. In procedural and object-oriented programming the separation of concerns works well for structural properties as they have entities like packages, classes, and methods. However, some concerns like logging or security management cannot be encapsulated in separate entities and they *cut across* every module in a program. For instance, in a business application almost every critical method will start and end with a logging statement.

The goal of AOP is to capture such cross-cutting concerns so that they can be encapsulated into separate entities. Such entities are usually called aspects. Along with aspects the following terms are used in the scope of aspect-oriented programming:

Aspect An aspect is an encapsulation of cross-cutting concerns in programming. An example is logging which cuts across all modules, classes, and methods of a program.

Advise Advises are provided by aspects and define additional behaviour at certain points in a program. Realtions like before, after, or around express when the advise is executed relative to these points. Sticking to the logging example, an advise would be to log a message before a certain point in a program is reached.

Join Point A join point is a point in a program where an advise can possibly be applied, in other words, where the advise can join the program. Examples of join points are methods and constructors.

Pointcut The role of a pointcut is to select a set of join points for a certain advise in an aspect. For logging, a pointcut could select a method based on its name and defining type.

Note that these term definitions base on the AspectJ project and might differ sligthy in other projects[2]. However, AspectJ which emerged from the original project at the Palo Alto Research Center is the most prominent aspect-oriented programming language for Java and can be counted as a reference for AOP [The06]. An example of AspectJ is given in Listing 3.22 wrapping up the previously introduced logging example. The goal is to log all method calls where the method is declared as private.

```
aspect LoggingPrivates {

    pointcut callPrivate() :  call(private * T.*(..));

```

[2] For instance naming can be different in other projects. This is similar to procedural programming and the notion of methods, functions, or procedures.

```
5      before(): callPrivate() {
6          System.err.println("about to call a private method");
7      }
8  }
```

Listing 3.22: An unsophisticated logging aspect written in AspectJ.

In line 1 the aspect LoggingPrivates is defined which encapsulates a pointcut and an advise. The pointcut-definition in line 3 consists of the pointcut's name callPrivate() and its capture; all calls of private methods in type T regardless their return type, name, and parameters. The before-advice (line 5 to 7) will be executed prior to all method executions, which are matched by the callPrivate() join point, and simply prints a message to the standard error stream.

An obvious application of aspect-oriented programming is design by contract and contract enforcement. Since contracts are all over the code they are clearly cross-cutting and can be implemented with AOP. The next section will show how a design by contract implementing aspect could look like.

An aspect for method assertions and class invariants

This section shows how method assertions and class-invariants might be implemented with aspect-oriented programming. The example at this place was kept deliberately simple and was modelled after a more sophisticated version from [Dio04].

The code in Listing 3.23 shows an aspect which models pre-, post-conditions, and class-invarinats for a method Buffer.add(Object). The invariant ensures that the size of the buffer is always larger than or equal to zero (line 11 & 25), whereas pre- and post-condition ensure that the element to add is not the null-reference nor that the buffer is full already (line 14) and, finally, that the element actually resides in the buffer (line 21). The invariant, pre-, and post-condition are encapsulated in an around-advise which can be applied to all join points captured by the pointcut contractTarget(). That pointcut is defined in line 3 and will select the method add(Object):void of the type Buffer regardless its visibility.

```
1   aspect BufferAddAspect {
2
3     pointcut contractTarget() : call(void Buffer.add(Object));
4
5     around() : contractTarget() {
6       // (1) capture target object and join point arguments
7       Object obj = thisJoinPoint.getTarget();
8       Object[] params = thisJoinPoint.getArgs();
9
10      // (2) check invariant
11      assert ((Buffer)obj).fSize >= 0;
12
13      // (3) check pre-condition
```

```
    assert param[0] == null || !((Buffer)obj).isFull() :
        "Pre-condition violated - cannot add null.";

    // (4) proceed with method execution
    proceed();

    // (5) check post-condition and invariants
    assert !((Buffer)obj).contains(param[0]) :
        "Post-condition violated - Adding element failed.";

    // (6) check invariant
    assert ((Buffer)obj).fSize >= 0;
  }
}
```

Listing 3.23: Method assertions and class-invariant for a buffer.

Applying the aspect as it can be seen in Listing 3.23 to a Java program will enforce the defined contracts for the type Buffer. Even though topics like inheritance, the notion of *pure*, or exceptional behaviour are not covered in this example, aspect-oriented programming has proven to be a candidate for implementing design by contract.

3.2.7 Using the Proxy-Pattern

Similar to the section above, a further approach to implement design by contract for Java will be outlined. It is based on an article and a prototypic implementation by Anders Eliasson [Eli02] who proposes a pre-processor based implementation which uses the proxy-pattern. In object oriented programming, the proxy-pattern defines a proxy-object which appears as a surrogate of the actual object. This gives the proxy-object the chance to perform additional operations every time the actual object is accessed e.g. checking a pre-condition before calling a method. Figure 3.1 visualises the proxy pattern and how it can be used to implement design by contract as an UML sequence diagram.

Technically, the proxy pattern approach proposes to create a *Contract*-proxy for every object in the virtual machine or for those that have contracts. Since a proxy-object stands-in for the actual object and implements its contracts, contract enforcement is guaranteed.

Class-invariants & method assertions

The previous section embraced the technical part of [Eli02] only and did not define how contracts are specified. In this DBC-implementation, class-invariants and method assertions are ought to be defined via custom javadoc-tags. Those are @pre, @post, and @invariant, presented in Listing 3.24, which need to be processed by a doclet generating appropriate code.

```
interface Buffer {
  /**
   * @invariant size > 0
   */
```

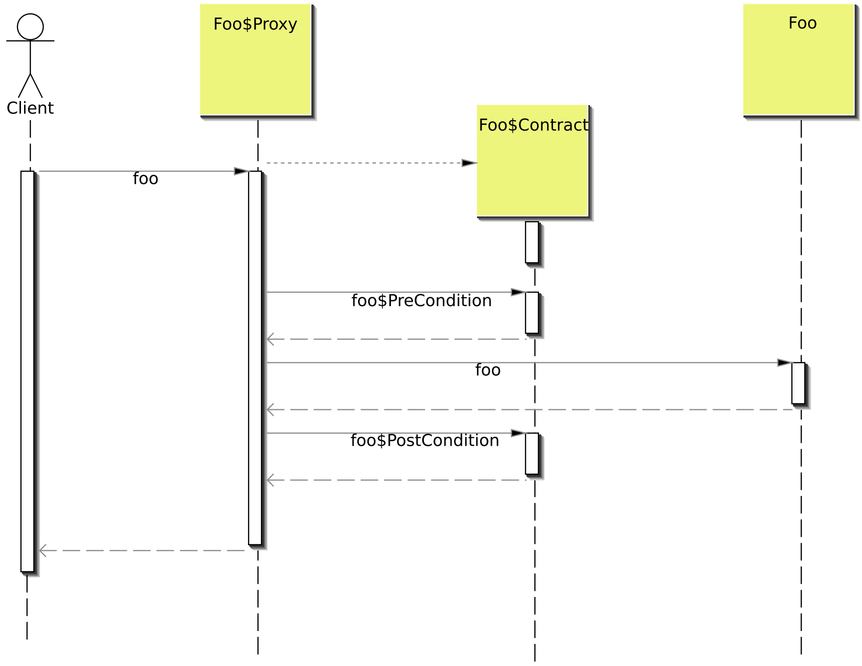


Figure 3.1: The proxy-object Foo$Proxy stands-in for Foo and checks pre- and post-conditions for method bar.

```
  int size;

  /**
   * @pre o != null
   * @post contains(o)
   */
  void add(Object o);

  boolean contains(Object o);
}
```

Listing 3.24: Using custom javadoc-tags to specify assertions.

For instance, a doclet may create the class Buffer$Contract, Listing 3.25, from the Javadoc-tags in Listing 3.24. For every contract-tag a method has been created which implements that contract. Later, reflective programming can be used to discover and invoke these contract methods.

```
class Buffer$Contract{

  boolean Buffer_invariant(){
    return size > 0;
  }

  boolean add_pre(Object o){
    return o != null;
  }

  ...
}
```

Listing 3.25: A *contract*-class created by a doclet.

However, when using the proxy-pattern one is obligated to manually create the proxy instances which might pollute the source code and makes it harder to remove DBC checks. Further, Java's native proxy support is constrained by the availability of interfaces. In Java proxy object can only be created for those objects that implement at least one interface. Consequently, if a type implements no interfaces, Java's proxy mechanism cannot create a proxy object for it [Suna].

3.3 Notes on Design by Contract in Java

At its heard, every DBC-implementation wants to avoid that contract checks have to be implemented manually and aims for a clean separation of code and contracts. The advantages of this separation are outlined below:

- Separating contracts and code makes contracts clearly identifiable. For instance, this might ease contract-inheritance or help contract-specific documentation tools.
- A DBC-tool which connects contracts and code, minimises boilerplate code as it can take care of recursive contract checks and contract violations. Reporting a contract violation, as one can see in line 3, Listing 3.26, would be done by a DBC-tool, keeping the actual code simpler.
- Usually, DBC is applied during the development phase so that programming errors can be detected early. To avoid runtime penalties, DBC-checks are removed or deactivated in productive use which is a call for a separation of contracts and code.

```
1 public void add(Object o){
2   if(!(o != null && !full())
3     throw new PreConditionException();
4
5     //...
6 }
```

Listing 3.26: Manually implementing a pre-condition is not desirable.

Usually, the separation of code and contracts is achieved by having two parts in a DBC-implementation. First, contracts need to be specified and somehow attached to source code and, secondly, it must be ensured that these contracts hold during runtime. In terms of software architecture, the phrases *front-end* and *back-end* can be used. Sections 3.3.1 and 3.3.2 classify and characterise different front- and back-ends which are taken from the tools introduced in 3.2.

3.3.1 Front-ends

In a DBC-implementation, the front-end constitutes how contracts are specified. Basically, two different approaches exist. A contract is specified as some kind of annotation which is subject to further transformation (see 2.1) or contracts are defined as external source code which is somehow bound to the actual source.

Java Comments Using Java comments, line- or block-comments, is the most flexible annotation-based specification mechanism. However, this flexibility comes with the cost of implementing and maintaining a parser for Java source code just to discover those comments that represent contracts. In addition, the validity of contracts, e.g if a field referenced in a contract actually exists, must be checked.

Javadoc-tags When using Javadoc-tags and a doclet, there is no need for a parser since the doclet API can be used to retrieve the content of the Javadoc-tag and, thus, the contracts. However, a the validity of contracts is not guaranteed and might require some coding effort.

Java 5 Annotations Analogue to Javadoc-tags and doclest, Java 5 Annotations can be processed by an annotation processor which saves the need for a special parser, but, alike doclets, does not guarantee that a contract is valid. However, when using a JSR 269 compliant annotation processor a lot information about the structure and members of annotated types is provided. Hence, contract validity-checks required less coding effort when an annotation processor is used.

Source Code Specifying contracts as source code is the most straightforward approach and minimises coding effort when implementing DBC. Contracts are checked for validity by the compiler and the only thing left to do, is to connect the contract source code with the actual source code. However, from the DBC-user's point of view, it might be harder to relate contracts and code when they are spread over different source files.

Aspects Using an aspect to represent contracts is actually very similar to representing contracts as source code. The difference is the fact, that an aspect is linked to its target differently. This enables more abstract architectures to actually check contracts (see 3.2.6 and [Dio04]).

3.3.2 Back-ends

In terms of a DBC-implementation, a *back-end* is a technical component which describes how contracts can be checked and how violations of contracts can be handled. The following will outline 5 different approaches to implement such a back-end.

Pre-Processor: A pre-processor based approach generates new source code, based on annotations present on the original source code. After that, the newly generated

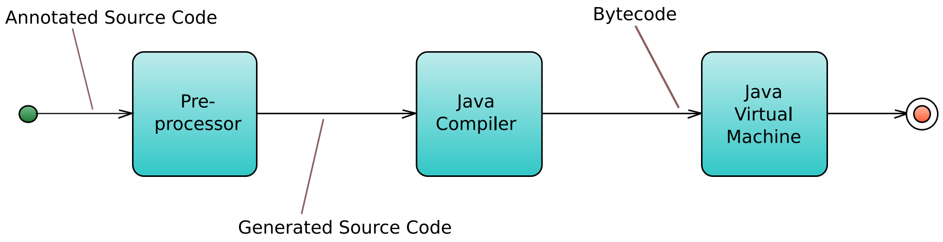


Figure 3.2: A pre-processor based approach.

source code can be compiled and executed. This process is shown in Figure 3.2. A pre-processor might be a free-standing program, a doclet, or an annotation processor, however, they do have in common that they generate new source files which are compiled and executed instead of the original, annotated source files. There are pros and cons that recommend and, respectively, discourage the use of pre-processors.

- The pre-processor architecture is pretty straightforward and easy to understand. If a doclet (2.1.1) or an annotation processor (2.1.3) is developed some elaborated APIs can be used and programming effort is kept at a minimum. However, if these APIs are not used, some development and maintenance effort has to be undertaken as Java source code needs to parsed and processed. Especially, the maintenance costs have shown to be a major drawback.
- Usually, a pre-processor adds methods to existent types and / or code to existent methods. This might hinder debuggers since the code executed, differs from the code being present in the IDE and confuses programmers because code appears on the screen they have never seen before. Further using a pre-processor might mess up stack traces as methods appear which the programmer did not write.

Bytecode Instrumentation: Usually, bytecode instrumentation is used to connect a class with its contract. This implies, that the contracts are available as bytecode at the time the instrumentation is performed. JContractor (3.2.4) adheres to this pattern as it adds invocations of *contract*-methods to the beginning of a method. These contract methods are selected by pattern-based matching process (e.g. a *pre-condition*-method is named *pre*_methodName).

Bytecode instrumentation can be performed statically or dynamically. The latter one can be implemented with a custom classloader or the instrumentation API (2.2.3), introduced in Java 5.

The following facts should be considered, when thinking of bytecode instrumentation to implement contract checks:

- Using bytecode instrumentation requires that the contracts are available as bytecode. Thus, contracts, specified as comments or annotations (3.3.1), need to be transformed into bytecode first.

- Bytecode instrumentation is very powerful, so that it is possible to keep stack traces clean and have valid debugging information like line numbers.

Compiler Developing a custom compiler for a DBC-tool is the most challenging task. It involves writing a parser for the extended Java syntax and a bytecode generation facility. Further, this implementation approach has high maintenance costs as the Java programming language and the Java bytecode evolves. The JML Common Tools use the compiler approach and suffer from the maintenance cost, as they are still not fully capable of processing Java 5 syntax and, even worst, recommend for a long period to uninstall Java 5 when using the JML Common Tools.

Proxy Pattern The proxy pattern is a valuable DBC-implementation which is characterised by the fact that proxies have to be instantiated manually and that contracts must be implemented in, or referenced by these proxies. The latter on might require a pre-processor, which creates contract-code from a specification. One should consider that:

- Although proxies can be created natively in Java, the instantiation is not done automatically. Further, the creation of proxies depends on the availability of interfaces (3.2.7) and executable contracts.

Aspect-Oriented Programming The cross-cutting concerns addressed by aspect-oriented programming matches the concept of DBC and, thus, an application for DBC is evident. Nevertheless, when using AOP one should keep in mind, that it is not only a different programming paradigm, but also a different programming language. Although most AOP-implementations base on Java, they do have a different syntax and require other compilers or load-time weavers.

3.3.3 The Semantics of 'Old'

As seen in the previous sections, all design by contract implementations for Java have an Old-construct. However, the semantics of Old alters, as for complex types state-capturing may not be trivial. In general, the semantic of Old is to store the state of an object or primitive type in the pre-state, so that it can be used for further operations, e.g. comparing an 'old' state with the current.

A straight forward approach to implement Old is to simply copy the target to a separate location in memory and keep it there unmodified. In the case of primitive types, this copy approach works fine as Java supports copying of primitive types natively. For complex types, however, only the reference would be copied and, thus, modification of the pre-state cannot be prevented. There are three ways to deal with this:

- A DBC implementation may simply deal with it and advises users accordingly. Especially, expressions like Old(buffer) == buffer && Old(buffer.size()) != Old(buffer).size() might be true in this case.

- User might be encouraged to make use of the clone-method defined in java.lang.Object, which returns and creates a copy of the calling object [Sund]. However, using clone() is not that easy, because, by default, it triggers a CloneNotSupportedException. If an object wants to become cloneable, it has to implement the marker-interface java.lang.Cloneable which signals to the method clone() that it can be executed [Sund]. Alternatively, one can also overwrite the java.lang.Object.clone()-implementation and create a custom clone.

- A third alternative is to use immutable classes only. An immutable class, java.lang.String for example, cannot change its state once it is created. Every operation which seems to be state changing, actually returns a new instance of that class. Consequently, the reference of such an immutable object can safely be stored and used for state comparisons as it may not change.

In practise, not all classes are immutable, implement java.lang.Cloneable, or overwrite clone() with the effect that the design by contract implementations have different semantics for the Old-construct. Table 3.2 lists the different semantics of the introduced DBC implementations. The column *reference* stands for copying the reference of an object and *clone* imposes the use of java.lang.Cloneable. Only Jass enforces the use of clone when the Old-construct is used; all other tools simply store references. Note that the latter approaches do not exclude the use of clone, it makes it a user specific decision.

Tool	reference	clone
Jass	–	●
JML	●	–
AOP	●	–
Contract4J	●	–
jContract	●	–

Table 3.2: Overview of Old and its semantics.

3.3.4 Overview

Table 3.3 summarises the main features and properties of the DBC-tools introduced in this chapter. The last two introduced approaches ‘Using Aspect Oriented Programming’ (Section 3.2.6) and ‘Using the Proxy-Pattern’ (Section 3.2.7) are not included in this table because they are merely prototypes or concepts and not deployed tools.

From the table it can be seen that the Java Modelling Language (JML), which has been introduced partially only, is by far the most powerful DBC-tool for Java. Jass offers a rich set of assertion kinds but has limitations when it comes to inheritance and it does not support model variables. Contract4J and jContractor are less powerful than JML or Jass as they offer no or very limited inheritance, do not support model variables, and support less assertion kinds (e.g. no exceptional post-conditions or loop-invariants).

	Jass	Common JML Tools	jContractor	Contract4J
Invariants	●	●, including history constrains	●	●
Pre- / Post-Conditions	●	●, including exceptional behaviour	●, including exceptional behaviour	●
Return	●	●	●, variable RESULT in contract method	●
Old	●, depends on clone & java.lang.Cloneable	●, stores references	●, stores references	●, stores references
Quantifier	●	●	●	–
Loop-Invariants	●	●	–	–
Model-Variables	–	●	–	–
Inheritance	–, refinement only. Not for interfaces.	●	–	●, very limited
Contract Validation	Custom parser	Custom parser	Java Compiler	–, evaluation engine at runtime only
Input Method	Comment-based	Comment-based	Java Source Code and methods	Java 5 Annotations
Back-End	Pre-Processor	Compiler	Bytecode-Instrumentation	Aspect Oriented Programming

Table 3.3: Overview of the Design by Contract tools.

Chapter 4

Design Decisions

This chapter reflects the process of creating a new design by contract implementation for Java matching up with the goals outlined in the first chapter of this thesis. Essentially, three major goals can be identified for the new design by contract implementation:

- Using Java 5 annotations (see Section 2.1.2, page 10) to add contracts to Java code, and utilising the annotation processing facilities to process such contract annotations.

- Focusing on non-functional features like seamless integration, so that writing contracts and running contract-protected code is made easier for users. In particular, invalid contracts are to be detected during compilation, and failing contracts must throw meaningful exceptions, especially regarding its origin.

- Using APIs and tools delivered with the Java standard platform (J2SE) and, thus, being easy to integrate into new and existent Java projects.

The purpose of this chapter is to answer the *Why* and *How* questions regarding these goals. For instance such questions are *'Why use Java 5 Annotations?'* or *'How to create meaningful errors?'*

In addition to the discussion on design decisions, the desired feature set of the new design by contract tool is defined and sorted by priority (Section 4.4). Last but not least, the newly designed DBC-tool gets a name (Section 4.5).

4.1 Java 5 Annotations & Annotation Processing

In this section the usage of Java 5 annotations is discussed and justified. It is to make clear why other approaches have been rejected in favour of annotations. The alternatives have been introduced in the previous chapter (Section 3.3.1 at page 45) and, shortly listed, are any kind of comment, plain source code, and aspects. Using Java 5 annotations has two drawbacks causing specifications to be more verbose and less expressive, because not all elements of a program can be annotated. The following gives an idea how these things put a damper on using annotations.

- As seen in Section 2.1.2, Java 5 annotations have a limited set of targets, meaning they can only be added to methods, classes, and variable declarations. Consequently, specifications like loop-invariants cannot be expressed with Java 5 annotations. Still, with the advent of JSR 308, annotations are going to be allowed on such program elements, and an annotation based implementation could easily be extended.

- The type of attributes in annotations is almost restricted to primitives, and their values must be compile time constants. As a consequence the most expressive attribute type is the String which may represent an arbitrary specification clause. However, when using plain Strings further parsing will be required and characters, that have a special meaning in Java, must be escaped. Parsing the values of annotations, makes this approach not differ significantly from comment-based specifications.

On the other side, using Java 5 annotations has some major advantages coming from the fact that they are pure Java, equipped with a rich processing environment. The following will outline these advantages:

- The purpose of annotations is to add features or constructs to Java which are not available by default. Hence, using Java 5 annotations should naturally be the first choice when design by contract is to be implemented for Java.

- Because annotations are standard Java, common processing tools and utilities are available. The javadoc-tool, for instance, includes the annotations of an element in its HTML documentation by default. Further, most IDEs offer functions like auto-completion or content proposals for annotations.

- In contrast to comments, annotations do have a bytecode representation making them accessible from raw bytecode and during runtime. Consequently, annotations can be processed by an arbitrary back-end and do not depend on a pre-processor which extracts the contract information from comments.

- The Java standard platform is equipped with a rich processing environment which is integrated into the Java compiler making annotation processing part of the compilation process. This is the basis for a true language extension, and guarantees a seamless integration into different build processes and tools.

In conclusion, using Java 5 annotation turns out to be the most suitable approach for design by contract. Even though some limitations exist, the advantages outweigh them easily. Especially the fact that annotations are standard Java, that they are undergoing further development (see JSR 308, Section 2.1.2), and that a processing environment exists, promotes the usage of Java 5 annotations strongly.

4.2 Validating Contracts

In contrast to the previous section which justified the requirement of using Java 5 annotations, this section reflects the discussion about finding a way to validate contracts. Validating contracts is not to be mistaken with checking contracts and can be explained with the following code snippet:

```
@SpecCase(
   pre = "name == 123",
   post = "@Result != null")
public void foo(String name){
   ...
}
```

Both, pre- and post-condition are invalid as, for the first, the pre-condition tries to compare a string with an integer and, for the second, the post-condition refers to the return value of a void method. Validating contracts is to programmatically find semantically and syntactically errors in contracts. Usually, contract validation is to ensure that a succeeding processing step, e.g. source- or bytecode generation, can be performed safely and produces valid results. From the DBC-tools examination in Section 3.2 three different approaches for contract validation have been extracted. Those are:

- Having a custom parser which validates the syntax of contracts and, equipped with context information, validates the semantics is the obvious approach for contract validation. By context information, type information is meant. For instance, to validate the pre-condition @Pre("debit != -123"), the parameter debit must be known to be an integer variable.

  Usually, a custom parser is used when the contract is specified in a Java comment, e.g. *Jass* (Section 3.2.2) and the *Common JML Tools* (Section 3.2.3) use a custom parser. The advantages and disadvantages of a custom parser are discussed in Section 3.3.2 but in almost every case high maintenance costs are to be expected.

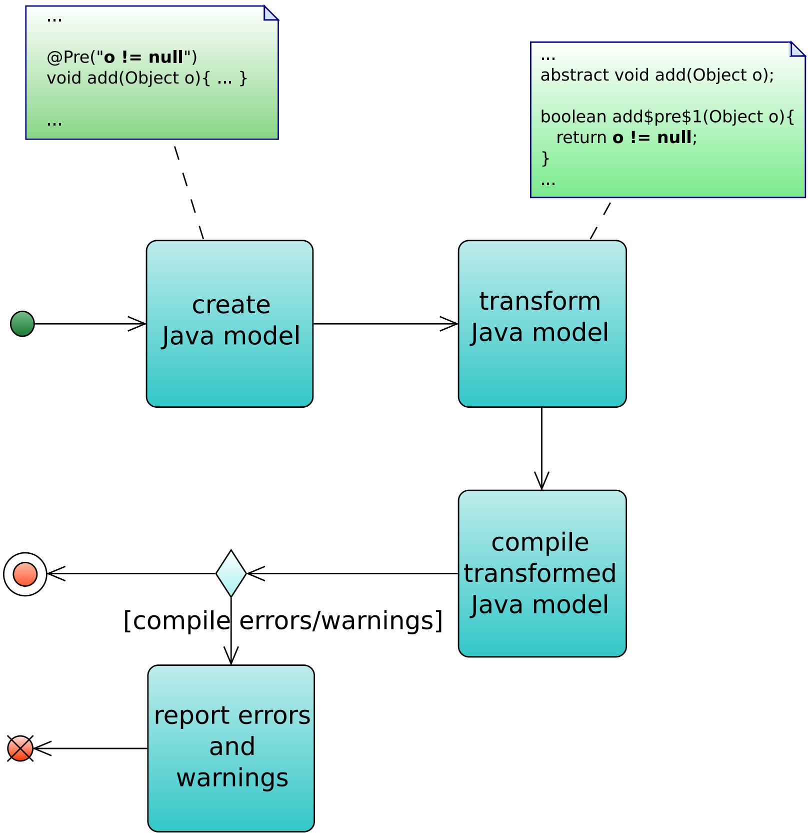
- The *jContractor*-tool (Section 3.2.4) utilises the Java compiler for contract validation because contracts are expressed as method bodies and, hence, targeted during the standard compilation process.

  Although being appealing, the prerequisite is that contract code is part of an element which is recognised and compiled by the Java compiler. Without any preprocessing this is not possible for contracts being embedded in comments or Java 5 annotations.

- When contracts are interpreted on-the-fly by an evaluation engine, no source- or bytecode needs to be generated and contract validation might be optional. *Contract4J* (Section 3.2.5), for instance, uses such an approach and does not validate contracts before they get executed. However, this might lead to an increasing number of failing contract executions due to invalid contract code.

Altogether, every approach has its downside and a more sophisticated way to validate contracts is desired. The Java compiler is capable of reading source code and to output bytecode or errors, but the chosen contract representation, Java 5 annotations, is only partially processed by the compiler. Annotations are translated into bytecode but the content of attribute values is only checked for its type, not validated to be a sound assertion. To give an example, the Java compiler does not care if an attribute value refers to a non-existent method parameter because it does not know the semantics of that particular attribute. The goal is to make the Java compiler aware of these semantics and to report violations.

Getting the Java compiler to validate contracts is achieved by using an annotation processor which performs some pre-processing and utilises the Java compiler API (see Section 2.2.2). The UML activity diagram in Figure 4.1 gives an overview about the necessary steps to check contracts with a combination of an annotation processor and the Java compiler.


```
...

@Pre("o != null")
void add(Object o){ ... }

...
```

```
...
abstract void add(Object o);

boolean add$pre$1(Object o){
   return o != null;
}
...
```
create Java model
transform Java model
compile transformed Java model
[compile errors/warnings]
report errors and warnings

Figure 4.1: UML activity diagram which shows the steps performed to validate a contract.

1. First, a model is created from the information provided by the annotation processing environment. It reflects the structure of the class currently being compiled, including type information, fields, method signatures, and annotations.

2. The model is modified so that every annotation attribute which represents a contract is transformed into the body of a so called *contract method.* Contract methods always return a boolean value, resulting from evaluating the contract code, and derive its signature and its name from the original method. For instance, the precondition of a method `void m(int a)` will be represented by a contract method with the signature `boolean m$pre$1(int a)`. Since the former annotation value is now a method body it is targeted and thereby validated by the Java compiler[1].

3. The transformed model is fed into the Java compiler using the compiler API (Section 2.2.2), and if a compile error or warning is generated, it is linked with the corresponding annotation and forwarded to the annotation processor.

   Note that the bytecode which is generated by the compiler is not used at this stage and simply ignored. For contract validation only error or warning messages are of interest.

Because annotation processors are plug-ins for the Java compiler and participate in the compilation process, they are a well-suited environment for contract validation. There is no external command required to start the contract validation and the Java infrastructure to report errors and warnings can be used. Further, delegating the actual contract validation to the Java compiler makes an external parser dispensable and, as an important consequence, when Java evolves, minimal maintenance cost will arise only.

4.3 Checking Contracts at Runtime

The previous section reviewed the design decision about finding a way to validate contracts. In this section, ways to check contracts at runtime are discussed. Checking contracts is to enforce that the conditions stated by the developer actually hold when the program runs and to abort the execution as soon as a contract is violated. Similar to validating contracts, the tools examinations in Section 3.2 showed different ways to check contracts.

- When using a pre-processor, as in *Jass*, code which checks the contracts is automatically created and executed instead of the original source code. Before the generated code can be executed, it must be compiled.

  The disadvantage of this approach is the fact that the code, that is getting executed, differs from the code that has been written by the programmer.

- The *jContractor*-tool uses bytecode instrumentation to weave contracts into program code. Because in *jContractor* specification are expressed as plain methods,

[1] The schemata from which contract methods are derived, are introduced in Section 6.2.

bytecode instrumentation is used to connect the actual code with the methods checking the contracts.

When performing bytecode instrumentation, two things are important. First, the bytecode for the contracts must be available and, secondly, the bytecode of a contract must be assigned to the right piece of program code. In *jContractor* assigning code with contracts is based on a naming pattern.

- A third alternative for checking contracts is a dynamic evaluation engine as it is used in *Contract4J*. This approach proposes to interpret contracts directly without further pre-processing. Depending on the result an error might be raised, due to a violated assertion, or execution continues.

For the new DBC-tool, bytecode instrumentation is a good candidate because Java agents (see Section 2.2.3) guarantee a seamless integration, and because the bytecode for contracts is a by-product of the contract validation, as discussed in the previous section. The overall process is very similar to the contract validation with the difference that the bytecode produced by the compiler is not ignored, but used for bytecode instrumentation. Figure 4.2 is an updated activity diagram showing that the bytecode is used for a further processing step. Besides, the entry point is a Java agent (Section 2.2.3 on page 22) instead of an annotation processor. The following steps are performed:

1. First, a model is created reflecting the structure of the class currently being instrumented. This model equals that one produced in step 1 of the contract validation but it is created from bytecode.

2. The model is transformed, so that contracts are represented as contract methods being targeted by the Java compiler. This processing step equals the one for contract validation.

3. Again, the transformed model is fed into the compiler and, if no compile errors occured, bytecode for each contract is created. This time, the bytecode is not ignored but kept for the next processing step.

4. Based on the name and signature of the contract method, the class currently being loaded is getting instrumented with the bytecode created in the previous steps (1-3). Now, the class is enriched with contract checks and it is getting loaded into the virtual machine.

Although contract validation and contract checks are very similar, one has to keep in mind that both are separate processes. Contract validation is performed during the development of contracts, analogue to compiling a program, and enabling contract checks is performed on-the-fly when classes are getting loaded into the virtual machine. Nevertheless, it is to question why the bytecode instrumentation is not performed by the annotation processor during the compilation phase, avoiding the expensive on-the-fly instrumentation. This is because the annotation processing environment does not allow to participate in the bytecode generation process. Annotation processors can access the

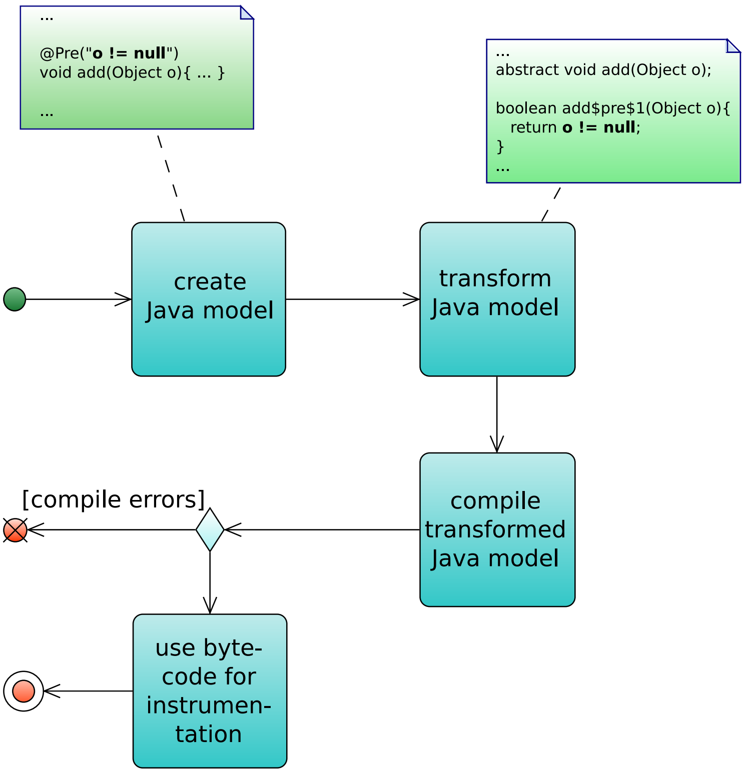

...
@Pre("o != null")
void add(Object o){ ... }
...
...
abstract void add(Object o);
boolean add$pre$1(Object o){
return o != null;
}
...
create Java model
transform Java model
compile transformed Java model
[compile errors]
use byte-code for instrumen-tation

Figure 4.2: UML activity diagram which shows the steps performed to enable contract checks at runtime.

structure of the program being compiled, may issue errors and warnings, and my create new files, but they cannot change the bytecode that is going to be output. Further, during compilation class files which are needed for bytecode instrumentation may not exist yet. Still, on-the-fly bytecode instrumentation can be speed up by saving the bytecode which is generated during contract validation and omitting steps 1 to 3 by simply re-using that bytecode.

4.4 The Feature Set

The goals outlined in the beginning of this chapter are non-functional and a list of functional features is not provided yet. Such a list is composed in this section containing mandatory and optional features. From the tools examination in Section 3.2 a minimal set of assertions and expressions can be extracted:

- *Invariants & Method assertions* – Class invariants, pre-, and normal post-conditions

- *Specification expressions* – Specification expressions to access the return value of method calls, and to access values from the pre-state in post-conditions

Above shows the most simplified feature set a DBC-tool may have, but to be more valuable, the following features are to be implemented in a DBC-tool:

- *Exceptional post-conditions* – Providing a mechanism which allows to specify method behaviour aside from normal termination by having exceptional post-conditions
- *User defined error messages* – Allowing user defined error messages for each assertion which is used in case the assertion does not hold
- *Side-effect freeness* – Supporting the notion of *pure* program elements which can safely be used in assertions because they are side-effect free
- *Quantifiers* – Enabling the use of quantifiers to express assertions that relate to a collection of elements
- *Inheritance* – Java compliant support for inheritance of assertions which means that assertions are inherited with respect to the standard visibility modifiers
- *Model Variables* – Having model variables similar to those that can be found in JML including rules for inheritance and representation of model variables
- *Lightweight Specifications* – Provide a way to ease writing specifications by offering a syntactical lightweight version of the assertions

4.5 Naming

Last but not least, a name for the new design by contract tool must be found. A lot of tools compose their names from the words *Contract*, *Design*, and *Java* like *Contract4J*, *C4J*, or *jContractor*, so that almost all possible combinations have been used already. The name *Jass* is derived from the phrase '*Ja*va with a*ss*ertions'. Since Jass has been developed at the same department where this master's thesis is written, a dependence on the name *Jass* is wanted. Jass and the newly design by contract tool implement DBC for Java. However, the tool developed in this thesis uses recent Java technologies so that the most suitable name seems to be:

Modern Jass

It reflects its heritage but also points out that its newer. Besides, *Modern Jass*, when pronouncing like the music, also denotes a music style that is related to Jazz but breaks away from its traditional roots. This is a nice analogy between the music styles and the DBC-tools.

Chapter 5

Specification Annotations

In this chapter the specification annotations of *Modern Jass* are introduced and explained. The purpose of this chapter is to give a precise definition of what *Modern Jass* annotations are, and how they are validated and evaluated. Tool developers should use this chapter as reference, when implementing a DBC-tool that uses specification annotations. Before introducing different specification annotations, the general semantics of Java 5 annotations is discussed. Afterwards, in Section 5.2, the main specification annotations are covered. A syntactically lighter, but less expressive version of these annotations, so called flyweight specifications, is introduced in Section 5.3. In Section 5.4, specification specific expressions are introduced, followed by a short section about container annotations, Section 5.5. At the end, Section 5.6, the intermediate results of the collaboration between the *Modern Jass* and the JML 5 project are presented.

5.1 Semantics of Annotations

The general semantics of Java 5 annotations are going to explained in this section. Chapter 9.7 of the Java language specification defines an annotation as a modifier that consists of the name of an annotation and zero or more element-value pairs [GJSB05]. Further it says '*The purpose of an annotation is simply to associate information with the annotated program element*' and thereby defines the default semantics of annotations as nothing else than additional information for a program element. The interpretation of annotations is left to third party tools and, hence, the semantics of annotations result from the processing tools. For *Modern Jass*, the specification annotations have two semantics depending on whether a program is compiled or executed. During compilation, the semantics of specification annotations are, that they may abort compilation with an error if the content of an annotation is invalid. At runtime, specification annotations are assertions and their semantics are that assertions must hold, or a runtime error occurs. In the following sections, for each *Modern Jass* specification annotation its assertion kind is identified, and what rules must be fulfilled in order to compile successfully.

Abstract Annotation Descriptor

In the following, different specification annotations are introduced. The goal of this introduction is to define the semantics and validation rules for the specification annotations, and thereby providing a reference for third party tool implementers. To keep things simple and to avoid ambiguity, a description template for annotations has been used. The following subsections can be found in that template:

Targets:

> In the *targets* subsection, possibly targets for the specification annotation are listed. Except for some flyweight annotations, targets are always defined via the java.lang.annotation.Target meta-annotation. The only attribute of this annotation is an array of elements of the enumeration java.lang.annotation.ElementType.

Attributes:

> The *attributes* subsection introduces all attributes of the corresponding annotation. For every attribute, the name, data type, and possibly default value are presented. Besides, further restrictions regarding the attribute value, and the semantics of the attribute are introduced.

Inheritance:

> If a specification annotation can be inherited and how inheritance can be controlled, is presented in the subsection *inheritance*.

Validation:

> The *validation* subsection provides rules to validate the corresponding annotation. It clearly states how an annotation is used properly, and in which case a compile error must be raised.

Evaluation:

> How a specification annotation is be evaluated, is stated in the *evaluation* subsection. In some cases a further subsection, called default values, is provided, so that annotations can be evaluated even though their attribute default values are only partially overwritten.

Desugaring:

> A flyweight annotation, Section 5.3, must provide additional information to support the transformation into a heavyweight annotation. The *desugaring* subsection provides this information.

Further on, most annotation descriptions provide a short introduction and a code sample.

5.2 Main Specification Annotations

In this section the main specification annotations for *Modern Jass* are introduced. With these annotations, the behaviour of Java programs can be specified and validated. The flyweight annotations, Section 5.3, can be desugared into these annotations.

5.2.1 @Invariant – jass.modern.Invariant

The @Invariant-annotation is used to add invariants to classes or interfaces. Examples of invariants are shown in Listing 5.1. The first invariant states that every object in the array data is not the null-reference, whereas the second invariant says that capacity is never less than zero and that its visibility is private.

```
@Invariant("@ForAll(Object o : data; o != null)")
public class Buffer {

  public Object[] data;

  @Invariant(value = "capacity >= 0", visibility = Visibility.PRIVATE)
  private int capacity;
}
```

Listing 5.1: Invariants in *Modern Jass*.

At runtime invariants are checked before starting a method execution and after returning from it. In particular, invariants are checked in the case of normal termination and in the case of abnormal termination, e.g. when an exception has been thrown.

Targets:

The @Invariant-annotation can be attached to type definitions and field declarations. The targets are defined via the standard @Target-annotation, so that the placement of the @Invariant-annotation is validated by the Java compiler. In addition, multiple invariants can be encapsulated in the container annotation @InvariantDefinitions (see Section 5.5).

Attributes:

- *value*: **@Code** java.lang.String

  The attribute *value* is of the type String and ought to hold an arbitrary Java expression which evaluates to a boolean. In addition, the specification expressions @ForAll and @Exists can be used in this attribute. More on quantifiers follows in Sections 5.4.4 and 5.4.5.

  There is no default value for the *value* attribute.

  Because the String of this attribute is supposed to be valid Java code, plus specification expressions, the attribute itself is annotated with the @Code-annotation, thereby supporting automatic validation of attribute values.

- *visibility*: jass.modern.Visibility – **default** Visibility.TARGET

  The *visibility* attribute is analogue to the Java visibility modifiers plus a fifth value which is Visibility.TARGET. The visibility attributes states what visibility an invariant has, and based on its value an invariant is getting inherited or not.

  The default value of this attribute is Visibility.TARGET which means that the invariant inherits the visibility of the element it is attached to. If this invariant is a member of the @InvariantDefinitions-container, the target element of that annotation is used to derive the invariants visibility.

- *context*: jass.modern.Context – **default** Context.INSTANCE

  The *context* attribute is used to define that an invariant is checked in instance contexts only or in static and instance contexts. Static contexts are static methods and the static object initialiser, instance contexts are non-static methods and constructors.

  The default value of this attribute is Context.INSTANCE, meaning the invariant is not checked in static contexts.

- *msg*: java.lang.String – **default** "" (empty string)

  The *msg* attribute can be used to define an arbitrary message which is presented in case this invariant does not hold.

  The default value is the empty string which is done for convenience because it enables the @Invariant-annotation to be used as single element annotation (see Section 2.1.2 on page 14).

Scope:

The scope describes the visible states in which an invariant is checked at runtime. They are:

- the end of a constructor invocation which is not marked with the @Helper annotation (see Section 5.2.6),
- the begin and end of a method invocation which is not marked with the @Helper annotation,
- the end of a static class initialiser if the *context* attribute is Context.STATIC, and
- the begin and end of a static method if the *context* attribute is Context.STATIC but not marked with the @Helper annotation.

In particular, visibility modifiers do not affect the scope of an invariant. E.g. a private invariant will be checked when a public method is invoked or vice versa. Visibility modifiers only state if and how an invariant is inherited.

Note that invariants for constructors of abstract classes may not hold in all cases. If an invariant refers to a field or a method that is initialised, respectively implemented, in a subtype, execution of invariants is not possible after the supertype constructor, but before the subtype constructor, is finished. There are different ways to deal with this situation:

- When the execution of an invariant fails, the invariant is ignored at that stage. Currently, the Common JML Tool implementation propagates this solution. However, when an invariant is private, meaning it is not getting inherited and checked at the end of the subtype construction, and it fails, it is getting lost for good.

- During contract validation, elements, which are not initialised in states in which they are part of contract checks, must be identified and must cause a compile error.

- An invariant might be marked as abstract expressing that it cannot be checked until the corresponding type is fully initialised. An abstact and private invariant will not be possible, and the corresponding type must be abstract.

With pre- and post-condition, Section 5.2.2, for constructors the same problems might arise. They can refer to elements which are not initialised in the states in which they are checked. Currently *Modern Jass* does not handle these situations and fails with a specification error.

Inheritance:

When a type is getting subtyped, its invariants are inherited. The rules for invariant inheritance equal those for member inheritance in Java (chaper 6.4 in [GJSB05]). In short, applied to invariants, the rules for inheritance are:

- If the visibility attribute of an invariant is set to Visibility.PUBLIC, it is inherited to all subtypes. An interface may only have public invariants.

- If the visibility attribute of an invariant is set to Visibility.PROTECTED it is inherited by all its subtypes.

- If the visibility attribute of an invariant is set to Visibility.PACKAGE_PRIVATE it is inherited to those subtypes which reside in the same package only.

- An invariant with visibility attribute Visibility.PRIVATE is never inherited.

Validation:

For the validation of an invariant the attributes *value* and *visibility* are checked. The *msg* attribute has no effect on validity of invariants. An invariant cannot be validated successfully if one of the following rules is violated:

- The value of the attribute *value* must be a valid Java expression which evaluates to a boolean. It might possibly be extended with the specification expressions @ForAll and @Exists, see Section 5.4.
- The members referenced by an invariant, e.g. fields or methods, must have the same or a less restrictive visibility than defined by the *visibility* attribute.
- If the context of an invariant is Context.STATIC it may only reference static members and elements.

The second rule ensures that an invariant is never inherited without the elements it refers to. Otherwise a subtype inheriting such an invariant cannot fulfil it, because the subtype cannot access the members required to satisfy the invariant.

Evaluation:

If a type has more than one invariant, either by directly declaring them or by inheriting them, the results of their evaluation are connected with logical *and*. Hence, the effective invariant for the invariants I_1, ..., I_n is:

$$I_{eff} = I_1 \wedge ... \wedge I_n$$

5.2.2 @SpecCase – jass.modern.SpecCase

The @SpecCase-annotation is used to define pre- and post-conditions for constructors and methods. In the example below (Listing 5.2), a pre-condition ensures that integers, passed on to the method number, are always even. Further, the pre-condition of a method add is shown, which ensures that only non-null references will be added to the buffer, and that an exception is thrown if the null reference is about to be added.

```
@SpecCase( pre ="n % 2 == 0", preMsg = "No oddities")
void number(int n){ ... }

...

@Also({
  @SpecCase(pre="o != null", post="contains(o)"),
  @SpecCase(pre="o == null", signals=NullPointerException.class) })
void add(Object o){ ... }
```

Listing 5.2: An example of method specifications.

Targets:

With the @Target meta-annotation (Section 2.1.2) the following element types are specified as valid targets for the @SpecCase annotation:

- java.lang.annotation.ElementType.CONSTRUCTOR, and
- java.lang.annotation.ElementType.METHOD.

Further, multiple @SpecCases can be encapsulated in a container annotation, called @Also[1], which has the same targets.

Attributes:

- *pre*: **@Code** java.lang.String - **default** "" (empty string)

  With the *pre* attribute a pre-condition can be expressed. The value of this attribute is ought to be valid Java code which evaluates to a boolean. In addition to the standard Java syntax, quantifiers, can be used with this attribute, Section 5.4.4 and 5.4.5.

  Analogue to invariants, the *pre* attribute itself is annotated with the @Code annotation and, thereby, supports automated processing.

  The default value of this attribute is the empty string which means it should have no impact on contract evaluation.

- *preMsg* : java.lang.String - **default** "" (empty string)

  The *preMsg* attribute can be used to set a message which is used as an error message in case the pre-condition specified in *pre* does not hold.

  The default value of *preMsg* is the empty string. If this default value does not get overwritten, tools are free to insert a meaningful message; e.g. the respective pre-condition.

- *post*: **@Code** java.lang.String - **default** "" (empty string)

  The *post* attribute is analogue to the *pre* attribute but is used to specify a post-condition. Post-conditions specified with this attribute apply only when the method terminates normally, e.g. reaching a return statement or the end of the code.

  The value must be valid Java code evaluating to a boolean. Besides, the expression might be extended with the specification expressions @ForAll, @Exists, @Return, and @Old (see Section 5.4).

  The default value of this attribute is the empty string which means it should have no impact on contract evaluation.

- *postMsg* : java.lang.String - **default** "" (empty string)

  This attribute is analogue to the *preMsg* attribute with the difference that it is used when a post-condition does not hold.

[1] Naming the container annotation for multiple method specification 'Also' comes from the Java Modelling Language which introduced this term.

- *signalsPost* : **@Code** java.lang.String - **default** "" (empty string)

  The *signalsPost* attribute is similar to the *post* attribute but it is used to define a post-condition which must hold in case the method execution terminates by an exception. Further, to check this post-condition, the thrown exception must be listed by the *signals* attribute.

  The value of this attribute must be valid Java syntax which evaluates to a boolean and might be extended with the specification expressions @ForAll, @Exists, @Signals, and @Old (see Section 5.4).

  The default value of this attribute is the empty string which means it should have no impact on contract evaluation.

- *signals*: java.lang.Class<? extends java.lang.Exception> - **default** java.lang.Exception

  The *signals* attribute is used to specify an exception which triggers checking the exceptional post-condition. The overall exceptional post-condition is made up from the value of this attribute and the exceptional post-condition. Effectively, the exceptional post-condition is:

  $$post_{exc} = (Exception\ \mathbf{instanceof}\ \mathsf{signals} \Rightarrow \mathsf{signalsPost})$$

  The default value of this attribute is java.lang.Exception.

- *signalsMsg* : java.lang.String - **default** "" (empty string)

  This attribute is analogue to the *preMsg* attribute with the difference that it is used when an exceptional post-condition does not hold.

- *visibility*: jass.modern.Visibility - **default** Visibility.TARGET

  The *visibility* attribute is used to specify the visibility of this specification. This attribute has the same semantics as the visibility attribute of the @Invariant annotation, see Section 5.2.1.

  The default value of the *visibility* attribute is Visibility.TARGET and, thereby, the visibility of the annotated element.

Note that all attributes of the @SpecCase-annotation have default values and, hence, it can be used as marker annotation (see Section 2.1.2 on page 14). However, such a specification would not be of any use.

Inheritance:

Method specifications are inherited just like other Java members are inherited. The rules for inheritance are derived from those for member inheritance in [GJSB05], chapter 6.4. However, a further specialisation is required because a method specification is always connected to a certain method. Thus, if a constructor or method is not getting inherited, its specification cannot be inherited either. On top of that, the visibility attribute allows a method specification be to excluded from inheritance although its target is inherited.

Validation:

The @SpecCase-annotation has three attributes (*pre*, *post*, and *signalsPost*) whose values are expected to be valid Java expressions possibly extended with the specification expressions defined in Section 5.4. It is a compile error if at least one of the following rules is not fulfilled:

- Only those specification expression can be used that are specified for the attribute, e.g. do not use @Return in the *pre* attribute.
- The value of an attribute which is marked with the @Code must be a valid Java expression possibly extended with a specification expression. On top, the expression must evaluate to a boolean.
- A method specification for a static method may not refer to non-static elements.
- Exceptions defined with the *signals* attribute must be a subtype of java.lang.RuntimeException or a subtype of any of the exceptions declared in the throws clause of the corresponding constructor or method.

Further, the value of the *visibility* attribute is validated. It must obey two rules or a compile error occurs:

- The visibility of a method specification cannot be greater, that is '*more visible*', than the visibility of its target. Otherwise a method specification could be inherited without the corresponding method.
- The members referenced by a method specification must have at least the same or a greater visibility than the specification itself. This is to ensure that a subtype must not fulfil contracts that reference members it cannot access and, hence, cannot observe or manipulate.

These rules ensure that a method specification can be executed safely, that it is not inherited without the corresponding method, and that a specification is not inherited when the elements it references are not inherited.

Evaluation:

A method specification defines pre- and post-conditions for constructors and methods. The general contract is that a pre-condition must be fulfilled before starting the method execution and that a post-condition must be fulfilled before the method finishes its execution. However, by inheritance or by using the @Also container annotation a method may have multiple specification cases and a overall pre- and post-condition must be computed. The overall pre- or post-condition is also called effective pre- respectively post-condition. The effective pre-condition for the pre-conditions $P_1, ..., P_n$ is evaluated by:

$$P_{eff} = P_1 \vee ... \vee P_n$$

A single post-condition Q is evaluated by the implication $@Old(P) \Rightarrow Q$, whereas $@Old(P)$ represents the result of evaluating the pre-condition P in the pre-state, thus, before executing the method. In the case of multiple specification cases the effective post-condition is given by:

$$Q_{eff} = (@Old(P_1) \Rightarrow Q_1) \wedge ... \wedge (@Old(P_n) \Rightarrow Q_n)$$

It is not a difference if the presence of multiple specification cases is the result of specification inheritance or by use of the @Also container. Besides, the Liskov Substitution principle (LSP) is satisfied, which states that a supertype can safely be substituted by one of its subtypes [LW93]. For design by contract this implies that pre-conditions can be weakened and that post-conditions can be strengthened.

An alternative effective post-condition is not to use the implication $Old(P) \Rightarrow Q$ but to simply use Q so that the effective post-condition for the post-conditions $Q_1, ..., Q_n$ is

$$Q_{alt} = Q_1 \wedge ... \wedge Q_n$$

Although the Liskov Substitution principle is satisfied as well, refining contracts is harder when effective post-conditions are evaluated this way. To give an example, take the contract in Listing 5.3 which defines that the null-reference may not added to a buffer and that every non-null element is actually stored in the buffer.

```
interface IBuffer {

  @SpecCase( pre = "o != null", post = "contains(o)")
  void add(Object o);
}
```

Listing 5.3: Interface of a buffer not allowing null.

A subtype of the buffer listed above, may allow the null-reference and must redefine the specification of the add method. As shown in Listing 5.4, null is allowed now and a placeholder object will be stored for it.

```
class Buffer implements IBuffer {
  private final Object _dummy = new Object();

  @SpecCase( pre = "o == null", post = "contains(_dummy)")
  void add(Object o){
    if(o == null) add(_dummy);
    ...
  }
}
```

Listing 5.4: Subtype of IBuffer allowing null.

The effective post-condition for the method add would be

$$Q_{eff} = contains(\_dummy) \ \wedge \ contains(o)$$

which evaluates to false in all cases making this specification impracticable. In contrast, when post-conditions imply the corresponding pre-condition, the effective post-condition for the method add changes to:

$$Q_{eff} = (o \neq null \Rightarrow contains(o)) \wedge (o == null \Rightarrow contains(\_dummy))$$

Default Values

Because all attributes have default values it is possible, to have a method specification that is only partially defined. E.g. a post-condition without a pre-condition, or an exceptional post-condition without an explicitly defined exception type. In such a case, the missing parts are set to undefined ($undef$), which means they have no impact on evaluation. For the effective pre-condition, an empty pre-condition ($undef$) can be omitted:

$$P_{eff} = P_1 \vee ... \vee P_n \vee undef \equiv P_{eff} = P_1 \vee ... \vee P_n$$

In particular, if there is no pre-condition at all, the effective pre-condition remains unchanged. If a method specification has no pre- but a post-condition:

$$(@Old(undef) \Rightarrow Q) \equiv (true \Rightarrow Q) \equiv Q$$

is getting evaluated, which means the missing pre-condition has no impact. In case of a missing post- but present pre-condition, the evaluation of the effective post-condition is not affected either:

$$(@Old(P) \Rightarrow undef) \equiv (@Old(P) \Rightarrow true) \equiv true$$

When looking at the default values of the attributes *signals* and *signalsPost*, a differentiation between normal and exceptional behaviour must be made. The normal behaviour expects that during method execution no exception is thrown, and thus the default for *signalsPost* is $false$. Combined with the default of *signal*, java.lang.Exception, every exception which is thrown during method execution will result in a failing exceptional post-condition. However, when *signal* is explicitly set, the default value for *signalsPost* changes to *true* and, thereby, specifying exceptional behaviour. The following examples outline the differences between normal and exception behaviour:

```
@SpecCase( pre = "true", post = "true")
public void m(){ ... }
```

Listing 5.5: A method specification that does not define exceptional behaviour.

For above method specification (Listing 5.5), the occurrence of an exception e will result in

$$e \textbf{ instanceof } \textsf{java.lang.Exception} \Rightarrow false$$

meaning, that every exception that occurs, invalids the exceptional post-condition. In contrast the method specification below (Listing 5.6), will result in an exceptional post condition which is:

$$e \textbf{ instanceof } \textsf{java.lang.Exception} \Rightarrow true$$

In this case, the exceptional post-condition will evaluate to true and the exception is simply re-thrown.

```
@SpecCase( pre = "true", post = "true", signals = Exception.class)
public void m(){ ... }
```

Listing 5.6: A method specification that defines exceptional behaviour.

Note that exceptional behaviour only considers exceptions and not errors. According to the Java language specification, section 11.2.4 in [GJSB05], an error is problem from which a program usually does not recover and it would not make sense to specify such problems.

Comparing JML with *Modern Jass*, it can be concluded, that with the varying default values of the @SpecCase annotation, *Modern Jass* implicitly defines exceptional and normal method specifications. In JML this differentiation is made explicit with the clauses normal_behaviour and exceptional_behaviour.

5.2.3 @Model – jass.modern.Model

A model variable is a variable which is much alike a Java field (see chapter 8.3 in [GJSB05]), but accessible by specifications only. A Java field must be declared, can be assigned a value, and can be read by other expressions. The @Model-annotation is used to declare a model variable. The declaration consists of the model variable name and data type. In contracts to a Java field, the visibility of a model variable cannot be changed and is public by default.

Targets:

The @Model-annotation can only be added to type definitions. This is expressed with the @Target meta annotation whose value is java.lang.annotation.ElementType.TYPE. Further, multiple model variable definitions can be encapsulated by the container annotation @ModelDefinitions.

Attributes:

- *name*: java.lang.String

  The name of the model variable is defined by this attribute. There is no default value.

- *type*: java.lang.Class<?>

  This attribute is used to set the data type of the model variable. The fully qualified class name plus the .class-literal is required, e.g. java.lang.String.class. If a model variable is supposed to have a primitive type (char, int, ...), the respective wrapper classes must be used, e.g. java.lang.Long for the primitive type long.

  There is no default value for this attribute.

Inheritance:

All model variables are inherited to subtypes of the defining type. Therefore, the @Model annotation has no visibility attribute.

Validation:

All model variables are validated by the following rules and its is a compile error if at least one rule cannot be validated successfully:

- The name of a model variable must be a valid Java identifier as defined by the Java language specification, chapter 3.8 in [GJSB05].
- Independent of the data type of a model variable, a name can only be used once in a class type. Note that this rule also applies to inherited model variables.
- The class type in which a model variable is declared must be abstract or the corresponding @Represents annotation must be present.

Note that a model variable can only be used if it is represented, meaning, it relates to a concrete value. The next section will introduce the @Represents annotation which is used to attach a value to a model variable.

5.2.4 @Represents – jass.modern.Represents

The @Represents annotation is the complement of the @Model annotation. While a model variable is declared by means of the @Model annotation, its value is defined by the @Represents annotation.

Targets:

The @Represents annotation can be added to a type definition or field declaration. The respective values of the @Target meta annotation are java.lang.annotation.ElementType.TYPE and java.lang.annotation.ElementType.FIELD. Additionally, multiple @Represents annotations can be encapsulated in the container annotation @RepresentsDefinitions.

Attributes:

- *name*: java.lang.String

  The *name* attribute represents the name of the model variable which is targeted by this annotation. There is no default value for this attribute.

- *by*: **@Code** java.lang.String

  With the *by* attribute an expression is defined whose value is attached to the referenced model variable. The value of this attribute must be valid Java code which evaluates to the type of the model variable (see *type* attribute of the @Model annotation).

Specification expressions (Section 5.4) are not allowed for the *by* attribute.

Validation:

The attributes of the @Represents annotation are validated in combination with the model variable declarations. It is an error if at least one of the following rules is violated:

- The value of the *name* attribute must match a model variable declaration (see @Model annotation) of the same class type or in one of the supertypes (super-class or implemented interfaces).

  There is no default value for this attribute.

- The value of the *by* attribute must be a valid Java expression which evaluates to the data type of the corresponding model variable. Expression evaluation and casting of data types is defined by the Java language specification [GJSB05], chapter 15.

  The default value of the *by* attribute is the empty string which gets replaced with the name of the field @Represents is added to. Nevertheless, the type of the field, must be compatible with the type of the corresponding model variable.

Evaluation:

The value of the @Represents annotation is defined by the attribute *by*. Note that the value of the corresponding model variables is not computed once and stored, like a variable assignment, but evaluated every time the model variable is accessed. In particular, this means that the value of a model variable might change over time.

5.2.5 @Pure – jass.modern.Pure

The @Pure annotation is a marker annotation which is used in *Modern Jass* to mark either a method or a type as side-effect free. A type is side-effect free if all of its methods are side-effect free. A method is side-effect free if it does not change the externally visible state of the class, which means that only local variables are allowed to get changed during execution of a method marked as *pure*. An example of a side-effect free method is the equals-method in Listing 5.7.

```
@Pure public boolean equals(Object other){
 return other != null && other == this;
}
```

Listing 5.7: Side-effect free equals method of a singleton class.

It is strongly encouraged to reference only those methods in contracts, e.g. invariants, pre-, or post-conditions, which are marked as pure. Otherwise, checking a method specification might have unforeseen side-effects. Nevertheless, some method might be side-effect free but the @Pure method cannot be added. For instance, closed-source libraries or project not support Java 5 annotations, can be named here.

Evaluation

It is a non trivial task to actually check whether a method is side-effect free and compliant with its @Pure annotation or not. There are basically two approaches. Static checks at compile time or runtime checks during execution. However, both approaches have drawbacks as they are computational expensive. In the following, it is shortly sketched how these checks could be implemented and what difficulties arise.

- At compile-time a static program-flow analysis can be performed that ensures that no command that changes a field is getting executed. However, such a flow analysis must make certain assumptions, e.g. how an execution flow branches, that might lead to invalid results.

- When checking side-effect freeness at runtime, bytecode instrumentation can be used. At bytecode level the opcode PUTFIELD exists that changes the value of a field specified by its fully qualified name. The bytecode instrumentation task would be to manipulate every class that is loaded by the system under test so that an event is emitted before the PUTFIELD opcode is executed. Before proceeding, this event must inform about the current stack trace and the field, including its declaring class, so that an event receiver can ensure that the field being modified is not in the scope of an @Pure annotation. The overall process is quite simple as one can see in Figure 5.1 and is easy to implement. Nevertheless, it imposes a huge performance penalty which makes it not applicable in practice.

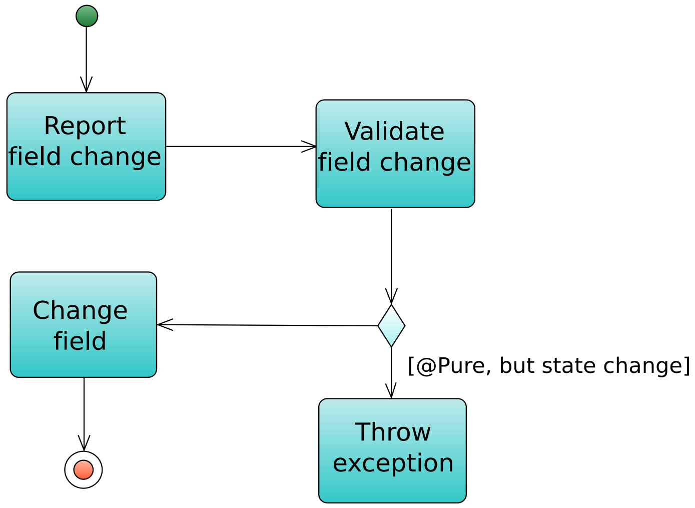


Figure 5.1: UML activity diagram showing how @Pure can be checked.

5.2.6 @Helper – jass.modern.Helper

The helper annotation is used to mark a method or constructor as helper and hence exclude it from invariant checks. Listing 5.8 shows an application of the @Helper-annotation.

```
@Invariant("data != null")
Object data[];

public void reinit(){
  reset();
  init();
}

@Post("data == null")
@Helper void reset(){
  data = null;
}

@Helper void init(){
  data = new Object[capacity];
}
```

Listing 5.8: Using @Helper-annotation to avoid invariant checks.

The methods reset and init are marked as helper because their invocations collide with the invariant that states the field data is never the null-reference. The @Helper annotation can be added to the following element types:

- java.lang.annotation.ElementType.CONSTRUCTOR, and
- java.lang.annotation.ElementType.METHOD.

Nevertheless, constructors and methods annotated with the @Helper-annotation must be private or a compile error occurs. There is no implicit @Helper-annotation for private constructors or methods.

Note that pre- and post-conditions are not affected by the @Helper-annotation. For instance, in the example above (Listing 5.8), the reset method is obligated to fulfil a post-condition although it is marked as helper.

At this stage, the main specification annotations have been introduced. They can be used to specify method behaviour and class invariants. Further, model variables and side-effect freeness, can be expressed with the presented annotations. In the following section, flyweight annotations are introduced. They are a syntactically lighter but less expressive way, to define invariants and method behaviour.

5.3 Flyweight Specification Annotations

This section introduces flyweight specification annotations which are syntactically lighter than the @SpecCase or @Invariant annotations but less expressive. Still, flyweight annotations might be preferred in case a single pre- or post-condition is to be expressed, or in case the desired assertion matches a pre-configured flyweight annotation. For instance, instead of writing '@Invariant("foo != null") Object foo', one can use a flyweight annotation, which changes the expression to '@NonNull Object foo'.

From the semantics point of view, flyweight annotations are syntactical sugar and can be transformed into 'heavier' method specifications or invariants. Transforming flyweight annotations into their heavier counterparts is called *desugaring*. In *Modern Jass* two different levels of desugaring exist: *Level 1* and *Level 2* desugaring. In the following sections each desugaring process and the corresponding sets of flyweight annotations are introduced.

5.3.1 Desugaring, Level 1

Level 1 desugaring targets those flyweight annotations that have a direct counterpart in the @SpecCase annotation. A flyweight annotation, for example, is the @Pre annotation which corresponds to the *pre* attribute of the @SpecCase annotation. Because of this, desugaring level 1 flyweight annotations is pretty straightforward:

1. Collect all level 1 flyweight annotations, compose a new @SpecCase annotation of them, and remove the flyweight annotations.

2. Add the @SpecCase annotation which has been composed in step 1 to the corresponding method. In case user-defined @SpecCases already exist, the container annotation @Also is used.

In Listing 5.9, the level 1 desugaring process is visualised. A single @Pre flyweight annotation is transformed into a @SpecCase and added to the existing specification.

```
@Pre("o != null")
@SpecCase( pre = "o == null", signals = NullPointerException.class)
void add(Object o){ ... }

  ↓

@Also({
 @SpecCase( pre = "o != null"),
 @SpecCase( pre = "o == null", signals = NullPointerException.class) })
void add(Object o){ ... }
```

Listing 5.9: Desugaring a level 1 flyweight annotation into a @SpecCase annotation.

The following sections introduce the level 1 flyweight annotations currently implemented in *Modern Jass*.

5.3.2 @Pre – jass.modern.Pre

The @Pre flyweight annotation is used to add a pre-condition to a constructor or method. Its is equivalent to the *pre* attribute of the @SpecCase annotation. An example which specifies that a given method parameter is never less than zero is given in Listing 5.10.

```
@Pre("index >= 0")
public Object get(int index) { ... }
```

Listing 5.10: The flyweight annotation @Pre.

Targets:

The @Pre annotation can be added to constructors and methods. These targets have been defined via the Target meta-annotation, making the Java compiler validate the placement of the @Pre annotation.

Attributes:

- *value*: **@Code** java.lang.String

  The *value* attribute is the only attribute of the @Pre annotation and is used to express a pre-condition. The value of this attribute must be a valid Java expression which evaluates to a boolean. On top, the specification expression @ForAll and @Exists are allowed.

  There is no default value for this attribute.

Validation

Validation of the @Pre annotation is the equal to the validation of the *pre* attribute of the @SpecCase annotation.

5.3.3 @Post – jass.modern.Post

With the @Post annotation a post-condition can be expressed. It can be used with constructor or methods and is equivalent to the *post* attribute of the @SpecCase annotation. The example in Listing 5.11 shows a post-condition which ensures that a method never returns the null reference.

```
@Post("@Result != null")
public Object get(int index){ ... }
```

Listing 5.11: A post-condition expressed with a flyweight specification annotation.

Targets:

The target of the @Post flyweight annotation are equal to those of the @Pre and @SpecCase annotation. The possible targets for the @Post annotation are constructors and methods.

Attributes:

- *value*: **@Code** java.lang.String

  The attribute *value* is used to express a post-condition. It must be a valid Java expression possibly extended with quantifiers and must evaluate to a boolean.

  There is no default value for this attribute.

Validation:

The value of the @Post flyweight annotation is equivalent to the *post* attribute of the @SpecCase annotation. Because flyweight annotation are desugared before validation is performed, the validation of the @Post annotation is the same as for the *post* attribute (see Section 5.2.2).

5.3.4 Desugaring, Level 2

In contrast to level 1 flyweight annotations, level 2 annotations have no direct counterpart in the @SpecCase annotation. Further, a level 2 flyweight annotation does not allow to express an assertion as Java code, but the annotation itself is expressing the assertion. Its attributes are used to specify parameters only. To give an example, the flyweight annotation @Length(long) denotes, simply by its name, that the length of the annotated element is asserted. An attribute is used to actually specify a value for the length, e.g. '@Length(12) int[] i'. In the course of this section, different level 2 flyweight annotations are introduced, but prior to that, desugaring is explained.

Desugaring level 2 flyweight annotation is more complex than desugaring level 1 flyweights because, they must be understood as asserting modifiers for program elements, and not as plain pre- or post-conditions. By *asserting modifier*, an assertion is meant, which must hold for all pre- or post-conditions. In other words, the assertions expressed by these modifiers strengthen every specification case. Generally, level 2 flyweight annotations can be added to fields, methods, and method parameters. Depending on these targets the semantics of level 2 flyweight annotations vary and desugaring must adjust.

- java.lang.annotation.ElementType.PARAMETER → *Pre-condition**

  If a level 2 flyweight annotation is added to a method parameter, it must be understood as an additional modifier which strengthens pre-conditions. In contrast to level 1 desugaring, it is not sufficient to simply create a new pre-condition and to add it to the existing ones. Instead, the assertions from a level 2 flyweight annotations are merged with every existent pre-condition. In this context, merging must be understood as the logical connection through *and.*

  If the corresponding method has no pre-conditions, a new one is created from the level 2 flyweight annotation.

- java.lang.annotation.ElementType.METHOD → *Post-condition**

  Level 2 flyweight annotations that are added to a method are analogue to those added to a method parameter. In this case, not the pre-conditions but post-conditons get strengthened with the assertion expressed in the flyweight annotation.

  In case no post-conditions exist, assertions are not merged, but a new one is created.

- java.lang.annotation.ElementType.FIELD → *Invariant*

  Whenever a level 2 flyweight annotation is added to a field, an invariant is created from it. In contrast to above, merging invariants is not required, because the effective invariant of a class consists of the conjunction of all invariants, see Section 5.2.1.

In Figure 5.12 the level 2 desugaring process is visualised. Two different flyweight annotations, @Length and @NonNull, have been used, and it can be seen how they are merged into pre- and post-conditions.

```
@Also({
  @SpecCase( post = "@Result.length != 0"),
  @SpecCase( pre = "index < 0 || index >= data.length",
    signals = ArrayIndexOutOfBoundsException.class) })
@NonNull public String toString(@Length(7) Object[] data, int index){

  return data[index].toString();
}

  ↓

@Also({
  @SpecCase( pre = "data.length == 7",
    post = "@Result.length() != 0"),
  @SpecCase( pre = "data.length == 7
      && (index < 0 || index >= data.length)",
    signals = ArrayIndexOutOfBoundsException.class) })
@NonNull public String toString(Object[] data, int index){

  return data[index].toString();
}

  ↓

@Also({
  @SpecCase( pre = "data.length == 7",
    post = "@Result != null && @Result.length != 0"),
  @SpecCase( pre = "data.length == 7
      && (index < 0 || index >= data.length)",
    signals = ArrayIndexOutOfBoundsException.class,
    post = "@Result != null" ) })
```

```
public String toString(Object[] data, int index){

  return data[index].toString();
}
```

Listing 5.12: Desugaring level 2 flyweight annotations into pre- and post-conditions. Firstly, the method parameter annotation is desugared, and secondly the method annotation is desugared.

Targets

Generally, level 2 flyweight annotations can be added to the program element types field, method parameter, and method. These targets are specified using the @Target meta-annotation, but on top of that the data type of these targets must be specifified. This is because the sole differentiation by the element type might not be sufficient for level 2 flyweight annotations as they also depend on the data type of a specific element. To give an example, the @Length annotation cannot be applied to types where measuring the length is not meaningful. Consequently, that target definition of a level 2 flyweight annotation must not only specify the element types (field, method, ...), but also the data type of these element types.

Desugaring-Pattern

The nature of level 2 flyweight annotation is that they express an assertion simply by their type and not by an user-defined assertion (e.g. @NonNull vs. param != null). Nevertheless, from level 2 annotations a Java expression must be generated that can be validated at compile-time and that can be checked at runtime. Code generation is based on a code pattern that every level 2 flyweight annotation must specify. The pattern mixes Java code with *Modern Jass* specific expressions and must evaluate to a boolean. Specific *Modern Jass* expressions are used to refer to attribute values, and to refer to the actual annotated element. The following expressions can be mixed with Java code:

- @AnnotatedElement – Represents the reference of the annotated element, e.g. the name of a parameter or field.
- @MeasureLength – Evaluates to the length of certain types. Supported types are Java arrays, Strings, and subtypes of java.lang.Collection. In the latter case, the method size of the collection type is used.
- @ValueOf(*attribue*) – A function that expects an attribute name as its argument and represents the value of that attribute.

An example is the @Length annotation which asserts that the length of an array, java.lang.String, or java.util.Collection is equals to the one specified through this annotation. The desugaring pattern for the @Length annotation is:

@AnnotatedElement.@MeasureLength == @ValueOf(*value*)

Above pattern is used during desugaring, and when writing '@Length(value = 12) int[] i', the Java expression 'i.length == 12' will be generated.

Validation

Because level 2 flyweight annotations are desugared into invariants, pre-, or post-conditions, they are targeted when these assertion get validated. Nevertheless, level 2 flyweight annotations use a stricter target selection mechanism as they not only differentiate by the element (field, parameter, or method) but also by the type of the target (e.g. primitive types, arrays, certain class types, ...). Consequently, the proper use of level 2 flyweight annotations must be validated and, hence, it is a compile error if an level 2 flyweight annotation is added to an element whose data type is not compatible with the types defined by the annotation. This validation step is performed during desugaring automatically. Furthermore, every flyweight annotation may define additional rules by which it is validated.

Inheritance

Because flyweight annotations are desugared into equivalent @SpecCase annotations there is no other support for inheritance than it is for @SpecCases (Section 5.2.2). In particular, this means that level 2 flyweight annotations are not merged with pre- or post-conditions of its subtypes.

Distinction between Level 1 & Level 2 Desugaring

Note that level 1 and level 2 flyweight annotations have different semantics when used in combination or with other specification cases. In particular, the feature of being a modifier instead of a pre- or post-condition distinguishes level 2 flyweight annotations from level 1 flyweights. The following code snippets outline and clarify the differences.

```
@Pre("o != null")
@SpecCase( pre = "o == null", signals = NullPointerException.class)
public void add1(Object o){ ... }
```

Listing 5.13: Having two distinct pre-conditions.

In Listing 5.13, effectively two pre-conditions are specified. One is expressed with the @Pre annotation and the other with the pre attribute of the @SpecCase annotation. The effective pre-condition for the method add1 will be ($pre_{eff} = (\mathsf{o} \neq \mathsf{null} \lor \mathsf{o} == \mathsf{null})$). In contrast, the effective pre-condition of method add2, Listing 5.14, is ($pre_{eff} = (\mathsf{o} \neq \mathsf{null} \land \mathsf{o} == \mathsf{null})$). This is the result of merging the modifier assertion into every pre-condition and might puzzle inexperienced users.

```
@SpecCase( pre = "o == null", signals = NullPointerException.class)
public void add2( @NonNull Object o){ ... }
```

Listing 5.14: Having a single pre-condition which never holds.

In the following subsections every flyweight annotation is presented. Each subsection will provide an example, information about possibly attributes, and desugaring information like the desugaring-pattern and a list of allowed data types.

5.3.5 @NonNull – jass.modern.NonNull

The @NonNull flyweight annotation is used to express that a field or parameter may not be null, or that a method never returns null. Listing 5.15 shows a field data that is marked with the @NonNull annotation. It is equivalent with an invariant stating 'data != null'.

```
...
private @NonNull Object[] data;
...
```

Listing 5.15: A level 2 flyweight annotation expressing an invariant.

A method signature with the @NonNull annotation is shown in Listing 5.16. It expresses a pre-condition which ensures that a parameter may not be the null reference.

```
public void add(@NonNull Object o){ ... }
```

Listing 5.16: Using the NonNull annotation with a method parameter.

Attributes:

The @NonNull flyweight annotation is a marker annotation, hence it has no attributes.

Desugaring:

- *pattern*: @AnnotatedElement != null

  During desugaring, the placeholder @AnnotatedElement is substituted with the name of the annotated field or parameter, or with a reference to the method return value. The result is Java code, which can be added to an invariant or method assertion.

- *targetTypes*: { java.lang.Object }

  In addition to the @Target meta-annotation, the allowed targets of the @NonNull annotations are defined by the type of the element. Only java.lang.Object and its subtypes are allowed.

  Note that with java.lang.Object every object is captured even primitives which will be automatically cast into the corresponding wrapper-class.

Note that @NonNull does make sense for primitive types. Since version 5, Java supports a technique called 'auto boxing' which converts a primitive type automatically into its wrapper reference type, see section 5.1.7 and 5.1.8 in [GJSB05]. As a consequence, a primitive type might be a boxed null reference, e.g. when writing 'int i = (Integer) null'.

5.3.6 @Length – jass.modern.Length

The @Length annotation is applicable for Strings, arrays, and all kinds of lists (java.util.List) and asserts that their length matches the one specified by the annotation. Listing 5.17 gives examples of the @Length annotation.

```
@Length(12) private List<Integer> data;

public void setPostCode(@Length(8) String str){ ... }
```

Listing 5.17: Usages of the @Length flyweight annotation.

Attribute:

- *value*: long

  The attribute *value* is used to define the length of the target. It must be an integer greater than or equal to zero.

  There is no default value for this attribute.

Desugaring:

- *pattern*: @AnnotatedElement.@MeasureLength == @ValueOf(*value*)

  The desugaring-pattern above uses three placeholders which stand for the reference of the target, a length function whose result depends on the type of the target, and the @ValueOf-function.

- *targetTypes*: {Java arrays, java.lang.String, java.util.List}

  The @Length annotation is applicable for all Java arrays, strings, and classes that implement the interface java.util.Collection.

5.3.7 @Min – jass.modern.Min

The @Min flyweight annotation can be used to specify the minimal value of a numerical field, parameter, or method return value. An example where a method parameter is checked to be greater or equal to zero is shown in Listing 5.18.

```
public void setAge(@Min(0) int age){ ... }
```

Listing 5.18: The @Min flyweight annotation.

Attributes:

- *value*: double

  With the attribute *value* a numerical value, which must be less or equal to the value of the annotated element, is specified.

  There is no default value for this attribute.

Desugaring:

- *pattern*: @ValueOf(*value*) $\leq$ @AnnotatedElement

  The pattern uses a placeholder for the annotation target and the @ValueOf evaluation function. For example, the expression '@Min(123) int a' will be desugared into the Java expression '123 <= a'.

- *targetTypes*: { java.lang.Number }

  The @Min flyweight annotation can be placed on all elements that are numbers. In Java, these are represented by the wrapper class java.lang.Number which also matches the numerical primitive types.

5.3.8 @Max – jass.modern.Max

The @Max flyweight annotation is the counterpart of the @Min annotation and can be used to specify the maximum value of numerical types. Listing 5.19 gives an example how the @Max annotation can be used.

```
public void openValve(@Max(100) double valve){ ... }
```

Listing 5.19: The @Max flyweight annotation.

Attributes:

- *value*: double

  The *value* is used to define a numerical value, which must be greater equal to the value of the annotated element.

  There is no default value for this attribute.

Desugaring:

- *pattern*: @AnnotatedElement $\leq$ @ValueOf(*value*)

  This pattern is used during desugaring so that the placeholder @AnnotatedElement is replaced with the annotated element and the @ValueOf function is evaluated.

- *targetTypes*: { java.lang.Number }

  The @Max annotation can only be used with numerical values. The class java.lang.Number is the supertype of all numerical types in Java, which includes primitive types as well.

5.3.9 @Range – jass.modern.Range

The @Range flyweight annotation can be seen as the combination of the @Min and @Max annotation. As shown in Listing 5.20, the @Range annotation can be used to specify the minimum and maximum value of a numerical type.

```
@Range(1, 10) int getVote(){ ... }
```

Listing 5.20: The @Range flyweight annotation

In this case, a post-condition is expressed, stating that the method getVote must return a value between 1 and 10, inclusively.

Attributes:

- *from*: double

  With the *from* attribute the lower bound (inclusive) of the range is specified. This attribute has no default value.

- *to*: double

  With the *to* attribute the upper bound (inclusive) of the range is specified. There is no default value for this attribute.

Desugaring:

- *pattern*:
  @ValueOf(*from*) ≤ @AnnotatedElement && @AnnotatedElement ≤ @ValueOf(*to*)

  The desugaring pattern of the @Range annotation uses the @AnnotatedElement reference and the @ValueOf function. Desugaring results in a conjunction ensuring the *from* value to be the lower bound, and the *to* value not to be exceeded by the annotated element.

- *targetTypes*: { java.lang.Number }

  The elements annotated with the @Range annotation must have number types. In Java these are the primitive integer and floating point types and the subtypes of the java.lang.Number. However, even the primitive numerical types are captured by this class.

To sum up the previous two sections, it must be concluded that a set of Java 5 annotations for specification purposes has been defined. At different levels of complexity and expressiveness, behaviour of Java programs may be specified with these annotations. In the following sections, specification expressions and container annotations are introduced. They are the final points in the description of specification annotations for *Modern Jass*.

5.4 Specification Expressions

Specification expressions are used to refer to special value like method return values, or to provide shorthands like quantifiers. They can be used in combination with Java expressions. This section introduces the specification expressions that are allowed in *Modern Jass* specifications.

5.4.1 @Result

The @Result specification expression is used to refer to the return value of a method. Its type equals the method return type, and thus it cannot be used with void methods. Naturally, @Result can only be used in post-conditions with normal termination. An example is given in Listing 5.21 where a post-condition ensures that odd numbers are returned only.

```java
@Post("@Result % 2 == 1")
public int oddities(){ ... }
```

Listing 5.21: Accessing the return value of a method in its post-condition.

Instead of @Result, the specification expression @Return can be used too. Both are exactly the same.

5.4.2 @Signal

The @Signal specification expression is analogue to @Result, but it is used in an exceptional post-condition and refers to the exception that has been thrown. In the following example, Listing 5.22, an exceptional post-condition ensures that every exception, that is getting thrown, has an error message.

```java
@SpecCase(
  signals = Exception.class,
  signalsPost = "@Signal.getMessage() != null")
public void m() throws Exception { ... }
```

Listing 5.22: An exceptional post-condition accessing its cause.

5.4.3 @Old

With the @Old specification expression values from the pre-state can be accessed in the post-state. The post-condition in Listing 5.23 uses the pre-state value of a variable size to ensure that it got increased by one.

```
@Post("@Old(size) + 1 == size")
public void add(Object o){ ... }
```

Listing 5.23: Using the pre-state values in a post-condition.

With @Old the state of primitive types and reference types can be captured. For reference types, however, only the reference is stored and not a *real* copy. If the referenced objects are supposed to remain unchanged, than immutable objects or Java's clone mechanism is suggested. More on pre-state capturing can be found in Section 3.3.3.

5.4.4 @ForAll

With the @ForAll specification expression an assertion can be stated which must hold for every element in a collection of elements. The syntax of the @ForAll expression is based on the enhanced-for-loop (chapter 14.14.2 in [GJSB05]), and can be described by the following grammar and remarks:

```
@ForAll(Type Identifier: Expression; Assertion)
```

Listing 5.24: Grammar of the @ForAll specification expression.

- *Type* – must be a Java type which is compatible with *Expression*.
- *Identifier* – must be a valid Java identifier, chapter 3.8 in [GJSB05].
- *Expression* – must either implement the interface java.util.Iterable, or be an array.
- *Assertion* – must be a valid Java expression that evaluates to a boolean.

The boolean expression (*Assertion*) is checked for every element that is provided by the Iterable or array. In Listing 5.25 an invariant is shown that ensures that no element in a collection is the null reference.

```
...
@Invariant("@ForAll(Number tmp : fNumbers ; tmp != null)")
private Collection<Number> fNumbers;
...
```

Listing 5.25: The @ForAll expression used with an invariant.

5.4.5 @Exists

The @Exists specification expression is analogue to @ForAll, but it states that an assertion holds for at least one element in a collection. Again, the grammar is based on Java's enhanced-for-loop:

```
@Exists(Type Identifier: Expression; Assertion)
```

Listing 5.26: Grammar of the @Exists specification expression.

The boolean expression specified in *Assertion* is checked against every element in the Iterable or array, and the whole expression evaluates to true if at least one element satisfied *Assertion*. Listing 5.27 shows a pre-condition that ensures that at least one element in an integer array has the value zero.

```
@Pre("@Exists(int n : numbers ; n == 0)")
public void foo(int[] numbers){ ... }
```

Listing 5.27: Pre-condition that uses the @Exists specification expression.

5.5 Container Annotations

This section briefly introduces container annotations. They already have been mentioned throughout the previous sections. The sole purpose of a container annotation is to add multiple annotations of the same type to a single element. Clearly, container annotations are workarounds and only required because Java does not allow to have the same annotation more than once on an element (see chaper 9.6 in [GJSB05] or Section 2.1.2). However, with the approval of JSR 308 [CE06] the Java specification might be changed, so that multiple annotations will be allowed on the same element. This will make container annotation dispensable. In *Modern Jass* there are four container annotations:

- jass.modern.Also – A container for multiple @SpecCase annotations.
- jass.modern.InvariantDefinitions – A container for multiple @Invariants.
- jass.modern.ModelDefinitions – A container for multiple @Model annotations.
- jass.modern.RepresentsDefinitions – A container for multiple @Represents annotations.

Except for @Also, all container annotations stick to the naming-pattern of appending 'Definitions' to the actual annotation name. @Also has been chosen to align with the Java Modelling Language (JML).

5.6 Outlook – JML 5

Kristina Boysen is a master's student under Gary T. Leavens at the Iowa State University where the Java Modelling Language (JML) has been developed. In her master's thesis, Kristina Boysen works on a Java 5 annotation based version of JML which is supposed to replace the current, Java comment based, specification syntax. Her project is called *JML 5* and has two main goals:

- Developing a set of Java 5 annotations that can be used for specification purposes instead of Java comments.
- Updating the Common JML Tools so that they are capable of processing annotations instead of the comment-based specifications.

During the development of *Modern Jass* and the project announcement of JML 5 quite a big overlap between the specification annotations of both projects could be identified. For instance, an `@Invariant` annotation or a `@SpecCase` annotation with very similar or even equal attributes can be found in both projects. Because of that and to avoid duplicated efforts, merging the annotations of JML 5 and *Modern Jass* has been started. After deciding what annotations are required and how they are named, currently, the discussion is about attributes. In particular, *Modern Jass* expects different types for some attributes. On top, the meta-annotations that *Modern Jass* uses for desugaring and contract validation have no complement in the shared annotation set and these parts of *Modern Jass* must be refactored.

Nevertheless, merging the specification annotations of both projects is desirable because a standardised set of Java 5 annotations may spread easier. Users will benefit from an unified specification language and tool developers do not have to reinvent these annotations. The current state of the collaboration of JML 5 and *Modern Jass*, that is the set of shared annotations, is publicly available in the JML version control system at: `https://jmlspecs.svn.sourceforge.net/svnroot/jmlspecs/trunk/JMLAnnotations`.

Chapter 6

The Modern Jass Tool

This chapter introduces basic concepts of the *Modern Jass* tool implementation. First, the software architecture with its components is described. To further the understanding, some important components and the inner working of *Modern Jass* are outlined (Section 6.2 and 6.3). Afterwards, the limitations, *Modern Jass* is coping with, are explained (Section 6.4). The end of this chapter is about different IDE integrations (Section 6.5), and about a converter for the Java Modelling Language (Section 6.6).

6.1 Architecture

Basically, *Modern Jass* is build around a contract compiler. The compiler takes an instance of a model of the program currently being compiled, transforms all contracts into Java code, and delegates the transformed model to the actual Java compiler. Depending on the use case, the contract compiler is fed by an annotation processor, or by a bytecode instrumentation agent. The UML diagram in Figure 6.1 gives an overview of these components. Altogether, these four components compose the architecture of the *Modern Jass* tool.

- Model – *jass.modern.core.model*

  Usually, Java programs are stored as class- or source-files, but when working programatically with Java programs an abstraction of these files is required. The *model* component is such an abstraction as it contains interfaces and classes reflecting the structure of Java programs. Further a TypeFactory exists, that creates instances of this model from bytecode, and from models provided by the Java annotation processing infrastructure (Section 2.1.3).

  Note that similar models of the Java programming language exist, and that it could be questioned why *Modern Jass* introduces a new one. For instance, popular and

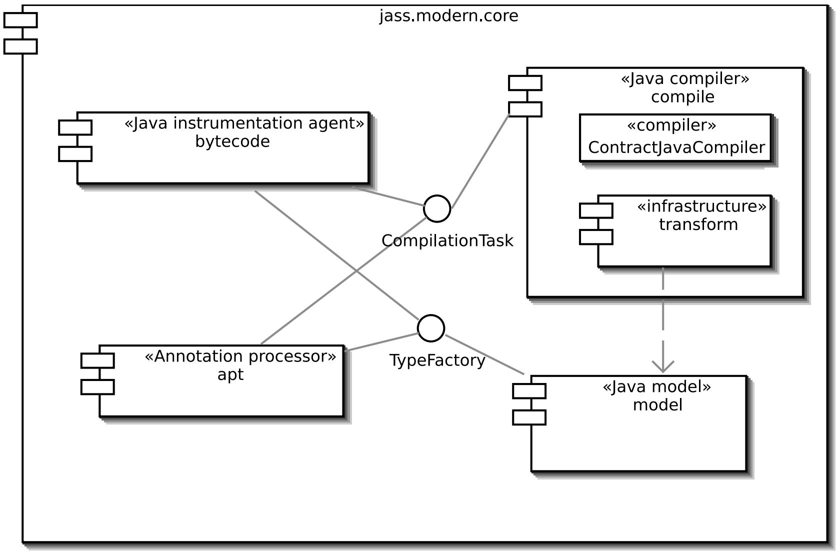


Figure 6.1: Component diagram of the *Modern Jass* architecture.

powerful models are provided by Eclipse JDT and by the annotation processing infrastructure itself. In contrast to designing a new model, these models could have been used. Still, the model provided by the annotation processing infrastructure has not been used, because it is designed to allow read-only access. However, desugaring and contract creation transform models and require write access. The Eclipse JDT model has not been used for two reasons. First, the size of the additional libraries is quite big and will blow up the overall size of *Modern Jass* and, secondly, the model offers a lot more feature than actually needed which might degrade performance. Still, in future is shall be investigated how *Modern Jass* can benefit from the code completion and content proposal infrastructures when using the JDT-based model.

- Annotation Processor – *jass.modern.core.apt*

  The annotation processing environment enables *Modern Jass* to be a compiler plug-in and thereby participating in the compilation process. The role of an annotation processor in *Modern Jass* is to validate contracts and to report errors and warnings in contracts (Section 4.2). In Java, two annotation processing environments exist, one has been introduced with Java 5 and an updated version was introduced with Java 6 (JSR 269). In Java 5 the annotation processing environment is a separate tool, while Java 6 annotation processors are compiler plug-ins. *Modern Jass* supports both annotation processing environments. It uses the information provided by the annotation processing environments, creates a model from it, and feeds it into the contract compiler.

  Note that although the annotation processors participate in the compilation process, contracts are validated only and not woven into the compiled bytecode. This is because annotation processors cannot change the output of the Java compiler (Section 4.2).

- Bytecode Instrumentation – *jass.modern.core.bytecode*

  In Section 4.3, *'Checking Contracts at Runtime'*, bytecode instrumentation has been identified to be the most suitable way to check contracts at runtime. In *Modern Jass*, the Java instrumentation API (Section 2.2.3) has been utilised to instrument bytecode with contracts. It builds a model from the bytecode of the class being loaded, feeds it into the contract compiler, and instruments the class with the resulting bytecode.

  Actually, the steps performed during bytecode instrumentation are very similar to those performed during annotation processing. The only difference is that the generated bytecode for contracts is woven into the bytecode of the running program. For performance reasons an improved version of *Modern Jass* should omit this re-creation of contract bytecode (Section 4.2 and 4.3).

- Contract Compilation – *jass.modern.core.compile*

  The contract compiler is used to validate contract annotations, and compile bytecode from contracts. Basically, the compiler performs two processing steps, as it, firstly, transforms the model and, secondly, compiles the resulting Java code. Section 6.2 describes this process in more detail. Delegating the transformed model to the Java compiler is done via the Java Compiler API (JSR 199) which has been added in Java 6 (Section 2.2.2).

6.2 Creating Contract Code

In the previous section, the contract compiler and the transformations to make the Java compiler validate contracts have been mentioned. This section will show the fundamentals of the contract compiler, and provides an example which steps through the whole transformation process. Basically, Java code which is written as an annotation attribute is not recognised by the Java compiler as Java code, but as a plain string. Consequently, some transformations must be applied to the original program. Since the Java compiler targets only code which is in the scope of classes or methods, contracts must be represented in such a way. For assertions, a single method which returns a boolean and that implements the contract seems most suitable. By doing so, the code of a contract is targeted by the compiler and it is checked that the contract evaluates to a boolean. The overall transformation process consists of the following steps:

1. Desugar Flyweight Annotations

2. Translate Specification Expressions

3. Create Contract-Checking Methods from Annotations

Contract method creation, step 3, is done following a simple scheme. To ensure that contract-checking methods have the right variable context, a signature is chosen that is similar to the signature of the annotated method. Thereby it is ensured that a contract-checking method can access only those parameters that are available in the context of that method. Further, a contract-checking method is always added to the same type, so that fields and other methods can be accessed, too. The schemata for deriving contract-checking methods is quite simple, and depending of the annotation type a different one must be used. In listings 6.1, 6.2, 6.3, and 6.4 the schemata for the contract-checking methods of invariants, pre-conditions, post-conditions, and exceptional post-conditons are shown[1]. In these listings placeholders are marked by angle brackets that must be replaced with values derived from the annotations or their targets. In short, the substitution rules for the placeholders are:

- <target_visibility> – The visibility of the contract which is either set via the visibility attribute or derived from the target.
- <target_name> – The name of the annotated element.
- <target_type> – The data type of the annotated element.
- <target_parameter_list> – The list of parameters when the annotated element is a method or constructor[2].
- <target_return_type> – The data type of the return value when the annotated element is a method.
- <n> – With inheritance or the usage of container annotations, multiple assertions for a target may exist. Numbers are used to distinguish them.
- <assertion> – The actual assertion code of the contract. Usually the attribute values marked with the @Code meta-annotation are used (Section 5.2).
- <signal> – The exception type that has been specified for a exceptional post-condition.

```
<target_visibility> boolean
<target_name>$invariant$<n>(<target_type> <target_name>) {
 return <assertion> ;
}
```

Listing 6.1: Scheme for an invariant contract method.

[1] Note that these schemata are simplified as they also catch any exception that is thrown during contract code execution, and as they enable to access a storage container for pre-state values.

[2] For methods with no return values (void methods), the type java.lang.Void is used.

```
<target_visibility> boolean
<target_name>$pre$<n>(<target_parameter_list>) {
 return <assertion> ;
}
```

Listing 6.2: Scheme for a pre-condition contract method.

```
<target_visibility> boolean
<target_name>$post$<n>(
  final <target_return_type> _Return , <target_parameter_list>) {

 return <assertion> ;
}
```

Listing 6.3: Scheme for a post-condition contract method.

```
<target_visibility> boolean
<target_name>$signalsPost$<n>(
  final <signal> _Signal , <target_parameter_list>) {

 return <assertion> ;
}
```

Listing 6.4: Scheme for an invariant contract method.

In the following section the application of the schemata above is shown. It gives an example which shows step by step how contract code is created so that it can be compiled.

Contract Compiler – The Buffer Example

This section shows the inner working of the contract compiler with the specification of a buffer (Listing 6.5). In this example the buffer has methods to add an element to it, and to check whether an element is contained in it or not. Further, a backup storage and a counter for the next storage location exist. Altogether, the buffer uses invariants, flyweights (level 1 and 2), and method specifications.

```
class Buffer {

  @NonNull Object[] data ... ;

  @Invariant("0 <= next && next <= data.length")
  int next;

  @Also({
    @SpecCase( pre = "o != null", post = "contains(o)"),
    @SpecCase( pre = "o == null", signals = NullPointerException.class) })
  void add(Object o){
    ...
  }

  @Pure
  @Post("@Result == @Exists(Object item : data ; o.equals(item))")
  boolean contains(@NonNull Object o){
    ...
  }
}
```

Listing 6.5: The buffer example.

Basically, the transformation process is divided into three distinct steps. Each of them is compulsory and they must be performed in the same order as it is done in this example. In the end, Java source code is output, which can be fed into the Java compiler making it validate contracts.

1a. *Level 1 Desugaring*

In the first processing step, the contract compiler desugars level 1 flyweight specifications into a corresponding specification case (Section 5.3.1). In the above example (Listing 6.5) the @Post annotation of the contains method will be desugared. How this changes the method specification is shown in Listing 6.6.

```
@Post("@Result == @Exists(Object item : data ; o.equals(item))")
public boolean contains(@NonNull Object o){ ... }
  ↓
@SpecCase(
  post = "@Result == @Exists(Object item: data; o.equals(item))")
public boolean contains(@NonNull Object o){ ... }
```

Listing 6.6: The buffer example – after desugaring level 1 annotations.

1b. *Level 2 Desugaring*

After desugaring level 1 flyweight annotations, level 2 flyweight annotations are desugared. They are different because they are asserting modifiers, which means, they strengthen every existing pre- respectively post-condition, instead of adding new conditions (Section 5.3.4). In the example, the @NonNull annotation is used with a method parameter and a field. Desugaring yields an updated method specification, and an invariant (see Listing 6.7).

```
@NonNull Object[] data ... ;
  ↓
@InvariantDefinitions({ @Invariant("data != null") }) ...

...

public boolean contains (@NonNull Object o) { ... }
  ↓
@SpecCase( pre = "o != null", post = "@Result == ...")
public boolean contains (Object o) { ... }
```

Listing 6.7: The buffer example – after desugaring level 2 annotations.

2. *Translating Specification Expressions*

   The third step is to translate the specification expressions (Section 5.4) into valid Java code. The buffer example uses specification expressions in the post-condition of the method contains. After translation it will be valid Java code as one can see in Listing 6.8.

```
@SpecCase (
  pre = "o != null",
  post = "boolean _Exists1 = false;
          for(Object item : data) { _Exists |= o.item(item); }
          return _Return == _Exists1;")
public boolean contains(Object o){ ... }
```

   Listing 6.8: The buffer example – after specification expression translation.

   Note that the variable _Result is referenced by the post-condition after transformation. This variable is created during contract method creation and represents the return value of a method.

3. *Contract Method Creation*

   By now, the specification annotations have been transformed only, but the actual goal, having the compiler check Java code that is expressed with the annotations, has not been reached yet. Hence, from the annotation values, contract methods are created and added to the current class. This final processing step has been sketched already in the previous section and relies on the contract method schemata. In Listing 6.9, the contract method for the invariant @NonNull Object[] is shown.

```
@Invariant("data != null")
  ↓
boolean data$invariant$1(Object[] data){
  return data != null;
}
```

   Listing 6.9: Contract method for an invariant.

   Besides adding contract methods, the original methods are made *abstract* which means they have no implementation. By doing so, for contract validation all referenced methods can still be resolved, but the amount of data is reduced.

Applying all steps to the buffer example results in a modified type that can be fed into the standard Java compiler. In Listing 6.10 a shortened version of the transformed type is presented[3].

```
public abstract class Buffer extends java.lang.Object {

  java.lang.Object[] data;
  int next;
```

[3] When starting the *Modern Jass* tool with the flag verbose, these listings are output to the console.

```
  abstract void add( java.lang.Object o);
  abstract boolean contains( java.lang.Object o);

  boolean next$invar$1(){
    return 0 <= next && next <= data.length;
  }

  boolean add$pre$0( java.lang.Object o){
    return o != null;
  }

  boolean add$post$0(final ContractContext _Context,
      final Void _Return, java.lang.Object o){

    return contains(o);
  }

  ...
}
```

Listing 6.10: The buffer example – transformed type.

It can be seen that methods implementing contracts have been derived from the annotations. These methods, and thereby the contracts, are validated by the compiler, and bytecode is created from them.

6.3 Avoiding Endless Recursion

When writing contracts, programmers might introduce endless recursive calls of interacting methods without being aware of it. This problem and a technique *Modern Jass* uses to deal with it are presented in this section. In Listing 6.11, two methods are shown which will end up stuck in an endless recursion because of their contracts. Without any kind of intervention, calling method a or b, with contract checks enabled, would result in an endless recursion, because the contract of method a calls method b and vice versa.

```
@Pre("b()")
boolean a(){ ... }

@Pre("a()")
boolean b(){ ... }
```

Listing 6.11: When checking contracts, an indirect recursion between both methods is introduced.

In order to prevent such endless recursions, a contract context, jass.modern.core.runtime.ContractContext, was introduced. A contract context can have two states: *busy* or *idle*. When a contract context is idle, contracts can safely be checked, when it is busy, contracts cannot be checked. Usage of the contract context is bound to a simple protocol which states that the very first operation in a method must be to acquire a contract context, and that the very last operation in a method is to dispose that contract context. Figure 6.2

shows an UML sequence diagram that displays this protocol. To acquire a contract context the static method ContractContext.getContractContext must be called. It expects a set of parameters that allow to identify every method call uniquely. Those parameters are the current thread, the names of the declaring class, and the signature of the method. Altogether, an identifier is created and managed on a so called *contract* stack. If an identifier is already present when a contract context is acquired, it is marked as busy which means contract checks are not allowed.

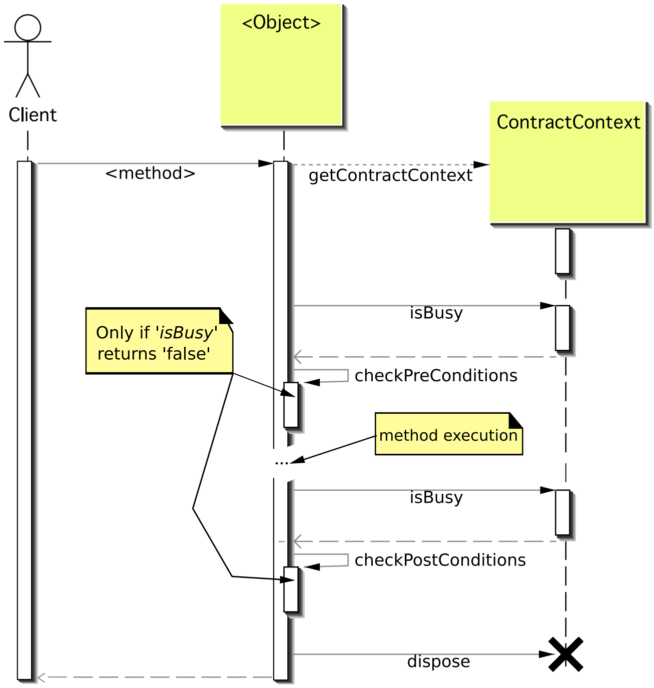


Figure 6.2: UML sequence diagram of the interaction with the contract context.

In above example, the contract context will prevent endless recursion. In Listing 6.12 the method calls and the state of the corresponding contract contexts are shown. It can be seen, that after the contracts have been checked once, further contract checks are skipped to avoid recursion.

```
a()
  → contract check of a() / ¬isBusy()
    → b()
      → contract check of b() / ¬isBusy()
        → a()
          → contract check of a() / isBusy()
          ← contract check of a() / dispose()
        ← a() / Return
      ← contract check of b() / dispose()
    ← b() / Return
  ← contract check of a() / dispose()
← a() / Return
```

Listing 6.12: Stack trace showing how the contract context prevents endless recursion.

Note that the contract context considers contracts only, and that recursive calls, that have been implemented manually, remain unchanged.

6.4 Limitations

The implementation of *Modern Jass* is limited in some ways. The goal of this section is to outline and explain these limitations, and to point to possible workarounds. It starts with limitations that come from bytecode artefacts and, afterwards focuses of restrictions of the applied technologies and implementation details.

6.4.1 Missing Debug Information

During contract validation and contract checking, *Modern Jass* creates a model of the program currently being compiled or executed. This model is either created from bytecode or from a model provided by the annotation processing environment. In such a model, types, fields, methods, and method parameters are represented. In particular, the names of these elements are important, because contracts refer to them by their names only. When creating a model from an annotation processor specific model, all the names are available, but when creating a model from bytecode the names of method parameters might be missing. This is because they are optional in Java bytecode, and only contained for debugging purposes. Therefore, when a model is created from bytecode, that lacks debugging information, the names of method parameters are simply made up from the pattern param*N*, where N is the index of the parameter starting at 1.

So, it might be possible that a contract uses the *real* names of method parameters, as specified in source code, but the model uses the generated names (param1, ...). In this case, contracts can neither be validated nor checked. Usually one can avoid this by compiling with debugging information enabled, but in the case of abstract methods, the parameter names are never represented in bytecode. For this special case, one can either use the naming scheme param*N* or the @Name annotation. The @Name annotation is a single member annotation which targets method parameters and accepts a string denoting the name of that parameter. An example is given in Listing 6.13 where the parameter name of the abstract method foo is represented by an annotation. Because annotations are represented in bytecode, the name of the method parameters can be inferred from the @Name annotation.

```
@SpecCase ( pre = "text != null" )
public abstract String foo(@Name("text") String text);
```

Listing 6.13: An abstract method that uses the @Name annotation.

An alternative is presented in Listing 6.14. It does not depend on the @Name annotation but uses a naming scheme which, from a software engineering point of view, might by considered bad style.

```
@SpecCase ( pre = "param1 != null")
public abstract String foo(String param1);
```

Listing 6.14: An abstract method that uses the param*N* naming scheme.

6.4.2 Limited Line Number Table

In Java bytecode, the line number table states from which line in the source code an opcode has been compiled. Similar to the names of local variables, the line number table is optional, and only added to the bytecode when the compiler is configured to do so. Especially with exceptions the line number table is useful, because it enables the Java virtual machine to point exactly to the source location where the exception occurred. For design by contract this should be utilised so that in case an assertion is violated and an exception is thrown, the origin of that exception points to the corresponding annotation in the source file. To give an example, when violating the pre-condition in Listing 6.15, the origin of the PreConditionError should point to line 2.

```
1 ...
2 @Pre("o != null")
3 boolean add(Object o){
4   data[next] = o;
5 }
6 ...
```

Listing 6.15: When violating the pre-condition, the JVM should point at line 2.

However, the line number tables considers method bodies only. For every method there is a line number table, that relates the opcodes to the corresponding source lines. In the above example, the line number table would start at line 4. This bytecode limitations leaves two alternatives for *Modern Jass*. Either the source location of the annotation is guessed, or the first, respectively last, code location in a method body is used. *Modern Jass* implements the latter alternative because it is impossible to infer the right code location of the corresponding annotation. Hence, when the pre-condition in the example is violated, the pre-condition error will point to line 4.

6.4.3 Maximum Number of Method Specifications

In *Modern Jass* the maximum number of declared method specifications, the number of @SpecCases at a method, is limited to 256. The reason for this limitation results from the fact that inherited specifications are represented by local contract methods, and that the maximum number of methods in a class may not exceed 65535. This limitation is defined by the Java virtual machine specification and results from the bytecode format (see chapter 4.10 of [LY96]).

As an example in which the maximum number methods is almost reached, assume a class C which has 12 methods and 6 supertypes, each with 12 methods. Every method has 256 method specifications, each with pre-, post-, and exceptional post-conditions, so that *Modern Jass* generates three contract methods for each specification. Because even inherited specifications get represented locally, the total number of methods in class C after bytecode instrumentation is:

$$\overbrace{12}^{declared\ methods} + \overbrace{12 * 256 * 3}^{contracts\ of\ declared\ methods} + \overbrace{6 * 12 * 256 * 3}^{contracts\ of\ all\ inherited\ methods}$$

which is a total of 64524 and thereby close to the maximum number of allowed method specifications. In conclusion, it can be said that the limitation of 256 method specifications per declared method exists, but will not play an important role in practice.

6.4.4 Annotation Processing for Annotated Types Only

The annotation processing framework that enables third parties to participate in the compilation process, considers only those types that are annotated with Java 5 annotations. However, when a type must fulfil obligations because of the annotations of other types, in particular supertypes, compiler plug-ins cannot be used.

For *Modern Jass* this limitation hinders the correct validation of model variables. Chapter 5, Section 5.2.4, states that a model variable must either be represented, or the defining type must be abstract. However, when a type inherits model variables, but is neither abstract nor representing the model variables, and has no other annotations at all, the annotation processing framework simply omits it. Consequently, in such a case, the proper use of model variables cannot be validated.

6.4.5 Anonymous Classes

An anonymous class is a class which is declared by its instantiation expression only (see chapter 15.9.5 in [GJSB05]). Currently, *Modern Jass* does not support these types of classes.

6.5 Integrating Modern Jass into IDEs

One goal in the design of *Modern Jass* was easy integration into today's IDEs and build processes. Choosing the annotation processing infrastructure to hook into the compilation process ensures such a seamless integration. This section introduces a plug-in, which integrates *Modern Jass* into Eclipse, and shows how to manually integrate *Modern Jass* into NetBeans and IntelliJ Idea.

6.5.1 Eclipse

Eclipse, more precisly, the Eclipse Java development tools (JDT) is an IDE for Java, which is build on top of the Eclipse platform. JDT has its own Java compiler and own user-interface elements, e.g. an editor or code browsing facilities. The Eclipse platform offers a very powerful extension mechanism based on *extension points*. Plug-ins make use of extensions point to be part of Eclipse and in return may define new extension points for other plug-ins. *Modern Jass* uses the Eclipse plug-in mechanism and adds the following features to Eclipse JDT:

Usability enhancements. The *Modern Jass* plug-in enhances the usability of *Modern Jass* in Eclipse by providing a special contract decoration and by making the *Modern Jass* runtime library available in a convenient way.

The contract decoration is a tiny overlay icon for the outline view and the package explorer. It indicates that a certain type is annotated with *Modern Jass* contracts (see Figure 6.3).

Providing the *Modern Jass* runtime library is achieved by using an extension point and, thereby, hooking into the standard extension mechanism.

Participating in the build process. JDT provides an extension point for registering an annotation processor as compilation participant. *Modern Jass* uses this extension point, and is started by JDT after every resource change. In return, Eclipse will display all error and warning messages that *Modern Jass* generates. Figure 6.4 is an example of this integration. It shows how Eclipse handles an error generated by *Modern Jass*.

Contract protected launch. In order to enable contract checks at runtime the -javaagent switch (see dynamic bytecode instrumentation in Section 2.2.3, page 23) must be set to the *Modern Jass* runtime library. In Eclipse, this can either be done manually or automatically with the help of the *Modern Jass* plug-in. Hooking into the launch management of Eclipse, *Modern Jass* creates a clone of every program launch configuration, and adds the corresponding switches to it (see Figure 6.3).

6.5.2 NetBeans & IntelliJ Idea

In addition to the plug-in based integration into Eclipse, *Modern Jass* can easily be used with other IDEs like NetBeans or IntelliJ Idea. NetBeans and IntelliJ Idea are both Java IDEs that are similar to Eclipse but competing with it.

Instead of utilising an extension mechanism, integration comes with the Java compiler itself. Because both IDEs use the Java compiler, *Modern Jass* can be used as a compiler plug-in In figures 6.5 and 6.6 compile errors are shown, that result from invalid annotation values. The sole step to integrate *Modern Jass* into NetBeans and Idea was to add the *Modern Jass*-library to the compiler classpath.

6.6 JML Converter

Since *Modern Jass* tries to align with the Java Modelling Language (JML), and since *Modern Jass* co-operates with the JML 5 project, a converter which translates specifications from JML to *Modern Jass* annotations is desirable. As part of this thesis such a converter has been designed and a proof-of-concept implementation has been made. It

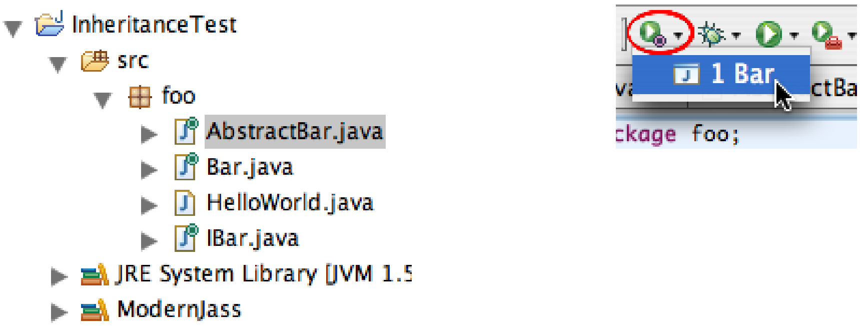
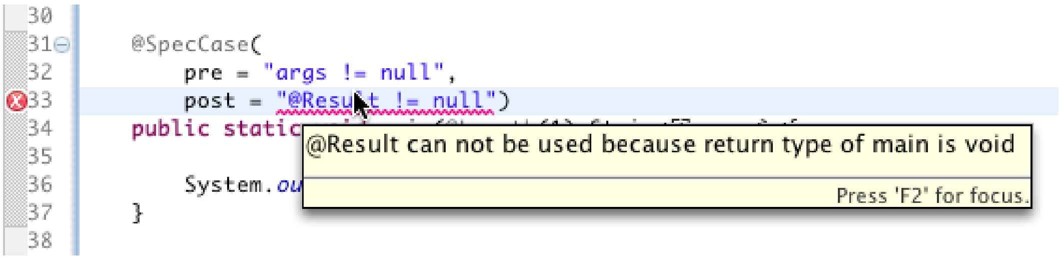


Figure 6.3: On the left, the *Modern Jass* type decoration is shown. The tiny circle on the upper right corner indicates, that a type has at least one *Modern Jass* annotation. The picture on the right shows the default Eclipse launch modes *debug program*, *run program*, and *run external program*. Beside these, the *Modern Jass* launch mode (marked with a red circle) is displayed.

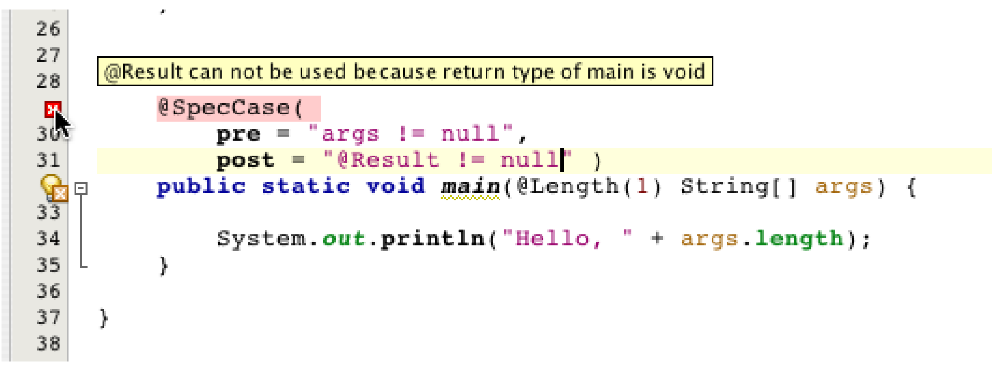


```

    @SpecCase(
        pre = "args != null",
        post = "@Result != null")
    public static

        System.ou
    }

```

@Result can not be used because return type of main is void

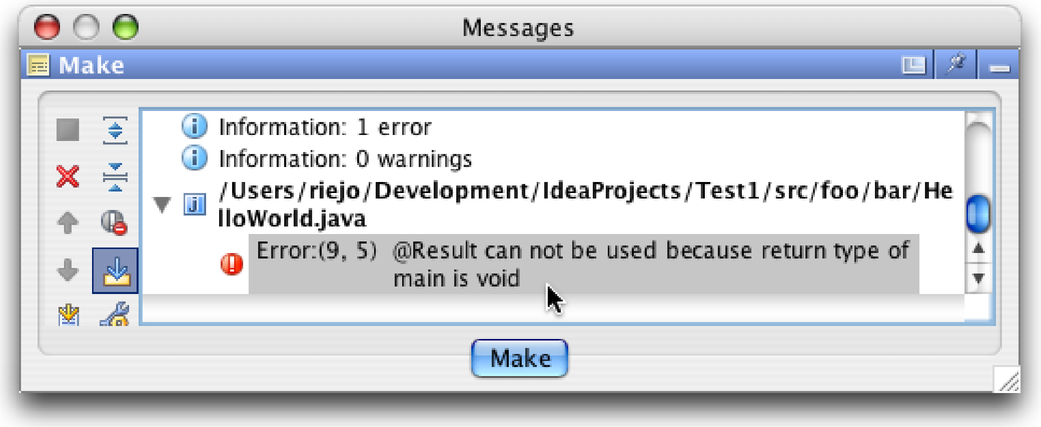
Press 'F2' for focus.

Figure 6.4: The post-condition tries to access the return value of a method with no return value (void). Consequently, *Modern Jass* generates an error which is visualised in Eclipse by a red squirrel, an error marker in the overview, and a hover message.



```
@Result can not be used because return type of main is void
    @SpecCase(
        pre = "args != null",
        post = "@Result != null" )
    public static void main(@Length(1) String[] args) {

        System.out.println("Hello, " + args.length);
    }

}
```

Figure 6.5: A compile error resulting from an invalid annotation value. NetBeans changes the background colour of the annotation to red and shows an error marker in the overview. When hovering over the error marker, the error message appears.



```
Information: 1 error
Information: 0 warnings
/Users/riejo/Development/IdeaProjects/Test1/src/foo/bar/HelloWorld.java
    Error:(9, 5) @Result can not be used because return type of
                 main is void
```

Figure 6.6: Again, a compile error results from an invalid annotation value. Idea shows a separate message window with output from the build and highlights the error message with a red expression mark and bold font.

is based on the work carried out by Martin Schnaidt in his master's thesis and on the Eclipse language toolkit (LTK) [Sch06, Fre06, Wid06].

Basic Ideas

In his master's thesis, Martin Schnaidt extended the Jass tool, so that it can use JML as its input language for runtime checking. In order to do that, a fully featured parser for JML and a reflection API has been developed. After parsing, the reflection API allows to query certain elements of a program for their specification. Listing 6.16 shows how all invariants of a class-type are retrieved.

```
ClassReflection classReflection = ...;
InvariantClause[] invariants = classReflection.getInvariantClauses();

for(InvariantClause invar : invariants){
  /* further processing of each invariant */
}
```

Listing 6.16: Retrieve all invariants of a class-type.

Martin Schnaidt's parser supports Java 5 syntax which makes it a good converter candidate. Besides parsing, specifications need to be translated, and the resulting annotations must be added to the original source document. In its simplest way, these steps can be performed by re-writing the source code based on the information provided by the reflection API. However, the Eclipse language toolkit provides an alternative way, as it supports the implementation of custom refactorings for the Eclipse IDE. The term refactoring is defined by the following quotation[4]:

> Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring.

Ever since Eclipse was available, refactorings for Java, like renaming of types or extracting methods, have been supported. In later versions, the Eclipse language toolkit (LTK) has been extracted from these features, as a language-neutral refactoring API. It can be used by third party developers and makes it very easy to implement refactorings that integrate into the Eclipse IDE, and is thereby compelling for a JML converter. Basically, the lifecycle of a refactoring consists of two steps:

1. Pre-conditions must be checked to make sure the refactoring can be applied. For the JML converter, this would be to ensure that the current source file is valid JML code.

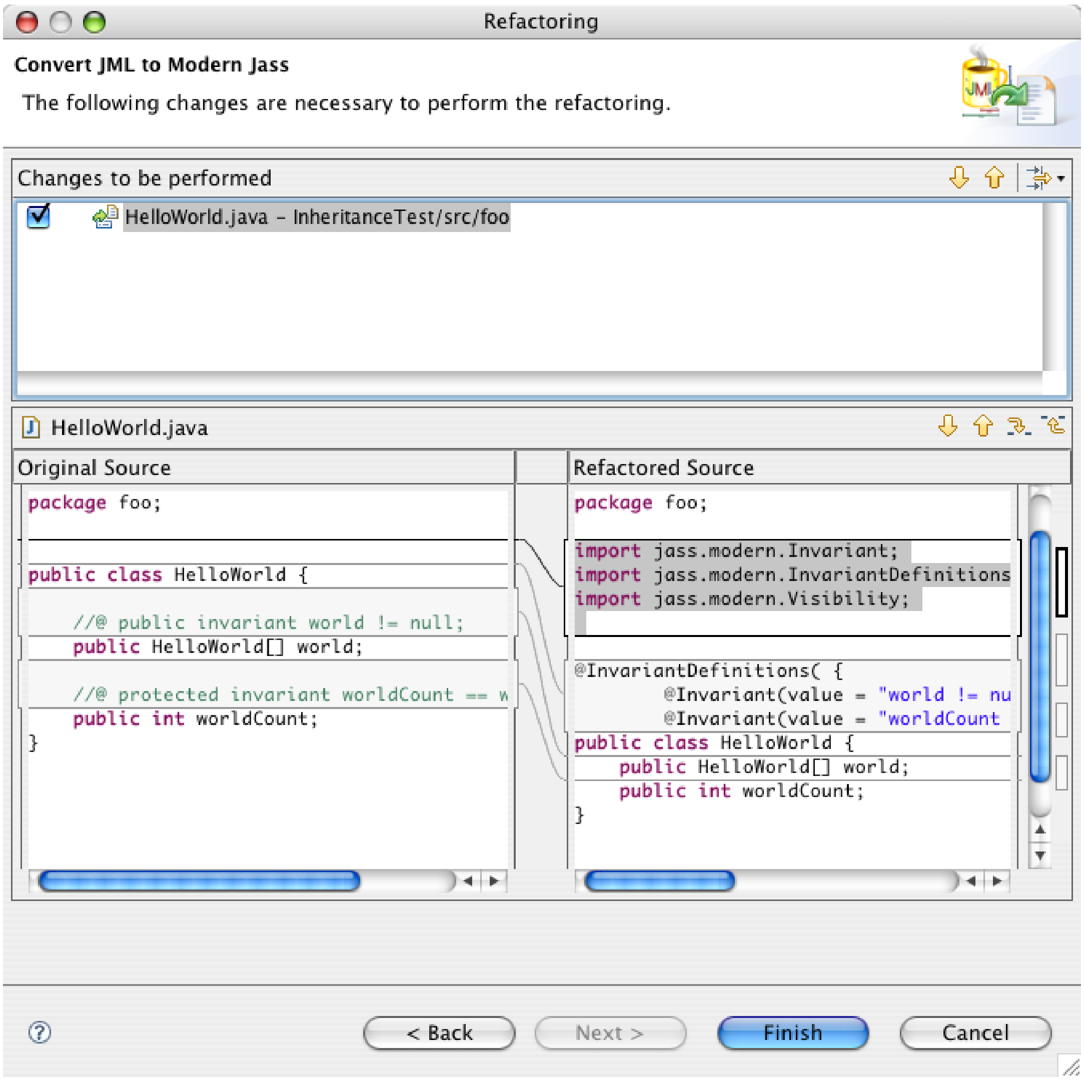
[4] Taken from `http://www.refactoring.com/`

2. A change object, which is abstraction of all transformations that are going to be applied to the underlying text file, must be computed. For JML this might be to replace a comment with a corresponding Java 5 annotation.

The Eclipse LTK provides the environment to create and mange such change objects. Further, user interface components exist, to guide through the refactoring and preview the changes. Figure 6.7 shows a screenshot of the JML converter, displaying the resource that is going to be modified, the original source, and the outcome of the refactoring.

Current State

As mentioned above, the current state of the JML converter is merely that of a proof-of-concept implementation, than a tool for productive use. Currently, only the translation of invariants is supported only. Method specifications, further field and class-level specifications are ignored. Nevertheless, the design and existing architecture of the converter should be stable enough, and it should be easy to extend it with the missing features.



Refactoring

Convert JML to Modern Jass

The following changes are necessary to perform the refactoring.

Changes to be performed

☑ HelloWorld.java – InheritanceTest/src/foo

HelloWorld.java

Original Source

```
package foo;

public class HelloWorld {

    //@ public invariant world != null;
    public HelloWorld[] world;

    //@ protected invariant worldCount == w
    public int worldCount;
}
```

Refactored Source

```
package foo;

import jass.modern.Invariant;
import jass.modern.InvariantDefinitions
import jass.modern.Visibility;

@InvariantDefinitions( {
        @Invariant(value = "world != nu
        @Invariant(value = "worldCount
public class HelloWorld {
    public HelloWorld[] world;
    public int worldCount;
}
```

Figure 6.7: Preview of the changes that are going to be applied by the JML converter. In the upper part, all resources that are going to be modified are listed and, in the lower part, a compare view shows what changes are going to be applied and how the result will look like.

Chapter 7

Case Studies

7.1 Eat Your Own Dog Food

In software development, the phrase *'eat your own dog food'* means that oneself should always use, and thereby test, the software which is currently developed. As this is not applicable for all kinds of software, a design by contract implementation is truly applicable by software developers and, thus, a candidate for the *'dog food'* paradigm. Nevertheless, a certain lead time is required before the software under development can be used for development. For *Modern Jass*, the application of this paradigm was planned and partially applied. The catch is that most of the code has been written before *Modern Jass* has reached a stable state, so most specification annotations have been added to the code afterwards.

Still, the development of *Modern Jass* has been backed up with a number of JUnit test cases. In addition to unit tests for the implementation, there are 84 test cases that explicitly check the *Modern Jass* tool. These test cases relate to a corresponding class that uses a single feature, e.g. method specifications, inheritance, desugaring, and so on.

7.2 Ring Buffer

This section introduces the ring buffer case study which has been carried out to expeditiously test *Modern Jass*. This case study is the balancing act between being easy to master and still calling on all features. Previously, Detlef Bartetzko and Martin Schnaidt used the ring buffer case study in their master's thesises to validate *Jass* respectively *JMLjass* [Bar99, Sch06]. The following will clarify what a ring buffer is, and shows its implementation and specification.

Explanation: Ring Buffer

A ring buffer is data structure for programming which is characterised by the metaphor of a ring or circle. Basically, a ring buffer consists of a fixed number of storage slots and two pointers. Both pointers move through the slots when data is red or written, and start at the beginning after passing the end. Figure 7.1 visualises a ring buffer, and how the pointer circle around its slots.

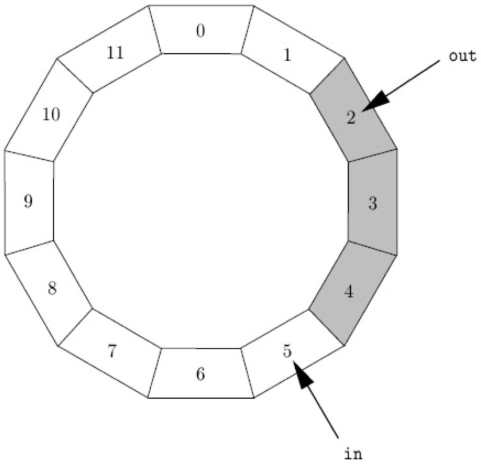


Figure 7.1: A ring buffer with 12 slots - `in` points to the next write location and `out` to the next read location.

Formal Specification

The implementation of the ring buffer is split up into an interface, BufferSpec (Listing 7.1), and a class, Buffer (Listing 7.2), implementing that interface. In the class BufferSpec, the public interface of a ring buffer and most of its behavioural specification is implemented. In that specification most of the assertions and specification expressions *Modern Jass* supports, are used:

- Model variables are used to model the two pointers of the ring buffer, and the actual data structure that backs up the ring buffer.
- Invariants, that reference model variables and methods of the current type, are used in the specification.
- Capturing and comparing pre-state values in the post-state with the help of @Old.
- The @Pure marker annotation is added to some methods, marking them as side-effect free.
- Method specifications with multiple specification cases and exceptional behaviour, as well as, flyweight annotations for simpler specifications.

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

  @Pure
  @Post("@Result == mTheBuffer.length")
  public int size();

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

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

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

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

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

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

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

Listing 7.1: Declaration and specification of the ring buffer interface.

The class Buffer implements the ring buffer specification, which means it inherits all methods and contracts. Especially model variables must be named here because they behave similar to abstract methods as they must be ‘implemented’ by the subtype. In the case of model variables, implementing is to have a corresponding @Represents-annotation.

The @Represents annotations, a flyweight invariant, and a constructor specification is shown in Listing 7.2. Because these are the only specification elements of the Buffer class, the listing has been shortened.

```
@RepresentsDefinitions({
  @Represents(name = "mTheBuffer", by = "buf"),
  @Represents(name = "mStoreIndex", by = "index"),
  @Represents(name = "mExtractIndex", by = "out") })
public class Buffer<T> implements BufferSpec<T> {

  protected int index, out;

  @NonNull protected Object[] buf;

  @SpecCase(
    pre = "capacity > 0",
    post = "size() == capacity")
  public Buffer(int capacity) {
    buf = new Object[capacity];
  }

  ...

}
```

Listing 7.2: Implementation of the ring buffer interface (shortened).

To write the code of this case study, the Eclipse plugin, introduced in Section 6.5, has been used. To check that the buffer works and that the specification is enforced at runtime, a JUnit test has been implemented. It calls every method with parameters that match the specification and with parameters violating the specification. In the latter case, the expected result is a pre-condition error, whereas in the former case no errors shall occur. In total, there are 14 test cases, checking every method at least once, and violating every pre-condition at least once. Post-conditions and invariants cannot be violated on purpose because the buffer implementation must be faulty to do so. Still, when executing all test cases, every specification is executed and enforced too.

At runtime, the class files of the ring buffer are dynamically instrumented with the consequence that the class loading process slows down noticeable and class file size increases. In Table 7.1, the sizes of the original and instrumented class files are compared.

	Instrumented	Original
BufferSpec	1872 byte	1739 byte
Buffer	14022 byte	2564 byte

Table 7.1: Comparing class file sizes.

When simply creating a buffer with capacity one, execution takes around 670 ms opposed to only 4 ms without contract checks. When adding and removing 10000 elements, execution takes 1893 ms with, and 24 ms without contract checks. From a pure theoretical performance analysis two main performance impacts can be identified. First, class loading is slowed down because contract annotations are transformed and compiled before their bytecode gets woven into the running program, and secondly, for every contract, contract method executions must be added to the actual called method.

7.3 The Automated Teller Machine

In this case study, the behaviour of a bank communicating with a number of automated teller machines (ATM) is examined. Compared to the ring buffer case study, this case study is more complex and can be seen as the better real world example. Previously, the ATM example has been used in the DFG project *ForMooS*.

Explanation: Bank, ATMs

Although an in-depth introduction of the ATM example can be found in [MORW07], this section wants to convey its basics. The idea of the ATM example is to specify the behaviour of a bank that communicates with a number of automated teller machines. To do so, a bank is modelled as a component that knows of its customer accounts, its ATMs, and its currently active transactions. ATMs and the bank communicate via service interfaces which provide the necessary operations and information. The class diagram in Figure 7.2 shows all classes, methods and fields, defined to model a bank. As one can see in Figure 7.3, an ATM is a component, too. In its fields it stores, to which bank it is connected, the current configuration, and data of a running withdrawal operation. Further, a number of interfaces enable the communication with the bank and the interaction with an user.

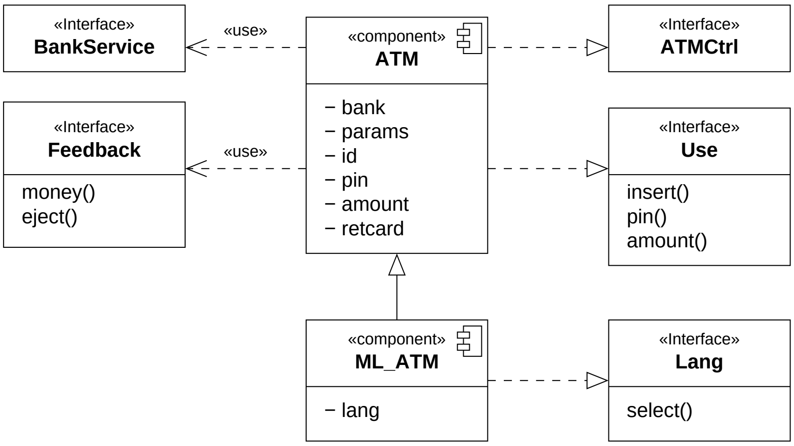


Figure 7.2: Class diagram for the bank [MORW07].

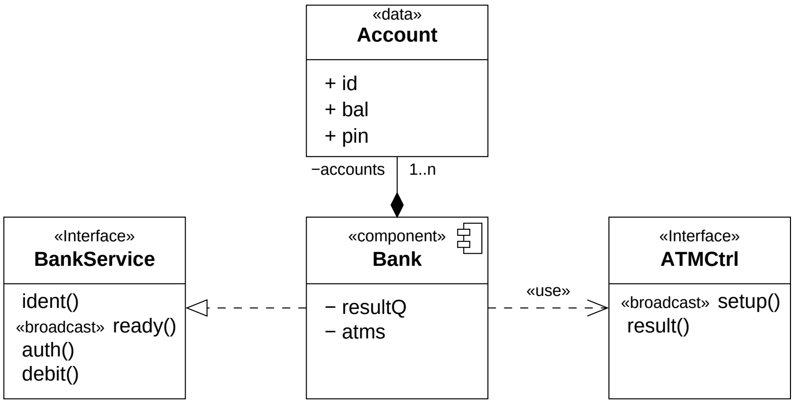


Figure 7.3: Class diagram for an automated teller machine [MORW07].

Formal Specification

Originally, the behaviour of the ATM case study has been specified with the Java Modelling Language (JML), but now the *Modern Jass* annotations have been used. In total, there are 30 classes and interfaces, and altogether 124 specification annotations are used. However, not all of these annotations define behaviour, but they define model variables and side-effect freeness, too. Table 7.2 shows how often the annotation types are used throughout the specification. The ATM example strictly separates the specification from the implementation, so that a lot model variables are used. Further, lots of methods and types are marked as side-effect free, leaving 34 specification annotations that either define method behaviour or class invariants. Almost every specification could be transformed manually from JML into a *Modern Jass* annotation. Still, there are some limitations and defects in the *Modern Jass* tool, so that around five specifications got lost or weakened. Further, the JML based specification used JML model classes, like org.jmlspecs.models.JMLValueSet, as model variables. These classes have been replaced with appropriate counterparts of the Java collections framework.

Annotation	Count
Invariants & Method behaviour	34
Model variables	52
Side-effect freeness	38
	124

Table 7.2: Specifications in the ATM case study.

In order to test the specified ATM system, two withdrawal-operations are simulated. In this scenario a bank with three ATMs is setup, and a user withdraws an amount of 111 at two different ATMs. These withdrawal-operations involve a lot single operations, touching on a fair amount of specifications. Briefly, the bank system must get initialised in a first phase, to perform the withdrawal-operations in a second phase. During the initialisation phase, every ATM sends an ident-signal in order to register itself in the bank system. After that, the overall system enters a state called 'up and running'. Aside from the model variable representations, seven different method specifications must hold in this phase. The second phase, withdrawing money from the ATM, involves checking 13 method specifications. Basically, this phase consists of the steps, authenticating a customer by its card, entering the pin and desired amount of money, checking preconditions and performing the debit, and ejecting the customer-card and money.

Summarising it can be concluded, that the ATM case study could successfully be realised. Still, some bugs and limitations have been brought out, so that the specification got weakened for some types. Starting the ATM system and performing a withdrawal showed that the specified contracts could be enforced at runtime. To further the investigation, a unit test suite, which provides a context to safely violate assertions and to check for expected behaviour, should be used. However, due to the complexity of the ATM example and deadline restrictions, such a test suite could not be realised.

Wrapping up all the case studies presented in this chapter, *Modern Jass* has proven to be valuable. It keeps up with the promise of a rich feature set at low integration and maintenance costs. Nevertheless, some software defects are not fixed yet, and performance must improve. In particular handling of model variables is erroneous in some situations and dynamic bytecode instrumentation is currently slow.

Chapter 8

Conclusion

Summarising, it can be said, that although design by contract for Java has been widely studied and different approaches have been implemented, it is still an ongoing field of research. As Java evolves, the possibilities to implement design by contract for Java evolve, too.

Within this thesis, a new approach for design by contract in Java that uses recent Java technologies has been identified, evaluated, implemented, and demonstrated. Basically, the results of this thesis can be divided into a set of specification annotations, that allow to define the behaviour of Java programs, and a DBC-tool, that implements design by contract based on these annotations.

The specification annotations that have been defined in Chapter 5 are first class citizens in the Java world and enable users to specify the behaviour of their programs. The design of these annotation has been influenced by the collaboration with the JML 5 project, in which a very similar set of specification annotations is developed. In future, both sets of annotations are going to merged and exclusively deployed, so that third parties can define behaviour of Java programs, independent of a tool implementation.

The *Modern Jass* tool combines existent Java technologies in a novel way to implement DBC for Java. By utilising the *Annotation Processing Infrastructure*, *Modern Jass* becomes a compiler plug-in, and is enabled to validate contracts are part of the standard compilation process. At runtime, *Modern Jass* performs bytecode instrumentation, using the *Java Instrumentation API*, to weave contract code into the running program.

Both integration points, annotation processing and bytecode instrumentation, are provided by the standard Java platform, so that no external tool, e.g. a pre-processor, is required. Consequently, every IDE and build tool, that supports the standard Java platform, is capable of running *Modern Jass* without any additional efforts.

Besides seamless integration, the *Modern Jass* tool uses the Java compiler (JSR 199) for contract validation and enforcement, keeping the maintenance costs at a minimum as Java evolves. This is, because the compiler provided by the standard Java platform is used, instead of maintaining a custom compiler or pre-processor.

	Modern Jass	Jass	Common JML Tools	jContractor	Contract4J
Invariants	●	●	●, including history constrains	●	●
Pre- / Post-Conditions	●, including exceptional behaviour	●	●, including exceptional behaviour	●, including exceptional behaviour	●
Return	●	●	●	●, variable RESULT in contract method	●
Old	●, stores references	●, depends on clone & java.lang.Cloneable	●, stores references	●, stores references	●, stores references
Quantifier	●	●	●	●	–
Loop-Invariants	–	●	●	–	
Model-Variables	●	–	●	–	–
Inheritance	●	–, refinement only. Not for interfaces.	●	–	●, very limited
Contract Validation	Annotation Processor & Java Compiler	Custom parser	Custom parser	Java Compiler	–, evaluation engine at runtime only
Input Method	Java 5 Annotations	Comment-based	Comment-based	Java Source Code and methods	Java 5 Annotations
Back-End	Bytecode-Instrumentation	Pre-Processor	Compiler	Bytecode-Instrumentation	Aspect Oriented Programming

Table 8.1: Overview of the Design by Contract tools, including *Modern Jass*.

In Chapter 7 different case studies are introduced, all proving the specification annotations and the tool implementation being valuable and powerful. *Modern Jass* tries to align with the Java Modelling Language and existent case studies have been converted from JML successfully. Even though currently some limitations exist, the future development of Java is likely to weaken or remove them.

In addition to the set of annotations and the *Modern Jass* tool, a converter, which translates JML assertions into corresponding Java 5 annotations, has been implemented. Still being a prototype, the overall concept has been proven to be valuable. In particular, after fully joining the sets of specification annotations from the JML 5 and *Modern Jass* project, such a tool will be useful, as JML users might switch to an annotation based tool, like *Modern Jass*.

In Section 3.3.4, page 48, an overview of the DBC-tools, which have been examined in this thesis, is presented. Subsequently, this overview is summarised with a new column listing the properties of *Modern Jass*. Table 8.1 shows that *Modern Jass* aligns with the Java Modelling Language. Still, JML is more powerful as only a subset of its features has been dealt with in this thesis. Finally, it should be mentioned that *Modern Jass* can be combined with other DBC-tools because it has no special requirements or dependencies.

8.1 Future Work

Although a rich set of specification annotations and a fully functioning DBC-tool could be implemented, there is still work to do. This is true for both, specification annotations and the *Modern Jass* tool.

Regarding the specification annotations, the JML 5 project and *Modern Jass* should fully join soon. This will ensure an uniform way to specify program behaviour with Java annotations. It is also to investigate if users wish for more flyweight annotations or not.

A further extension of the annotation set might also be considered in respect to assertion types of JML that are not supported or checked in *Modern Jass*. Such might be history constraints, the assignable clause, or checking side-effect freeness.

Further, the progress of JSR 305 and JSR 308 must be monitored so that their results is reflected in the specification annotations. Such might be new annotation targets (e.g. annotations for loops) or specification annotations that are going to be part of the official Java platform (e.g. `@NonNull`).

When it comes to the *Modern Jass* tool, performance of bytecode instrumentation must be improved. Currently, specification annotations are transformed and validated again, before their bytecode can be woven into the running program. In future, the bytecode, that is currently a by product of the contract validation, should be reused instead being recreated during bytecode instrumentation.

In addition to performance improvements, it is to investigate how the code completion infrastructures from the open source IDEs Eclipse and NetBeans can be used in *Modern Jass*. The goal is to ease writing specification by proposing completion and to offer quick fixes when specifications are erroneous.

Currently, the JML developers discuss the future of the JML tools, and, having in mind that JML might switch to Java 5 annotations, the approach *Modern Jass* uses should be proposed.

8.2 Related Work

In her master's thesis, Kristina Boysen works on an annotation based version of JML and the Common JML tools. The JML 5 project and *Modern Jass* are going to merge the specification annotations into an uniform set of annotations.

The *OVal* project (object validation framework) provides a framework to dynamically validate objects and, in combination with aspect oriented programming, to use design by contract in Java. *OVal* uses Java annotations to express assertions that are similar to flyweight annotations in *Modern Jass* [OVa].

The tool *FindBugs* analyses Java programs and is capable to identify potential programming problems like referring to the null reference. To assist this analysis, *FindBugs* offers a set of annotations that are used to specify the behaviour of programs [HP07].

There are numerous tools that implement design by contract for Java and only a few of them have been introduced in this thesis. More tools that implement DBC for Java are listed in the Wikipedia encyclopedia [Con07].

Glossary

DBC Design by Contract. A software development methodology which treats two software components as client and servant having a contract with each other, p. iii.

IDE An Integrated Development Environment (IDE) is a programming environment combining several tools in one. For instance, an IDE might consist of an editor, a debugger, source control, project management, and so one, p. 3.

Oak Forerunner of the Java programming language which was develop by James Gosling and his team in the early 90s, p. 7.

GPL The General Public License (GPL) is an open-source license, p. 7.

JSR A Java Specification Request is, in terms of the Java Community Process, a request for some enhancement or addition to the Java programming language or the Java platform, p. 8.

HTML The Hyper Text Markup Language is used define the content and layout of web-sites, p. 9.

API The abbreviation for Application Programming Interface which is an interface of an application or library, making it reusable in different contexts, p. 9.

JVM The Java Virtual Machine is a platform specific program which executes Java bytecode, p. 20.

ISV Indepentend Software Vendors are third party software developers. Usually, the term ISV is used when talking about others extending a program or using an API, p. 21.

JDT The Java development tools is an Eclipse project which provides tools to support Java development. For instance, a compiler, an editor, and a debugger are part of the JDT, p. 21.

classloader A classloader is used to create a class object (java.lang.Class) from an accordent data source (e.g. a file or stream) and to load it into the Java Virtual Machine. The base class for all classloaders is java.lang.ClassLoader which must be extended in order to provide a custom classloader, p. 23.

signature The signature of a method is made up from the name, the number and types of parameters, and its return type, p. 36.

AOP Stands for Aspect Oriented Programming. A programming paradigm which addresses crosscutting concerns like logging or event handling. A prominent AOP implementation based on Java is AspectJ, p. 40.

UML The Unified Modelling Language (UML) is graphical, general-purpose notation for systems in computing. The UML defines serval diagram types, e.g. class-diagrams, sequence-diagrams, or usecase-diagrams, p. 42.

stack trace Every time a method calls another method, a new frame is pushed onto an execution stack, and every time a method finished its execution, its frame is removed from the stack. A stack trace is a trace of method execution frames from the current method to the first method, p. 46.

opcode An opcode is an instruction which can be executed by the Java Virtual Machine. In Java, method bodies are translated into a number of opcodes, and stored in class files., p. 99.

Eclipse Eclipse is an open platform to development arbitrary applications. The Eclipse Java development toolkit (JDT) is the most famous Eclipse based application, p. 100.

DFG Deutsche Forschungsgemeinschaft (German Research Foundation). A funding organisation that supports university and research institutions, p. 111.

Bibliography

[Art06] J. Arthorne. *Compiler API (JSR 199).* The Eclipse Foundation, 2006. https://bugs.eclipse.org/bugs/show_bug.cgi?id=154111 - Retrieved 2006-12-28.

[Asp06] Aspect Research Associates. *Contract4J5 – Contracts Using Java 5 Annotations*, September 2006. http://www.contract4j.org/contract4j/c4j5 - Received 2006-11-06.

[Bar99] D. Bartetzko. Parallelität und Vererbung beim 'Programmieren mit Vertrag'. Master's thesis, Universität Oldenburg, 1999.

[BCC$^+$05] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer*, volume 7, pages 212–232, 2005.

[Ber06] J. Bergström. *C4J - Design By Contract for Java*, 2006. http://c4j.sourceforge.net/ - Retrieved 2007-01-31.

[BFMW01] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with Assertions. In *Proceedings of the First Workshop on Runtime Verification (RV'01)*, July 2001.

[Blo99] J. Bloch. *JSR 41: A Simple Assertion Facility*, November 1999. http://jcp.org/en/jsr/detail?id=41.

[Blo02] J. Bloch. *JSR 175: A Metadata Facility for the JavaTM Programming Language*, 2002. http://jcp.org/en/jsr/detail?id=175#orig - Retrieved 2007-02-09.

[BV06] TIOBE Software BV. TIOBE Programming Community Index for October 2006. Internet, October 2006.

[Byo] J. Byous. Java Technology: The Early Years. *Feature Stories About Java Technology.* http://java.sun.com/features/1998/05/birthday.html - Retrieved 2007-02-26.

[CCC+06] Y. Cheon, C. Clifton, D. Cok, J. Kiniry, G. T. Leavens, P. Müller, and C. Ruby. *JML Reference Manual.* Iowa State University, Department of Computer Science, 2006. `http://www.cs.iastate.edu/~leavens/JML/jmlrefman/jmlrefman.html` – Retrieved 2006-11-16.

[CE06] D. Coward and M. Ernst. *JSR 308: Annotations on Java Types*, 2006. `http://jcp.org/en/jsr/detail?id=308` - Retrieved 2007-03-05.

[CLSE05] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model Variables: Cleanly Supporting Abstraction in Design By Contract. Technical report, Software, Practice & Experience, May 2005.

[Con07] Wikipedia Contributors. Design by contract — Wikipedia, The Free Encyclopedia. *Wikipedia, The Free Encyclopedia.*, April 2007. `http://en.wikipedia.org/w/index.php?title=Design_by_contract&oldid=121309037` - Retrieved 2007-04-10.

[Dar06] J. Darcy. *JSR 269: Pluggable Annotation Processing API*, November 2006. `http://jcp.org/en/jsr/detail?id=269`.

[Dio04] F. Diotalevi. Contract enforcement with AOP – Apply Design by Contract to Java software development with AspectJ. *IBM – developerWorks*, 2004. `http://www-128.ibm.com/developerworks/library/j-ceaop/` - Received 2006-11-07.

[ECM06] ECMA International. *Eiffel: Analysis, Design and Programming Language*, 2nd edition, 2006. Standard ECMA-367.

[Eli02] A. Eliasson. Implement Design by Contract for Java using dynamic proxies. *JavaWorld.com*, February 2002. `http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-dbcproxy.html` - Retrieved 2007-01-21.

[Fir94] FirstPerson Inc. *Oak Language Specification*, 1994. `https://duke.dev.java.net/green/OakSpec0.2.ps` – Retrieved 2006-11-13.

[Fre06] L. Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. *Eclipse Magazin*, 5, 2006.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addision-Wesley, third edition edition, 2005.

[GvdA02] N. Gafter and P. von der Ahe. *JSR 199: JavaTM Compiler API*, October 2002. `http://jcp.org/en/jsr/detail?id=199`.

[Har06] W. Harley. *Bug Report - [jsr269] Need interfaces between jdt compiler and jsr269 impl.* Eclipse, October 2006. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=160773` - Retrieved 2007-01-29.

[HGH06] G. Horen, J. Garms, and W. Harley. *Java Annotation Processing (APT) in the Eclipse JDT.* Bea Systems, March 2006. `http://www.eclipse.org/jdt/apt/Eclipsecon2006.ppt` - Retrieved 2006-11-12.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[HP07] D. H. Hovemeyer and W. W. Pugh. *FindBugs^{TM} Manual.* University of Maryland, April 2007. `http://findbugs.sourceforge.net/manual/index.html` - Retrieved 2007-04-07.

[Jas05] Jass - Correct System Design Group University of Oldenburg. *Jass - Project Website*, September 2005. `http://csd.informatik.uni-oldenburg.de/~jass/` – Retrieved 2006-11-06.

[Jav98] Java Community Process. Java Community Process Program, 1998. `http://jcp.org/en/home/index` - Retrieved 2006-12-28.

[KA05] M. Karaorman and P. Abercrombie. jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation. *Formal Methods in System Design*, 27(3):275–312, November 2005.

[KHB98] M. Karaorman, U. Hölzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design By Contract. Technical report, Department of Computer Science, University of California, 1998.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[LW93] B. Liskov and J. Wing. Family Value: A Behavioural Notion Of Subtyping. Technical Report MIT/LCS/TR-562b, 1993.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addisom-Wesley, 1996.

[Mey92] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[MORW07] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a Formal Method into a Software Engineering Process with UML and Java. Under consideration for publication in Formal Aspects of Computing, 2007.

[Net06] NetBeans Wiki. *FaqApt - Can I run an annotation processor from my project?*, September 2006. `http://wiki.netbeans.org/wiki/view/FaqApt` - Retrieved 2007-01-29.

[OVa] *OVal - the object validation framework for Java^{TM} 5 or later.* http://oval.sourceforge.net/ - Retrieved 2007-04-07.

[Pug06] W. Pugh. *JSR 305: Annotations for Software Defect Detection*, 2006. http://jcp.org/en/jsr/detail?id=305 - Retrieved 2007-03-06.

[Sch06] M. Schnaidt. Runtime-Checking von JML-Spezifikationen mit Jass. Master's thesis, Universität Oldenburg, 2006.

[Suna] *Class description – java.lang.reflect.Proxy.* http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html - Retrieved 2006-03-24.

[Sunb] Sun Microsystems Inc. *Doclet Overview.* http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/doclet/overview.html - Retrieved 2007-01-25.

[Sunc] Sun Microsystems Inc. *Inherited (Java 2 Platform SE 5.0).* http://java.sun.com/j2se/1.5.0/docs/api/java/lang/annotation/Inherited.html - Retrieved 2007-02-10.

[Sund] Sun Microsystems Inc. *Method 'clone' – java.lang.Object.* http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#clone() - Retrieved 2006-11-06.

[Sune] Sun Microsystems Inc. *Package description – java.lang.instrument.* http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html#package_description - Retrieved 2006-02-03.

[Sun04a] Sun Microsystems Inc. *Annotation Processing Tool (apt)*, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html - Retrieved 2006-09-10.

[Sun04b] Sun Microsystems Inc. *Annotations*, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html - Retrieved 2007-25-01.

[Sun06a] Sun Microsystems, Inc. *Free and Open Source Java*, December 2006. http://www.sun.com/software/opensource/java/ - Retrieved 2006-12-28.

[Sun06b] Sun Microsystems, Inc. *javac - Java programming language compiler*, 2006. http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html#processing - Retrieved 2007-01-23.

[Sup01] *Support For 'Design by Contract', beyond "a simple assertion facility"*, April 2001. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4449383 - Retrieved 2007-02-26.

[The06] The AspectJ Team. *The AspectJ Programming Guide*, 2006. http://www.eclipse.org/aspectj/doc/released/progguide/index.html – Retrieved 2006-11-13.

[Wam06a] D. Wampler. AOP@Work: Component design with Contract4J. *IBM – developerWorks*, 2006. `http://www-128.ibm.com/developerworks/java/library/j-aopwork17.html` - Received 2006-11-07.

[Wam06b] D. Wampler. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, 2006.

[Wid06] T. Widmer. Unleashing the Power of Refactoring. *Eclipse Magazine*, 2006.

[Wik07] Wikibooks. *D Programming – From Wikibooks, the open-content textbooks collection*, March 2007. `http://en.wikibooks.org/w/index.php?title=D_Programming&oldid=807825` – Retrieved 2007-04-17.