Adapting the Java Modeling Language for Java 5 Annotations

Kristina B. Taylor and Johannes Rieken and Gary T. Leavens

TR #08-06
April 2008

Keywords: JML, Modern Jass, Java, Specification, Annotation.

2008 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Adapting the Java Modeling Language for Java 5 Annotations

Kristina B. Taylor[1], Johannes Rieken[2], and Gary T. Leavens[3]

[1] Iowa State University, Ames IA 50011, USA*
[2] IBM Rational Zurich Research Lab, 8001 Zurich, Switzerland**
[3] University of Central Florida, Orlando FL 32816, USA

Abstract. The Java Modeling Language (JML) is a formal specification language for Java that allows to express intended behavior through assertions. Currently, users must embed these assertions in Java comments, which complicates parsing and hinders tool support, leading to poor usability. This paper describes a set of proposed Java 5 annotations which reflect current JML assertions and provides for better tool support. We consider three alternative designs for such annotations and explain why the chosen design is preferred. This syntax is designed to support both a design-by-contract subset of JML, and to be extensible to the full language. We demonstrate that by building two tools: Modern Jass, which provides almost-native support for design by contract, and a prototype that works with a much larger set of JML.

1 Introduction

The Java Modeling Language (JML) is a specification language for Java that uses Java comments to express its assertions throughout the body of the code. This technique allows assertions to appear anywhere throughout the code but requires quite a bit of preprocessing and does not provide any native code completion support in integrated development environments (IDEs) such as Eclipse. Since JML's inception, the designers of Java have added new language constructs called *annotations* to provide language metadata alongside the Java language [3]. We decided to investigate whether there was a way to express JML assertions using these annotations.

Loosely speaking, Java 5 annotations are type-safe comments. Their definition is similar to the interface type and they can be used to add arbitrary metadata to Java statements such as types, fields, methods, method parameters, and local variables. To define an annotation the `interface` keyword, preceded by the `@`-sign, must be used. An annotation body can specify attributes to allow varying instances of that annotation. Figure 1 shows the declaration and instantiation of a possible requires-assertion.

The declaration of the annotation shows one attribute, named *assertion*, of the string type. Attributes allow all primitive types, the *class* and *enum* type,

* Master's thesis[1]
** Master's thesis at the University of Oldenburg, Germany[2]

```
public @interface Requires {
  String assertion();
}

/* ... */

@Requires(assertion = "amount > 0")
abstract int doWithDraw(int amount);
```

Fig. 1. Declaration and exemplary use of an annotation

other annotations, and arrays as their data type. Annotating the *doWithDraw* method is done by stating the annotation, preceded by the @-sign, and assigning a value to the attribute.

In order to motivate the use Java 5 annotations to specify behavior, consider an example of the current JML syntax in Figure 2, inspired by one in the JML Reference Manual [4]:

```
public abstract class IntList {

  //@ public model non_null int [] elements;

  //@ ensures \result == elements.length;
  public abstract /*@ pure @*/ int size();

  /*@ signals (IndexOutOfBoundsException e)
    @         index < 0 || index >= elements.length;
    @*/
  public abstract /*@ pure @*/ int elementAt(int index)
    throws IndexOutOfBoundsException;

}
```

Fig. 2. Base Example

Notice that the assertions in Figure 2 must appear both in `//` or `/*` style comments and these comments must start with an at-sign (`@`) as their first character. These special comments are recognized by and processed by JML's tools. This could be confusing to new JML users who are familiar with the `@Identifier` syntax for Java annotations. This example could be translated into Java 5 annotations in several ways, from an all-encompassing annotation to many smaller annotations. Several criteria were considered when judging how usable these annotations were to the users and designers:

Consistency Annotations may appear in several places, and they must be written the same way in each place.
Readability Annotations must appear without too much syntax around them so that readers can clearly find and easily read the predicate.
Usability Annotations must be flexible in how they are used: intuitive and straightforward for new users while being fully expressive for veteran users.
Extensibility Annotations must be able to be extended without disrupting the current syntax.

Thus, in this paper, we aim to answer two questions:

– Is there a way to write design-by-contract annotations such that they are consistent, readable, usable, and extensible?
– Is it possible that multiple design-by-contract implementation languages can successfully use a subset of these annotations as a language?

The rest of the paper explores related projects, explains the three approaches in more detail, and evaluates them using the criteria above.

2 Related Work

Both JSR 305 [5] and JSR 308 [6] discuss the idea of simple Java annotation code checking constructs, by suggesting such annotations as `@NonNull`, `@Nullity`, and `@Pure`. JSR 308 aims to allow Java annotations in more places (specifically, type expressions), and JSR 305 aims to create a standard set of annotations to assist code checking tools. However, neither of these propose adding annotations for design-by-contract clauses such as `ensures`.

XVP [7] uses Java 5 annotations to manage assertions in a declarative form. While it improves on JML by representing assertions using reflection, it only defines two annotations (`@Constraint` and `@Pure`). The authors explain that this is to allow for any constraint language to be substituted in place of JML. However, this makes the constraints difficult to read; it is hard to find the starts and ends of separate clauses without clear string formatting.

Contract4J [8] goes further by splitting the contracts into three annotations: `@Pre` (preconditions), `@Post` (postconditions), and `@Invar` (invariants). However, its expression language is not as expressive as JML, as there are no mechanisms to denote `assignable`, `pure`, or `model` constructs.

OVal with AspectJ [9] defines many annotations similar to JML, such as `@NonNull` and the purity indicator `@IsInvariant`. It allows the user to select the expression language inside of the annotations from five different expression languages (BeanShell, Groovy, JavaScript, MVEL, and OGNL) and also comes with a translator from Enterprise JavaBeans 3.0 (EJB3) to OVal annotations, allowing users to put constraints on database objects. Again, while the annotations and expressions as a whole cover more than Contract4J, they are not as expressive as JML, leaving out full `assignable` support and `model` constructs altogether.

3 Three Approaches

This section discusses three different approaches to embedding a specification language in Java annotations.

3.1 The Single Annotation

One extreme technique is to just put all of the assertion inside one grand annotation. As mentioned previously, XVP takes this approach by using the encompassing `@Constraint` annotation to hold the entirety of a JML expression[4]. An example is given in Figure 3.

```
@JML("public model non_null int [] elements;")
public abstract class IntList {

  @JML("ensures #result == elements.length;")
  public abstract @JML("pure") int size();

  @JML("signals (IndexOutOfBoundsException e)"
             + "index < 0 || index >= elements.length;")
  public abstract @JML("pure") int elementAt(int index)
    throws IndexOutOfBoundsException;

}
```

Fig. 3. Single Annotation Example

3.2 The Parameter Annotation

The other extreme technique is to separate every single part of the annotation into its parts inside of the annotation. This was the original design of the JML annotations, but later analysis will show that going fully by this philosophy causes problems. The same example in the previous section is modified for this approach in Figure 4.

3.3 The Clausal Annotation

Since the two previous techniques are opposite of each other, it makes sense to try to compromise between them. The driving design decision behind this approach is to give as much work to the current parser as possible without losing the type

[4] To avoid confusion with JML's `constraint` keyword, this chapter will use `@JML` as a replacement for `@Constraint`

```
@Model(visibility = Visibility.PUBLIC, nonnull = true,
       value = "int [] elements;")
public abstract class IntList {

  @Ensures("#result == elements.length;")
  public abstract @Pure int size();

  @Signals(type = "IndexOutOfBoundsException", ident = "e",
           value = "index < 0 || index >= elements.length;")
  public abstract @Pure int elementAt(int index)
    throws IndexOutOfBoundsException;

}
```

Fig. 4. Parameter Annotation Example

of the assertion. This means that every clause that can occur by itself has an annotation with that name and retains that structure throughout all expressions, even within strings. Again, the same example in previous sections is modified for this approach in Figure 5.

```
@Model("public @NonNull int [] elements;")
public abstract class IntList {

  @Ensures("#result == elements.length;")
  public abstract @Pure int size();

  @Signals("(IndexOutOfBoundsException e)"
         + "index < 0 || index >= elements.length;")
  public abstract @Pure int elementAt(int index)
    throws IndexOutOfBoundsException;

}
```

Fig. 5. Clausal Annotation Example

4 Discussion

Now we evaluate each approach based on the criteria stated above.

4.1 Consistency

The single annotation approach is the most consistent of the three, since the annotation encompasses the entire grammar. This becomes slightly less consistent when using syntactic sugar like `@Pure`, since there are now two ways of specifying a single `pure` assertion on a method. The clausal approach is slightly less consistent than the single annotation approach, especially when it comes to more advanced features of the language, such as a specification case[5]. However, since the clausal approach requires that the content inside the strings is the same and thus does not split up assertions, it is much more consistent than the parameter approach.

4.2 Readability

The clausal approach is the most readable of the three, since it strikes a compromise between the two extreme approaches. The single annotation approach obfuscates the assertion meaning by forcing the reader to read past `@JML(..)` to figure out the actual assertion type. The parameter approach separates every component of the predicate into different pieces in the annotation, eliminating the sentence-like structure of the JML expressions. The `signals` clause in the example illustrates this point well, since it translates into

```
@Signals(type = "IndexOutOfBoundsException", ident = "e",
         value = "index < 0 || index >= elements.length;")
```

This translation becomes even more complicated for modifiers that turn into their own language constructs, such as `model`. The parameter approach transforms the `model` statement into

```
@Model(visibility = Visibility.PUBLIC, nonnull = true,
       value = "int [] elements;")
```

It is unreasonable to create fields for all of the other modifiers that can attach to `model` since even slight language redesigns would require changes in this annotation.

The clausal approach strikes a nice balance between the two, specifying the type of the assertion while not splitting the predicate into pieces. In this approach `signals` becomes

```
@Signals("(IndexOutOfBoundsException e)"
       + "index < 0 || index >= elements.length;")
```

while `model` turns into

```
@Model("public @NonNull int [] elements;")
```

a much shorter and more compact annotation.

[5] For more information on the decisions on the specification case design, see Kristina Taylor's masters thesis [1]

4.3 Usability

The single annotation approach may be more usable by JML veterans, but it does not guide new users on the different elements of the annotations. The parameter approach is the most usable, as it lists in its specification the elements of the annotation, their types, and whether or not they are required. For example, the `@Signals` clause shows the user that while merely a type of exception is required, the user can also specify an identifier and a predicate. The clausal approach only specifies the type of the annotation and nothing about the inner syntax, so it is only slightly more usable than the single annotation approach.

4.4 Extensibility

The single annotation and clausal approaches are the most extensible, since most or all of the language extensions are handled by the parser. Thus, the parser continues to be responsible for the error checking, while the hand-written code only checks the type of the annotation and feeds all relevant information back to the parser. In the parameter approach, a language change requires adding a new annotation component as well as new hand-written code to feed it to the parser, increasing the workload for the tool programmers and the possibility for errors.

4.5 Summary

Using these criteria, we conclude that the single annotation approach is not helpful to new users and that the parameter annotation approach is too complicated. Thus, the clausal annotation approach seems the best compromise to us. In addition, since the clausal annotation forwards much of the work to the parser, it allows different projects, such as the Modern Jass project, to use the basic JML language while implementing their own language enhancements and expression grammars.

5 Tool Integration

This section discusses how we have experimented with this design by giving an overview of two tools that use it: Modern Jass and JML5.

5.1 Modern Jass

Modern Jass is about enabling formal specifications without the cost of custom development tools like pre-processors. To achieve this only Java features and standardized programming interfaces were to be used. The components from which Modern Jass is build, reflect the main requirements a specification tool has to fulfill. Those are to:

– provide a way to write down specifications,

– validate specifications, so they can be evaluated without failure, and
– enforce that the specified behavior holds at runtime, and emit an event in case it does not.

These requirements can be implemented using native Java programming interfaces, integrating seamlessly into every build process, being easy to use, and independent of a development tool. To do this, Modern Jass uses Java 5 annotations [3], the Annotation Processing and Compiler API [10,11], and the Bytecode Instrumentation API [12].

The following lays out the workings of Modern Jass in detail. The behavior of a program is specified using a set of Java 5 annotations, like the JMLAnnotations. Currently, Modern Jass *understands* a subset of the JMLAnnotations which contains the heavyweight specification case, invariants, and model variables as well as lightweight specifications like the `@Requires`, `@Ensures`, and `@NonNull` annotations. Before expounding on the Modern Jass annotation processor and bytecode instrumenter, the corresponding programming interfaces are to be introduced.

The Annotation Processing API emerged from JSR 269 and was added to the Java platform in its sixth major release (Java 6). It enables third party developers to provide compiler plugins, called annotation processors, to discover and validate the use of annotations. Access to all annotations and a signature AST is provided, and an annotation processor can issue an error or warning if an annotation is not used as expected. For example, if a specification refers to a variable which is out of scope, the annotation processor will create a compile error. Those errors will appear like *ordinary* compile errors, hence, be familiar to most programmers. Having a framework to process annotations, raises the question how invalid annotation values can be detected? Figure 6 shows a simple specification which requires an index to be positive. However, the specification is faulty, as it refers to the variable *imdex* instead of *index*[6]. A part of specification-

```
@Requires("imdex >= 0")    // wrong!
public abstract int elementAt(int index)
  throws ArrayIndexOutOfBoundsException;
```

Fig. 6. A faulty specification (referring to an invalid variable).

validation is to detect such errors and report them to the end-user. Instead of having a parser and code analyzer Modern Jass delegates this task to the compiler using the Java Compiler API. This API (added via JSR 199 to Java 6), allows one to program

The last requirement is to enforce the specification during runtime. This requires a bytecode representation of the specified behavior that can be exe-

[6] Note that the Java compiler will not detect such errors as it just checks the syntax (proper use of the string data type in this case) and not the actual semantics of the annotation (referring to a method parameter).

cuted during program execution. To accomplish this, the Java Compiler API is used to produce bytecode for runtime checking. When successfully compiling specification code, bytecode is generated. This bytecode is used by a Java Instrumentation Agent and gets inserted, at runtime, while the corresponding classes are loading into the Java virtual machine.

Summarizing, it can be said that using Java 5 annotation to specify the behavior of programs establishes a new way how specification tools can be written. It allows for a much more native tool implementation which lowers the burden for novice users, pleases veterans, and does not depend on a compiler implementation, build-process, or IDE.

5.2 JML5

While Modern Jass uses a subset of the annotation syntax described above, JML5 uses all of its features. JML5 is based on the current JML compiler, JML2, and so represents a different route to implementing annotation-based specification languages. In JML2, abstract syntax trees (ASTs) are made up of classes representing Java constructs (such as methods or fields) mixed in with JML constructs (such as requires and ensures). JML5 works with these custom ASTs without using the Annotation Processing API, and instead extends the JML2 parser and tree walkers. The presence of visitor methods in all of the necessary JML2 classes made this process easy since it allows a single visitor class to walk through each element of the AST.

Parsing in JML5 happens in several phases. First, the compiler parses the code just like Java code, ignoring the content inside of the strings of any JML annotation it encounters. Second, it goes back to parse the strings and inserts them into the previously created AST. In order to create these ASTs from the strings in the annotations, we used a custom ANTLR parser-generator [13] to create the parsing code for us. Every annotation (such as `@Requires` or `@Ensures`) has a method in the parser, making it easy to call directly from the visitor class. So, for example, adding a `requires` clause to a method in the JML AST involves first calling the parser to get the AST for the clause, and then adding the resulting `JmlRequiresClause` to the set of clauses in the `JmlMethodDeclaration` container node.

6 Conclusion

In the introduction, we introduced two questions which this paper would hopefully answer:

– Is there a way to write design-by-contract annotations such that they are consistent, readable, usable, and extensible?
– Is it possible that multiple design-by-contract implementation languages can successfully use a subset of these annotations as a language?

We have demonstrated that the clausal annotation approach is consistent, readable, usable, and extensible, making it the best choice for our new syntax. We have also demonstrated two tools, Modern Jass and JML, that use this syntax, showing that multiple design-by-contract tools based on Java can agree on a single annotation structure. Adoption of this structure by other Java-based tools would make it easier to switch from one tool to another and immediately understand the language between two tools.

This opens up several avenues for future work. One is to expand the above set of annotations to include all of the features of both JML and Modern Jass. Another is to include other design-by-contract languages for Java in this effort, making the above language even more standard for the community. Collaborations like these are necessary in order to come up with a standard design-by-contract syntax for Java and make formal methods for languages more accessible and easier to use for a wider audience.

References

1. Taylor, K.B.: A specification language design for the Java Modeling Language (JML) using Java 5 annotations. Master's thesis, Iowa State University (2008)
2. Rieken, J.: Design by Contract for Java - Revised. Master's thesis, Universität Oldenburg (2007)
3. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition. The Java Series. Addison-Wesley, Boston, Mass. (2005)
4. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. Available from `http://www.jmlspecs.org` (October 2007)
5. Sun Microsystems, Inc.: JSR 305: Annotations for software defect detection. From `http://jcp.org/en/jsr/detail?id=305` (Date retrieved: March 19, 2008) (2006)
6. Sun Microsystems, Inc.: JSR 308: Annotations on java types. From `http://jcp.org/en/jsr/detail?id=308` (Date retrieved: March 19, 2008) (2007)
7. Royer, M., Alagić, S., Dillon, D.: Reflective constraint management for languages on virtual platforms. Journal of Object Technology **6**(10) (Nov.-Dec. 2007) 59–79
8. Wampler, D.: Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. In Coady, Y., Lorenz, D.H., Spinczyk, O., Wohlstadter, E., eds.: Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Published as University of Virginia Computer Science Technical Report CS–2006–01 (2006) 27–30
9. Thomschke, S.: OVal – the object validation framework for Java 5 or later. From `http://oval.sourceforge.net/` (Date retrieved: March 19, 2008) (2007)
10. Darcy, J.: JSR 269: Pluggable Annotation Processing API. From `http://jcp.org/en/jsr/detail?id=269` (November 2006)
11. Gafter, N., von der Ahe, P.: JSR 199: JavaTM Compiler API. From `http://jcp.org/en/jsr/detail?id=199` (October 2002)
12. Sun Mircosystems Inc.: Package description – java.lang.instrument. From `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html#package_description` - (Date Retrieved 2006-02-03)
13. Parr, T.J., Quong, R.W.: ANTLR: A predicated-LL(k) parser generator. Software—Practice & Experience **25**(7) (1995) 789–810