Design of a JMLdoclet for JMLdoc in OpenJML

2016

Arjun Mitra Reddy Donthala
University of Central Florida

DESIGN OF A JMLDOCLET FOR JMLDOC IN OPENJML

by

ARJUN MITRA REDDY DONTHALA
B.S. GITAM University, 2014

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2016

ABSTRACT

The Java Modeling Language (JML) is a behavioral interface specification language designed for specifying Java classes and interfaces. OpenJML is a tool for processing JML specifications of Java programs. To facilitate viewing of these specifications in a user-friendly manner, a tool JMLdoc was created. The JMLdoc tool adds JML specifications to the usual Javadoc documentation. JMLdoc is an enhancement of Javadoc that adds to the Javadoc documentation the JML specifications that are present in the source code. The JMLdoc tool is a drop-in replacement for Javadoc, with additional functionality and additional options. The current design of JMLdoc uses the standard Javadoc's doclet. The current design lacks the provision for doclet extensions, unlike Javadoc. This thesis proposes a new design which is more aligned with the design of Javadoc and its provision for doclet extensions by implementing a JMLdoclet: a new doclet for OpenJML with support for JML elements. The new design makes JMLdoc independent of Javadoc's internals. This way maintenance is reduced as Javadoc evolves. The new design also combines specifications from inheritance and refinements and presents the complete JML specification to the user. This new doclet based design will be more maintainable and easier to extend.

I dedicate this thesis to my parents Dr. Raghunath Reddy and Sridevi. I hope that this achievement will complete the dream that you had for me for all those many years ago when you chose to give me the best education you could.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my sincere thanks to those who helped me with my work during the course of my Master's program. First and foremost, Dr. Gary T. Leavens for giving me this opportunity to pursue research under him and for his guidance, patience and support throughout this research and my course of study at University of Central Florida. His words of support and encouragement have always motivated me. I would also like to thank my committee members: Dr. Sumit Kumar Jha, for his guidance, patience, support and understanding towards me and also Dr. Damla Turgut for her guidance and support. I would additionally like to thank Dr. Mostafa Bassiouni for his guidance, support and his excellent way of teaching, and Dr. Rochelle Elva for her encouragement to pursue research, for her guidance and support throughout my course of study. Finally, I would also like to thank my family for showing me support throughout the Master's course of study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AOP	Aspect Oriented Programming
BISL	Behavioral Interface Specification Language
COFOJA	Contracts for Java
DBC	Design by Contract
IDL	Interactive Data Language
JDK	Java Development Kit
JML	Java Modeling Language
SMT-LIB	Satisfiability Modulo Theories Library

CHAPTER ONE: INTRODUCTION

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) [11] which was designed in order to specify Java modules. It is based on the Design by Contracts (DBC) approach proposed by Bertrand Meyer and also based on the model-based specification approach of the Larch family of interface specification languages [3].

Introduction

JML specifications are written for Java compilation units containing classes and interfaces. The behavioral features of classes and interfaces can be specified using JML specifications. These classes and interfaces will be referred as Java modules. The behavioral features of Java modules are bidirectional to the client and module. JML specifications can be designed as annotations as well as non-annotations. The specifications are like contracts between the application and the client (end-user).

Java classes contain elements which can represent an object's state and define the interaction of the object with outside world. These elements can be fields, methods or constituents of nested classes or interface declarations. The Queue interface shown in Figure 1 has JML specifications. C-language style comments, beginning with `/*@`, or C++ style comments immediately followed by at-sign, `//@`, are JML annotations. These annotations are just left out as comments when the Java Compiler starts the process compilation, where as in JML the text following the `//@` marker and the text between the `/*@` and `@*/` annotation markers is meaningful to the JML compiler.

A model field such as the Queue in Figure 1, acts as an abstraction and requires no implementation. Model fields are only used for specification purposes; they do not make any sense to the Java compiler. A model field's value is derived from the concrete fields it abstracts from. In Figure 1 model variable *theQueue* is declared using the keyword *instance* to be treated as a Java instance field. The keyword *instance* is used to indicate that each instance of a class that implements the Stack interface has its own copy of the variable *theQueue*. The implementing classes of Queue interface or other interfaces which inherit from Queue interface will have these model instance fields inherited in the specifications. *JMLObjectSequence* is a type of *JMLCollection*, and it is provided in the org.jmlspecs.models package. The Queue in Figure 1 has a type of *JMLObjectSequence*. To specify the initial value of the a model field, the *initially* clause is used, which is followed by the corresponding model field. The specifications also has another model instance field declared called *size*. Other clauses declared after the declaration of field **size** are some of the features presented in JML which provide a mechanism to relate model fields with other concrete fields of objects. The *depends* clause indicates the *size* can change whenever *theQueue* changes i.e. *size* depends on *theQueue*. The *represents* clause indicates the relation between *size* and *theQueue*. Next an invariant condition is specified using the *invariant* clause. Invariants are used to indicate what must be true in each visible state. The *invariant* clause for the Queue interface indicates that *theQueue* should be non-null before and after any method invocations on a *Queue* type.

Lines 16-24 specify the behavioral specifications of the method *dequeue*. The keyword *public normal_behavior* is used for describing the normal behavior of a method. It implicitly states that the specification is intended for the clients and when the preconditions are satisfied it

must return in a normal way without throwing any exceptions. These specifications implicitly use a false signals clause. The *requires* clause specifies the precondition, and the *modifiable* clauses indicates which data can be modified by the method and the *ensures* clause indicates the post condition. According to the contract requirement is the DBC design pattern, the caller i.e. client is required to satisfy the preconditions and the developer is required to make the post conditions hold when the preconditions are met. Any additional behavior of a method can be specified using the keyword *also*. Any exceptional behavior of a method can be described using the keyword *exceptional_behavior*. It is used for indicating any error conditions at which the methods should throw an exception using an explicit *signals* clause.

OpenJML is a preprocessor tool for JML specifications of Java programs. OpenJML is used for parsing and type-checking the specifications in Java programs. It is also used for statically (Compile-Time) or dynamically (Run-Time) checking the validity of specifications. OpenJML is based on the OpenJDK (the open source Java compiler) and makes use of specifications in a program written in JML. OpenJML does not handle the task of proving theorems and checking models by itself. It uses external SMT solvers to handle these tasks. In order to, make SMT solvers parse the JML specifications, they need to be first converted into SMT-LIB format. Once they are converted into the SMT-LIB format, the JML programs are passed to the backend SMT solvers to verify the implied proof problems in the programs. Major SMT solvers such as CVC4, Yices and Z3 are supported by OpenJML. The success of the proofs depends on the following factors- capability of the SMT solver, the particular logical encoding of the code and specifications, and the complexity and style in which the code and specifications are written.

```
//@ model import org.jmlspecs.models.*;                                    //1
                                                                           //2
public interface Queue {                                                   //3
                                                                           //4
      /*@ public model instance non-null JMLObjectSequence theQueue       //5
        @     initially theQueue.isEmpty();                                //6
        @*/                                                                //7
                                                                           //8
      public final static int CAPACITY = 1000;                             //9
                                                                          //10
      //@ public model instance int size;                                //11
                                                                          //12
      //@ public represents size <- theQueue.size();                     //13
                                                                          //14
                                                                          //15
      /*@ public normal_behavior                                         //16
        @    requires !theQueue.isEmpty();                               //17
        @    modifiable size, theQueue                                   //18
        @    ensures theQueue.equals(\old(theQueue.trailer()));          //19
        @   also                                                         //20
        @ public_exceptional_behavior                                    //21
        @    requires theQueue.isEmpty();                                //22
        @      signals (QueueException);                                 //23
        @*/                                                              //24
      public void dequeue() throws QueueException;                       //25
                                                                         //26
      /*@ public normal_behavior                                       //27
        @   modifiable size, theQueue:                                 //28
        @   ensures theQueue.equals(\old(theQueue.insertBack(x)));     //29
        @ also                                                         //30
        @ public exceptional_behavior                                  //31
        @   requires theQueue.size() == CAPACITY;                      //32
        @   signals (QueueException);                                  //33
        @*/                                                            //34
      public void enqueue(Object x) throws QueueException;             //35
      /*@ public normal_behavior                                       //36
        @   requires !theQueue.isEmpty();                              //37
        @   ensures \result == theQueue.first()                        //38
        @ also                                                         //39
        @ public exceptional_behavior                                  //40
        @   requires theQueue.isEmpty();                               //41
        @   signals (QueueException);                                  //42
        @*/                                                            //43
      public Object peek() throws QueueException;                      //44
      /*@ public normal_behavior                                       //45
        @   requires x != null;                                        //46
        @   ensures \result ==(\exists int index; 0<=index && index <  //47
            this.int_size(); theQueue.itemAt(index) == x);
        @   also                                                       //48
        @   requires x == null;                                        //49
        @   ensures \result ==(\exists int index; 0<=index && index <  //50
            this.int_size();theQueue.itemAt(index) == null);
        @  also                                                        //51
        @*/                                                            //52
      public /*@ pure@*/ boolean contains(Object x)                    //53
      throws QueueException;                                           //54
}
```

Figure 1. Source file Queue.java annotated with JML Specification (adapted from [1] changed implementation to Queue and extended)

The Need for JMLdoc

JML specifications for a Java module can be refined using the JML specifications for the same module. The specifications shown in Figure 1 could actually be divided into two parts. Figure 1 shows how the specifications of Figure 1 could be divided into two parts. The exceptional behavior specifications can be refined separately as show in Figure 2. JML specifications also support inheritance so they can also be inherited from a super class or an interface. As a result of the inheritance property of the specifications, it is not possible to obtain the complete specifications by just viewing a single file or even specification of a single module. The complete specifications can only be obtained by combining both the refined and inherited specifications.

```
//@ refine Queue <- "Queue.java";                                    //1
//@ model import org.jmlspecs.models.*;                              //2
public interface Queue {                                             //3
      /*@ also                                                       //4
        @         public_exceptional_behavior                      //5
        @     requires theQueue.isEmpty();                         //6
        @         signals (QueueException);                        //7
        @*/                                                        //8
      public void dequeue() throws QueueException;                 //9
                                                                   //10
      /*@ also                                                     //11
        @         public_exceptional_behavior                      //12
        @     requires theQueue.size() == CAPACITY;                //13
        @         signals (QueueException);                        //14
        @*/                                                        //15
      public void enqueue(Object x) throws QueueException;         //16
                                                                   //17
      /*@ also                                                     //18
        @ public_exceptional_behavior                              //19
        @   requires theQueue.isEmpty();                           //20
        @   signals (QueueException);                              //21
        @*/                                                        //22
      public Object peek() throws QueueException;                  //23
                                                                   //24
      /*@  also                                                    //25
        @ public_exceptional_behavior                              //26
        @   requires theQueue.isEmpty();                           //27
        @   signals (QueueException);                              //28
        @*/                                                        //29
      public boolean contains(Object x) throws QueueException;     //30
}                                                                  //31
```

Figure 2. Refined Specifications for file QueueRefine.java

It is not very efficient for a user of a Java module to look into the related files for inheritance and refinement, to get the complete specification of a specific module. This is highly inefficient in a huge application with several specification files.

A Java module provider might need to make specifications of those modules available to the clients. An efficient and convenient distribution medium of thesis specifications is the internet. The standard Java Development Kit (JDK) documents and publishes the built-in classes and interfaces using Javadoc, which is a tool for generating documentation for Java. Javadoc requires documentation to be written in a certain way which will be explained in later chapters.

JML specifications cannot be interpreted by existing documentation generators like Javadoc. Also Javadoc does not provide a mechanism to combine refined specifications with inherited specifications. So, a new documentation generation like Javadoc is needed for JML in order to parse the specifications. A new documentation generation should be able to work with the existing distribution mechanism like internet, so it should be able to work on web browsers. Also, the new documentation generator should also be able to perform the tasks of Javadoc documentation generator. This was the main motivation for JMLdoc and hence it was invented [2] [12]. JMLdoc was introduced with the goal of generating browsable HTML from JML specifications keeping in mind the following goals:

- Produce HTML documents from JML annotations in a source file, which can be viewed and be compatible with existing browsers.
- For the purpose of integrating inherited and refined specifications from different files into a complete specification.
- To aid in referencing and browsing by utilizing hyperlinks

- To be able to perform the tasks of Java's documentation generator, Javadoc, as well.

Overview of the Current JMLdoc Implementation

The current implementation of JMLdoc, the JML type checker creates an environment for the current file under compilation which can be thought of as a compilation unit. The tool then creates a HTML document for each class, interface and inner classes in the compilation unit. The information of class is stored in its type attributes. Using these type attributes of inherited members of a class or interface, the JML specifications that are inherited are obtained. The current specifications of a class are combined with its inherited specifications and then they are converted into HTML format. Javadoc style comments are also converted to HTML. The output of the tool contains HTML files containing documentation regarding each class or interface in the compilation unit used as input. The amount if details in the output is user selectable; by default, public and protected class members are documented.

The current JMLdoc tool is an extension of the OpenJDK source code. The JMLdoc tool has to combine functionality from two sources: the OpenJML tool to parse the specifications associated with Java program elements and the Javadoc tool to output the desired HTML pages. These two tools enter program elements into symbol tables differently (OpenJML uses the techniques supplied by the OpenJDK Java compiler). It would require a maintenance-intensive merging of the two implementations into a third one to use these tools directly. Instead, the current JMLdoc uses the OpenJDK facility of independent, co-existing compilation contexts. The Java code is parsed and represented using Javadoc, then parsed and separately represented using OpenJML, and then, information from the OpenJML specifications is added into the Javadoc

HTML pages as those HTML pages are generated. This approach has a few drawbacks: the code is parsed twice, symbol table entries from one compilation context have to be translated into the other context and the Javadoc structure does not actually contain JML elements such as model fields, methods, and classes.

Classes used & Extended to Implement the Current JMLdoc

The Main class that contains the entry points for the JMLdoc tool could, derive from either the Main class of OpenJML or the Main class of Javadoc. Since OpenJML requires more tool initialization, it was more convenient to derive JMLdoc's Main from OpenJML's. Command-line options that JMLdoc adds to Javadoc are processed in Main and help information is provided by the JmlStart class.

In order to generate JML information in the HTML output, the current JMLdoc implementation extends the Javadoc source code as follows. Javadoc operates by executing a set of Writers for various program elements. These have interfaces (e.g. *ClassWriter*) and specific implementations (e.g. *ClassWriterImpl*). The JMLdoc tool implemented its own set of writers (e.g. *ClassWriterJml*) that extend the corresponding Javadoc writers. Thus the classes *ClassWriterJml*, *FieldWriterJml*, *MethodWriterJml*, *ConstructorWriterJml*, and *NestedClassWriterJml* were created.

Javadoc instantiates these writers using a writer factory, *WriterFactoryImpl*, which is an implementation of the interface *WriterFactory*. In order to have the various JML writers instantiated at the correct time, JMLdoc has its own implementation of *WriterFactory*, namely, *WriterFactoryJml*. The *WriterFactoryImpl* class is not written to be extended, so

WriterFactoryJml is a direct implementation of *WriterFactory*, at the cost of duplicating code from *WriterFactoryImpl*.

Javadoc instantiates the *WriterFactory* in a *Configuration* object. Javadoc's implementation of *Configuration* (an abstract class) is *ConfigurationImpl*. JMLdoc extends *ConfigurationImpl* as *ConfigurationJml* in order to instantiate a *WriterFactoryJml* instead of a *WriterFactoryImpl*. This class also supplies a version string giving the date of the build represented by the tool.

The *Configuration* object used to generate the *WriterFactory* is instantiated as a singleton object. Consequently, in order that the tool use a *ConfigurationJml* object instead of Javadoc's *ConfigurationImpl*, the singleton is instantiated and initialized in Main, before other initialization code has a chance to create the *ConfigurationImpl* alternative. In this design, the standard Javadoc doclet is still used.

Overview of Doclets in Java

Doclets are the programs which are used as input to the Javadoc tool in order to specify the content and output produced by the Javadoc tool. These doclets are written in Java programming language as well, they are written using the doclet API. The default doclet utilized by Javadoc to create API documentation in HTML format, is, the standard doclet provided by Sun. However, custom doclets can also be supplied to customize the output of Javadoc as desired. An example of a doclet is shown in Figure 3. In lines 4 and 5, all the classes are being collected and are being iterated, lines 6-10 iterate over the methods of each class and print them out. The remaining part

of the code iterates of the fields of each class and prints them out and also the corresponding tags of each fields are printed.

```
import com.sun.javadoc.*;                                                    //1
import java.text.*;                                                          //2
public static boolean start(RootDoc root) {                                 //3
  ClassDoc[] classes = root.classes();                                       //4
  for (int i=0; i< classes.length; i++) {                                    //5
    MethodDoc[] methods = classes[i].methods();                              //6
    System.out.println("Methods");                                           //7
    System.out.println ("-------");                                          //8
    for (int j=0; j<methods.length; j++) {                                   //9
      System.out.println ("Method: name = " + methods[j].name());           //10
    }                                                                        //11
    System.out.println ("Fields");                                          //12
    System.out.println ("------");                                          //13
                                                                             //14
    FieldDoc[] fields = classes[i].fields();                                 //15
for (int j=0; j<fields.length; j++) {                                        //16
      Object[] field_info = {fields[j].name(), fields[j].commentText(),      //17
                  fields[j].type()};                                         //18
      System.out.println (FIELDINFO.format(field_info));                     //19
                                                                             //20
      Tag[] tags = fields[j].tags();                                         //21
      for (int k=0; k<tags.length; k++) {                                    //22
     System.out.println ("\tField Tag Name= " + tags[k].name());            //23
     System.out.println ("\tField Tag Value = " + tags[k].text());          //24
      }                                                                      //25
    }                                                                        //26
  }                                                                          //27
  return true;                                                               //28
}                                                                            //29
private static void out(String msg) {                                       //30
  System.out.println(msg);                                                   //31
}                                                                            //32
private static MessageFormat METHODINFO =                                    //33
  new MessageFormat("Method: return type {0}, name = {1};");                 //34
private static MessageFormat FIELDINFO =                                     //35
  new MessageFormat("Field: name = {0}, comment = {1}, type = {2};");        //36
}                                                                            //37
```

Figure 3. An example of a simple doclet. Source file SimpleDoclet.java

The code in Figure 3 shows that the Doclet API is contained in the package com.sun.javadoc. Since the code is plugging in to the Javadoc tool and not creating a standalone application, Javadoc calls the doclet from the method public static Boolean start (RootDoc root). The above doclet is run for the following code below shown in Figure 4

```
public class SimpleOrder  {                                                    //1
public SimpleOrder() { }                                                       //2
public String getSymbol() {                                                    //3
return Symbol;                                                                 //4
}                                                                              //5
public int getQuantity() {{                                                    //6
return Quantity;                                                               //7
}                                                                              //8
/**                                                                            //9
* A valid stock symbol.                                                        //10
*                                                                             //11
* @see A big book of valid symbols for more information.                      //12
*/                                                                            //13
private String Symbol;                                                         //14
/**                                                                            //15
* The total order volume.                                                      //16
*                                                                             //17
* @mytag My custom tag.                                                       //18
*/                                                                            //19
private int Quantity;                                                          //20
private String OrderType;                                                      //21
private float Price;                                                           //22
private String Duration;                                                       //23
private int AccountType;                                                       //24
private int TransactionType;                                                   //25
}                                                                              //26
```

Figure 4. Code used to demonstrate the SimpleDoclet. Source file: SimpleOrder.java

The Javadoc tool is now invoked using the SimpleDoclet on SimpleOrder. The output generated by the Javadoc tool can be seen below in Figure 5

```
Loading source file SimpleOrder.java...
Constructing Javadoc information...
Methods
-------
Method: name = getSymbol
Method: name = getQuantity
Fields
------
Field: name = Symbol, comment = A valid stock symbol., type =
java.lang.String;
      Field Tag Name= @see
      Field Tag Value = A big book of valid symbols for more information.
Field: name = Quantity, comment = The total order volume., type = int;
      Field Tag Name= @mytag
      Field Tag Value = My custom tag.
Field: name = OrderType, comment = , type = java.lang.String;
Field: name = Price, comment = , type = float;
Field: name = Duration, comment = , type = java.lang.String;
Field: name = AccountType, comment = , type = int;
Field: name = TransactionType, comment = , type = int;
```

Figure 5. Output of the Javadoc tool using SimpleDoclet.

Once the *start* method starts executing, all the information parsed by Javadoc is stored in RootDoc. The RootDoc has a method classes(), which can be used to iterate over all the parsed classes. The classes() method returns an array of type ClassDoc, describing all the parsed classes and interfaces. Methods such as fields() and methods() are present in ClassDoc, which return arrays of type FieldDoc and MethodDoc respectively. These can be used for describing all the parsed methods and fields. A tag() method is present in all the Doc classes, which returns an array of type Tag, which can be used for describing both standard and custom tags. The standard tag used in this example is @see. To format the output according to a fixed templated, the MessageFormat class is used.

The Problem with the Current Design

The problem with current design of JMLdoc is that it depends on the standard doclet of Javadoc making it very inflexible. It lacks the support for doclet provisions. Figure 6 shows the structure of the current JMLdoc implementation. The output of the Javadoc tools is controlled by a XML resource file. The standard doclet in Javadoc reads the XML file and calls various builders corresponding to the XML elements. These builders thereby call the corresponding writers. The current JMLdoc implementation extends these writers in order to add the JML specifications in the output files generated by Javadoc. The problem with this approach is that any changes to the standard doclet would need to be reflected into the JML writers in order for them to function properly. Also changing the doclet would require re-implementing all the JML writers again.

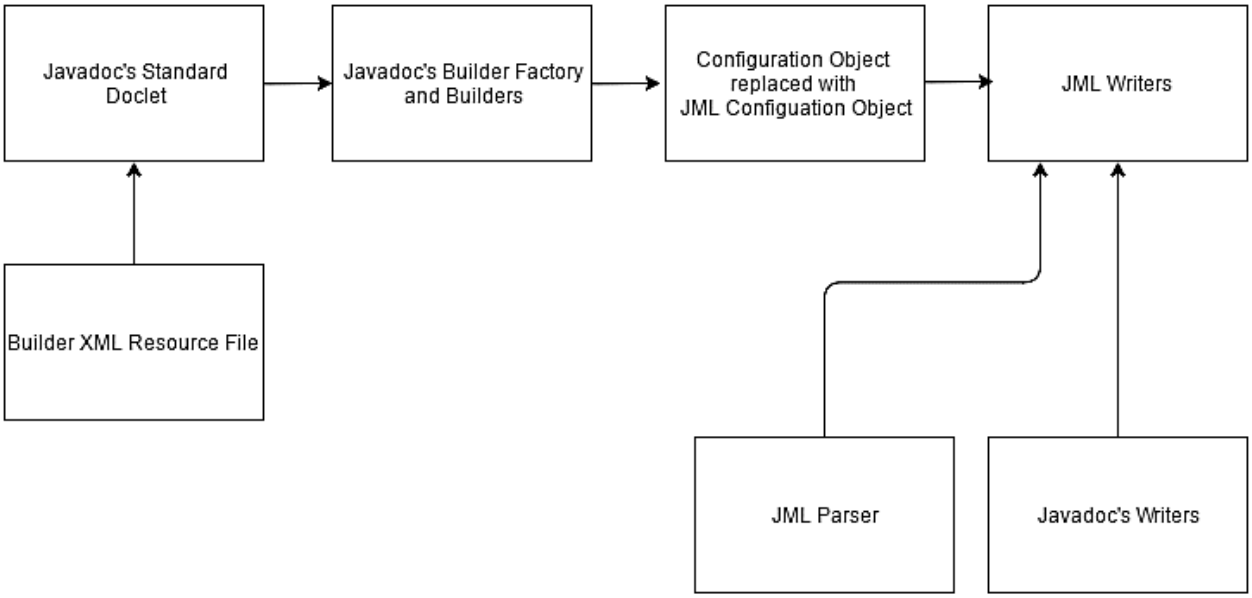


Figure 6. The current JMLdoc Implementation Structure

A JMLdoclet Based Design

Using Javadoc with custom doclets makes Javadoc really flexible and customizable. Doclets can be used to generate any kind of text file output, such as HTML, SGML, XML, RTF, and MIF. In this thesis, a doclet based design for JMLdoc in OpenJML is proposed and a JMLdoclet is presented [10].

This approach is more aligned with the design of Javadoc and its provision for doclet extensions, and meaning more minimization of maintenance as Javadoc evolves. This takes advantage of the expected and supported extension capability of Javadoc.

The organization of Javadoc's HTML output is controlled by a XML resource file. This file is parsed and for each XML element in the document a corresponding build method is called by reflection. For example, the doclet.xml file specifies that one part of the description of a `ClassDoc` is a `ClassHeader`. To generate the corresponding HTML, the `buildClassHeader`

method of *ClassBuilder* is called by reflection. So one means of adding additional information into the HTML is to provide a different XML description file. In order to do that, a new doclet (JMLdoclet) is created that references the new XML file. This design is a good solution for the current drawbacks in the current design as it makes JMLdoc architecturally strong. This design would make maintenance of JMLdoc easier as Javadoc evolves to newer versions. Also, if any changes are required in the structure of JMLdoc's HTML documents, then making appropriate changes in the JMLdoclet would be enough. Any changes to Java Modeling Language can also be easily reflected in JMLdoc by modifying the JMLdoclet correspondingly.

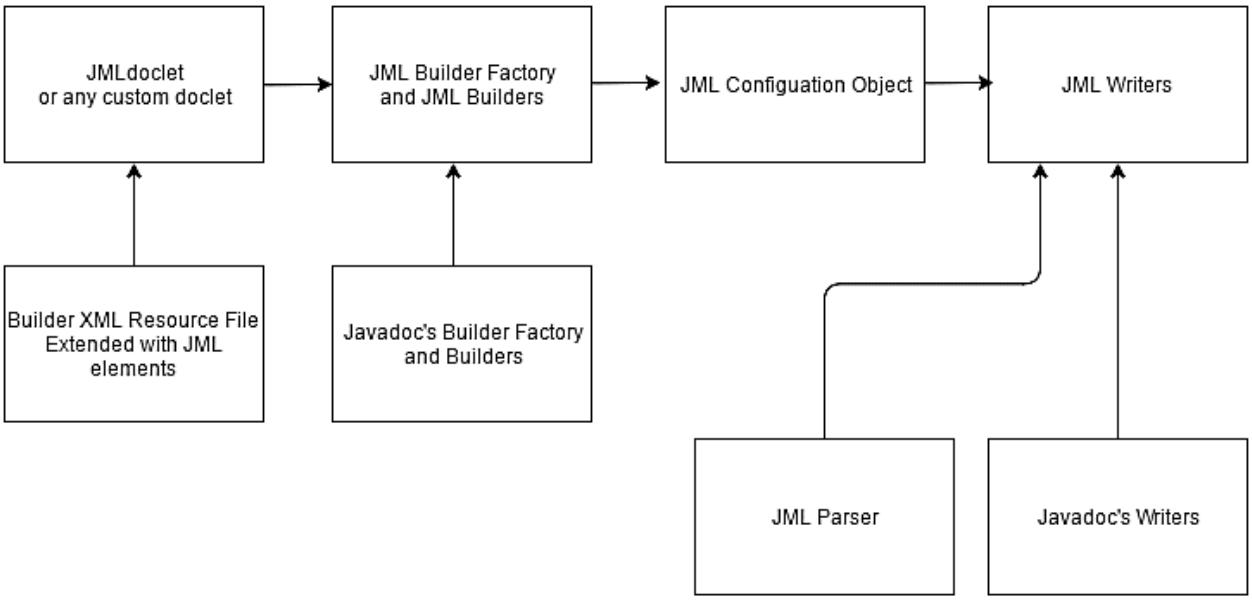


Figure 7. JMLdoclet based JMLdoc Implementation Structure

Overview of the Thesis

Chapter Two explains in details about the requirements of a documentation generator. All the different functions a documentation generator needs to perform as explained in detail in this chapter. Chapter Three discusses the implementation details of the JMLdoclet proposed in this

thesis. Related work is discussed in Chapter Four and it is compared with the current proposed JMLdoclet. Discussion about future work, alternate designs proposals and contributions are presented in Chapter Five.

CHAPTER TWO:
REQUIRMENTS FOR A DOCUMENTATION GENERATOR

Introduction

The previous chapter has introduced JML specifications and what it means to combine these specifications. A documentation generator for JML should generate browsable specifications which can be viewed using a web browser. This chapter discusses the requirements for generating browsable specifications.

Compatibility with Javadoc Style Comments

Javadoc is a documentation generator for Java. It is the accepted standard for documenting API's in java. The documentation tool for JML should also interprets Javadoc style comments, so that a provider of a java module can write JML specifications and Javadoc documentation and be able to see both at once.

Cross-Referenced Documentation

A Java class can contain members that are instances of classes. In order for a user of a Java module to browse the specifications in an easy way the documentation should be cross referenced. This would allow a user of a class to see the specifications of classes that are referenced by the current class with the click of a button. HTML allows cross-referencing through hyperlinks. The documentation generator for JML should make use of hyperlinks to create a cross-referenced document.

Desugaring the Specifications

The main requirement is to combine the specifications from refinement and inheritance i.e. desugaring the specifications [8]. The documentation generator should combine the specifications from the separate files in which the refinements and classes are written in to a single HTML document. An example of this is shown in Figures 8 and 9.

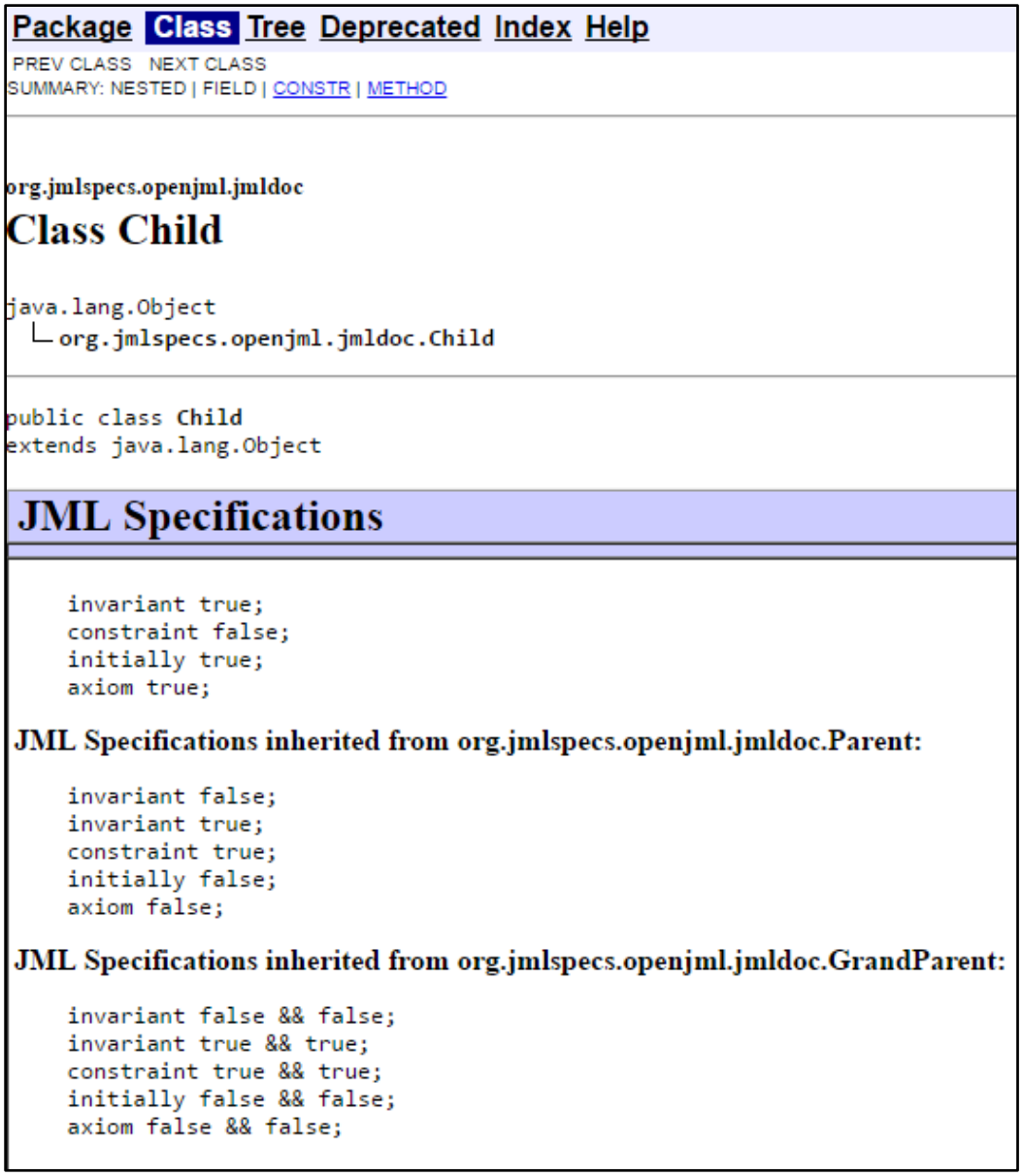
```
package org.jmlspecs.openjml.jmldoc;

public class Child extends Parent {
    //@ invariant true;
    //@ constraint false;
    //@ initially true;
    //@ axiom true;
}

class Parent extends GrandParent {
    //@ invariant false;
    //@ invariant true;
    //@ constraint true;
    //@ initially false;
    //@ axiom false;
}

class GrandParent {
    //@ invariant false && false;
    //@ invariant true && true;
    //@ constraint true && true;
    //@ initially false && false;
    //@ axiom false && false;
}
```

Figure 8. Class child which inherits from class Parent, Parent inherits from class GrandParent



Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

org.jmlspecs.openjml.jmldoc

Class Child

java.lang.Object
  └ org.jmlspecs.openjml.jmldoc.Child

public class Child
extends java.lang.Object

JML Specifications

```
invariant true;
constraint false;
initially true;
axiom true;
```

JML Specifications inherited from org.jmlspecs.openjml.jmldoc.Parent:

```
invariant false;
invariant true;
constraint true;
initially false;
axiom false;
```

JML Specifications inherited from org.jmlspecs.openjml.jmldoc.GrandParent:

```
invariant false && false;
invariant true && true;
constraint true && true;
initially false && false;
axiom false && false;
```

Figure 9. Combined Specifications for class Child in the output of JMLdoc using JMLdoclet.

User Interface (Output) Design

The user interface or output design has been modeled on that produced by Javadoc. The HTML file for a class has the class hierarchy at the very top of the page. This allows the user to see the inheritance tree.

The inheritance tree is followed by a series of tables. Each series of tables consists of a table showing the members of the current class followed by tables showing the members inherited from classes in the inheritance hierarchy. The members in the tables are hyperlinked to provide easy access to the full documentation for those members. The tables naming the inner classes and interfaces (if any) are displayed first followed by the tables for fields, constructors and finally methods.

The tables are linked to the documentation for each member of the class/interface. The documentation for the methods is the most interesting. It consists of the Javadoc style documentation followed by the complete JML specification for that method if it exists. Figures 10 and 11 show a sample HTML file generated from HTML specifications.

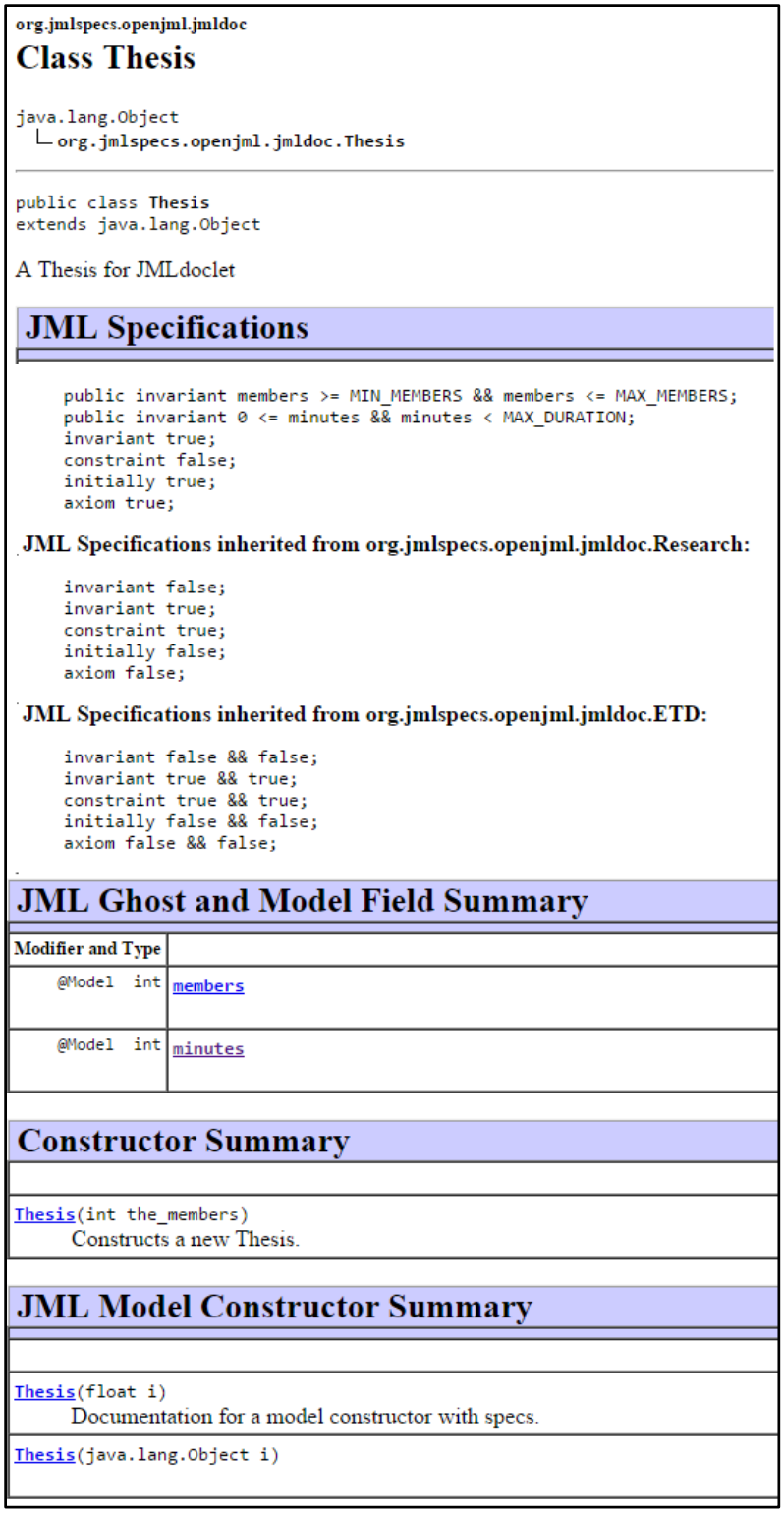
User Options

The tool is used to generate browsable documentation. This means creating a HTML file for each reference type. Sometimes more than one reference type can appear in a single file. The tool creates HTML files for each reference type. A reference type has public, private, protected and package protected members. A JML specification can also be tagged as public, private, protected or package protected. The documentation tool produces documentation for the public and protected members by default. It also generates the complete specification for public and protected parts of the JML specifications. A provider of a package might want to document the private members and private specifications. To allow the user increased flexibility in generating documentation various options are provided by the tool. An option is given to the user to allow documentation for inherited members (members that are not overridden) to be generated. By

default the detailed documentation for these members is not printed. A table below the description table in the output indicates which members are inherited from which superclass and/or super interfaces. The user can choose to suppress the documentation for the private and protected members and specifications. The following gives the options available and the result of using each option.

Table 1. Some of the options available for JMLdoc

Command-line Option	**Action**
private	This tells JMLdoc to generate browsable documentation for the private members of a class or interface.
noprot	This option prevents the browsable documentation for protected members from being generated.
nopkgprot	This option prevents the browsable documentation for package protected members from being generated.
Inherited	This causes JMLdoc to generate browsable documentation.



org.jmlspecs.openjml.jmldoc

Class Thesis

java.lang.Object
  └ org.jmlspecs.openjml.jmldoc.Thesis

---

public class **Thesis**
extends java.lang.Object

A Thesis for JMLdoclet

JML Specifications

```
public invariant members >= MIN_MEMBERS && members <= MAX_MEMBERS;
public invariant 0 <= minutes && minutes < MAX_DURATION;
invariant true;
constraint false;
initially true;
axiom true;
```

JML Specifications inherited from org.jmlspecs.openjml.jmldoc.Research:

```
invariant false;
invariant true;
constraint true;
initially false;
axiom false;
```

JML Specifications inherited from org.jmlspecs.openjml.jmldoc.ETD:

```
invariant false && false;
invariant true && true;
constraint true && true;
initially false && false;
axiom false && false;
```

JML Ghost and Model Field Summary

Modifier and Type	
@Model int	members
@Model int	minutes

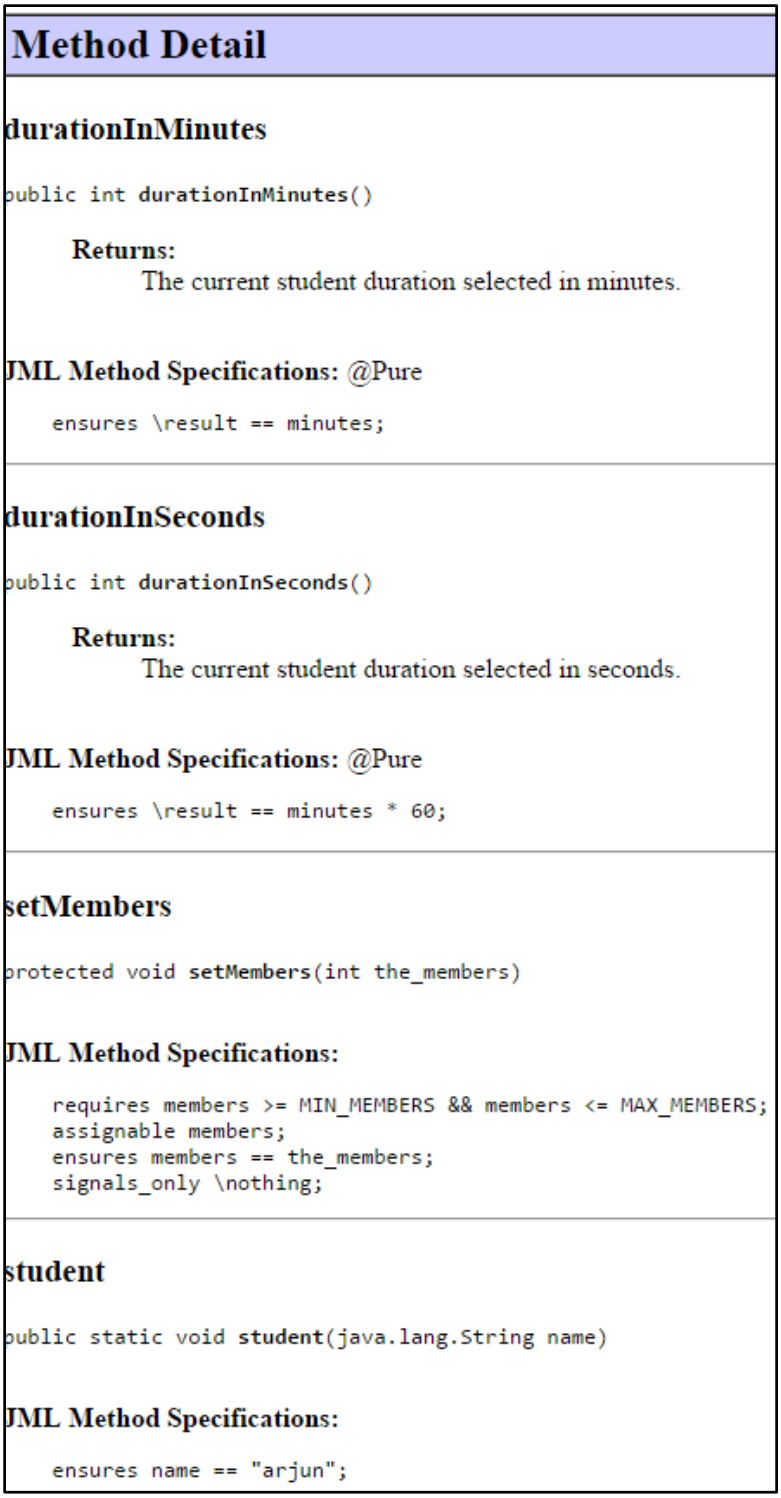
Constructor Summary

Thesis(int the_members) Constructs a new Thesis.

JML Model Constructor Summary

Thesis(float i) Documentation for a model constructor with specs.
Thesis(java.lang.Object i)

Figure 10. Screenshot of a generated HTML file using the new JMLdoclet Design



Method Detail

durationInMinutes

```
public int durationInMinutes()
```

Returns:
The current student duration selected in minutes.

JML Method Specifications: @Pure

```
ensures \result == minutes;
```

---

durationInSeconds

```
public int durationInSeconds()
```

Returns:
The current student duration selected in seconds.

JML Method Specifications: @Pure

```
ensures \result == minutes * 60;
```

---

setMembers

```
protected void setMembers(int the_members)
```

JML Method Specifications:

```
requires members >= MIN_MEMBERS && members <= MAX_MEMBERS;
assignable members;
ensures members == the_members;
signals_only \nothing;
```

---

student

```
public static void student(java.lang.String name)
```

JML Method Specifications:

```
ensures name == "arjun";
```

Figure 11. Screenshot of a JMLdoc generated HTML file showing Method Specifications

CHAPTER THREE:
IMPLEMENTATION OF THE JMLDOCLET

Introduction

This Chapter discusses the key implementation details and the process of implementation about the JMLdoclet presented in this thesis. The key change being made to the existing implementation of the JMLdoc is that the specifications for all the JML elements are going to be generated using Java's Reflection API. Utilizing Java Reflection, it is conceivable to get data about classes, interfaces, fields and methods at runtime, without having information about the names of classes, methods, etc. during compile time. It is additionally conceivable to instantiate new objects, invoke methods and get/set field values utilizing reflection. Using Java Reflection and the resource file doclet.xml used by Javadoc, an alternate implementation of JMLdoc is presented which supports doclet provision capability for JMLdoc in OpenJML

Overview of Javadoc Implementation

In order to get a better understanding of the implementation of the JMLdoclet, first a brief overview of the current implementation of Javadoc's implementation of the default doclet is presented. The JMLdoclet will be presented in a similar mechanism of the Javadoc's default doclet. The default doclet of Javadoc is a *HtmlDoclet* which is an extension of *AbstractDoclet.* The *HtmlDoclet* has a *Configuration* Object in which the *WriterFactory* is instantiated. The *WriterFactory* instantiates various other writers like *ClassWriter*, *PackageSummaryWriter*, *MethodWriter*, etc. as needed. The *HtmlDoclet* calls

generateOtherFiles, *generateClassFiles*, *and generatePackageFiles* methods in order to generate the documentation files.

When these above methods are called the process of parsing the doclet.xml file takes place. An example of the parsing process in the code is explained in Figure 8 when *generateClassFiles* is called. When the *generateClassFiles* method is called the current element is analyzed and if it is a type of Class, then a *BuilderFactory* is instantiated in the *Configuration* and using the *BuilderFactory* a *ClassBuilder* is instantiated. Finally, the *build* method of the *ClassBuilder* is called. This is where the XML parsing starts and the current *XMLNode* is analyzed and depending on the text, a corresponding method in the *ClassBuilder* is called by Reflection**.** The parent abstract class *AbstractBuilder* handles the reflection mechanism and the process for the reflection is explained and shown in Figure 9.

```
public XMLNode parseXML(String root) {                                    //1
        if (xmlElementsMap.containsKey(root)) {                           //2
            return xmlElementsMap.get(root);                              //3
        }                                                                 //4
        try {                                                             //5
            currentRoot = root;                                           //6
            isParsing = false;                                            //7
            SAXParserFactory factory = SAXParserFactory.newInstance();    //8
            SAXParser saxParser = factory.newSAXParser();                 //9
            InputStream in = configuration.getBuilderXML();              //10
            saxParser.parse(in, this);                                   //11
            return xmlElementsMap.get(root);                             //12
        } catch (Throwable t) {                                          //13
            throw new DocletAbortException();                            //14
        }                                                                //15
    }                                                                    //16
```

Figure 12. *LayoutParser'*s *parseXML* method

The *AbstractBuilder* handles the reflection mechanism based on the *XMLNode* passed to it. The *invokeMethod* shown in Figure 9 takes in three arguments- a string build appended with the current *XMLNode* value, the class objects of the *XMLNode* and *Content* and the objects

themselves. The *invokeMethod* then grabs the corresponding method using the string value passed into the first argument and then invokes that particular method. An example of this is demonstrated in Figure 10, it can also be seen in the figure, how the corresponding child *XMLNode*s are processed for each parent *XMLNode* using the *buildChildren* method of *AbstractBuilder*. The *buildChildren* just iterates through all the children of *XMLNode* and calls *build* for each of them

```
protected void build(XMLNode node, Content contentTree) {                  //1
        String component = node.name;                                      //2
        try {                                                              //3
            invokeMethod("build" + component,                              //4
                    new Class<?>[]{XMLNode.class, Content.class},          //5
                    new Object[]{node, contentTree});                      //6
        } catch (NoSuchMethodException e) {                                //7
            e.printStackTrace();                                           //8
            configuration.root.printError("Unknown element:"+component);  //9
            throw new DocletAbortException();                              //10
        } catch (InvocationTargetException e) {                            //11
            e.getCause().printStackTrace();                                //12
        } catch (Exception e) {                                            //13
            e.printStackTrace();                                           //14
            configuration.root.printError("Exception " +                   //15
                    e.getClass().getName() +                               //16
                    " thrown while processing element: " + component);     //17
            throw new DocletAbortException();                              //18
        }                                                                  //19
    }                                                                      //20
```

Figure 13. *AbstractBuilder*'s *build* and *buildChildren* methods to handle reflection.

The parsing process begins instantiating a *LayoutParser* using the *Configuration* object and then *LayoutParser's parseXML* method is called, which will be passed a String corresponding to the specific root node in the XML file, for example *Classdoc* is root node for all the class files in the XML file. The *LayoutParser* maintains a *HashMap* called *xmlElementsMap*. Whenever the parseXML method is called, the *xmlElementsMap* is

populated with the *XMLNodes.* The *parseXML* method uses *SAXParser* library in order to parse the XML, once a node is parsed it will be placed into *xmlElementsMap.* Once the *parseXML* methods returns the build method of *ClassBuilder* checks the return value and calls the corresponding method by reflection which is demonstrated clearly in Figure 10. An example for the case when the XMLNode has the text *<Classdoc>* is shown in Figure 10 *LayoutParser* extends *DefaultHandler* and overrides the *startElement* and *endElement* methods, in which the process on insertion into *xmlElementsMap* takes place.

```
public void build() throws IOException {
        build(LayoutParser.getInstance(configuration).parseXML(ROOT),
contentTree);
}

public void buildClassDoc(XMLNode node, Content contentTree) throws Exception
{

         String key;
         if (isInterface) {
             key =  "doclet.Interface";
         } else if (isEnum) {
             key = "doclet.Enum";
         } else {
             key =  "doclet.Class";
         }

         contentTree = writer.getHeader(configuration.getText(key) + " " +
                 classDoc.name());
         Content classContentTree = writer.getClassContentHeader();
         buildChildren(node, classContentTree);
         contentTree.addContent(classContentTree);
         writer.addFooter(contentTree);
         writer.printDocument(contentTree);
         writer.close();
         copyDocFiles();
  }
```

Figure 14. Example of *ClassBuilder* calling the corresponding *buildClassDoc* method by reflection for the XML element *<ClassDoc>*

Approach for the New Design

The idea for JMLdoclet is to extend the doclet.xml file by adding XML elements corresponding to the JML elements and to create a JMLdoclet which extends the *BuilderFactory* and various builders like *ClassBuilder*, *FieldBuilder*, etc. by adding methods which can be called reflection when a XML element corresponding to a JML element is encountered. This is the main idea behind the implementation.

Extending the Doclet.xml

The doclet.xml is the resource file which controls the HTML output of Javadoc. Each XML element in this resource file will be parsed and corresponding builders and writers will be called by reflection. This is how Javadoc's doclet provision capability was implemented. A similar approach will be used to implement JMLdoc's doclet provision capability. The resource file is extended with some JML nodes in order to create the JMLdoclet. The newly added fields contain the text JML. This resource file will be passed as the *DEFAULT_BUILDER_XML* in the *Configuration* object. In order to swap to a different doclet, replacing this resource with a different one would suffice.

Refer to Appendix A for the extended doclet.xml structure. The XML resource file has been converted into JSON format for clear understanding of the structure. This new doclet.xml file will be set in *ConfigurationJml* class as *DEFAULT_BUILDER_XML*.

Creating a JMLdoclet

A *JMLdoclet* class is created which extends *AbstractDoclet*. The *JMLdoclet* class has methods from the standard *HtmlDoclet* as well as additional methods corresponding to JML specifications. The new methods are *generateClassSpecifications* which internally uses other methods in order to instantiate the newly extended builders and writers. This new JMLdoclet references the newly created doclet.xml resource file. The same code of *generateClassFiles* has been except that *generateClassSpecifications* uses a *classSpecsBuilder* instead of a *classBuilder*.

Extending the Builders

The builders extended in the thesis are shown below in Table 2. All the previous builders and the *BuilderFactory* have been extended. The *builderXMLPath* in the *ConfigurationJml* has been set to the new doclet resource file jmldoclet.xml. A new method *getBuilderFactory* has been added to *ConfigurationJml* in order to process the new extended *BuilderFactoryJml*. The source code for all the extended builders can be referenced from Appendix B. The corresponding writers associated with the builders are also modified.

Table 2. Original builder classes and corresponding extended JML builder classes

BuilderFactory	*BuilderFactoryJml*
ClassBuilder	*ClassBuilderJml*
ConstructorBuilder	*ConstructorBuilderJml*
FieldBuilder	*FieldBuilderJml*
MethodBuilder	*MethodBuilderJml*
MemeberSummaryBuilder	*MemberSummaryBuilderJml*

CHAPTER FOUR: RELATED WORK

This chapter discusses the related work in the area of documentation generators with and without support for doclet extension capabilities. The current proposed design is compared with other documentation generators and also possible alternate designs are suggested. The documentation generators which are capable of parsing annotations (which are either in the form of pusedo-code or decryptions written as comments) and produce a readable output either in HTML or other formats are only compared.

iDoclet from iContract Project

iContract [6] is a preprocessor for Java. iContract first processes the Java file and produces a set of decorated Java files. Once these files are generated, they can be compiled with the standard Java Compiler.

iContract directives in Java code, just like Javadoc directives, reside in the class and method comments. This enables iContract to have backwards-compatibility with existing Java code. So, without iContract assertions, Java code can always be directly compiled with the Java compiler.

Typically, a program would start in a development phase and transition into a testing phase and then transition into a production phase. This process can be made efficient by instrumenting the code with iContract assertions in the development phase. This enables detection of newly introduced bugs, early on, in the development process itself. In the test environment the bulk of the assertions can still be kept enabled, but usually are taken out for performance-critical classes. In iContract it possible to select explicitly, the classes that require instrumentation with assertions.

The documentation generator used in iContract is Javadoc with the use of a special doclet named iDoclet. iDoclet is an extension of Sun's standard doclet from JDK 1.2 that includes assertion information in the generated API docs. It supports three new tags: *@pre*, *@post* and *@inv*. The iDoclet extends implementations for various writers like *ClassWriter*, *PackageWriter*, *TreeWriter*, etc. However, this doclet does not recognize JML specifications and cannot combine JML specifications from refinements and inheritance into a complete specification. This doclet only works with JDK version 1.2. It fails to work with later versions of JDK as the packages this doclet uses have been removed from JDK 1.4 and later. This project is seems to be not well maintained.

DocGen

DocGen is a Java doclet developed on the basis of the standard doclet. As long as a specific format is trailed by the documentation, DocGen can extract and show any contract, even the inherited contracts which are documented in a formal comment unit. DocGen is built upon the standard doclet of Java, but it cannot parse the JML specifications. So, it fails to combine JML specifications from refinements and inheritance into a complete specification.

XJavadoc from JTest/JContract Tools

JTest [7], is an automated unit testing tool for Java, which utilizes the Design by Contract format specification information present in the classes and automates unit level functionality. JContract is a tool which can be used as an add-on for JTest or it can be used as a standalone tool.

It checks DBC contracts during runtime and identifies any misuse of a class/component. It can also verify system-level functionality.

Using JContract, functionality testing at the system level using DBC can be done by Java programmers. In the wake of utilizing JTest to thoroughly test a class or component at the unit level, JContract will instrument and compile the DBC-commented code. In an instrumented class/component, JContract automatically tries to identify any contracts which are violated during runtime. For the task of identifying any misuse of a class or component, JContract would be particularly useful. When used in conjunction with JTest, JContract helps automate functionality testing of Java applications, and improves the overall testing of components, such as EJBs.

The documentation generator used in the JContract project is Javadoc with a specialized doclet called XDoclet. The standard Javadoc in Java has been renamed to XJavadoc (Extended Javadoc) in this project. XJavadoc replaces the standard doclet provide by Sun, for use with Javadoc, by a new doclet called XDoclet. XJavadoc adds support for Design by Contract tags and generates API documentation in either HTML or XML format. However, XJavadoc is not compatible with JML. It cannot provide documentation for JML elements cannot recognize JML specifications from Java programs.

Literate Programming

Literate programming [3] is an effort to produce better and easily understandable documentation for programs. Just like annotations, the documentations in a literate programming style would require the documentation be written along with the source code. This documentation can be similarly parsed by literate programming tools into desired output formats like RTF, etc.

Literate programming usually requires a typesetting language like TeX for formatting source code. Literate programming also supports cross referencing while browsing the documentations. Javadoc and JMLdoc also support cross referencing as well. However, Literate Programming tools like DOC++, CWEB do not support JML specifications in the source code. Hence, they cannot combine the JML specifications from inheritance and refinement.

Tools Similar to Javadoc

A popular tool used for producing documentation from annotations in C++ programs is Doxygen [4]. Doxygen additionally supports other prevalent programming languages, for example, C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL, Tcl, and D. It can be utilized to either create on-line documentation in HTML design and/or offline documentation in TeX format. The tool also supports many other formats as well. Some of them are-RTF, hyperlinked PDF, PostScript, UNIX man pages and compressed HTML. Since the documentation is extracted directly from the sources, it makes it easier to maintain consistency of the documentation with source code.

Another tool used for automated technical documentation production is Document! X [5]. It supports C#, VB.NET, C++, CLI and other .NET language assemblies, databases, Java projects, COM components, XSD schemas, type libraries, ASP.NET, JavaScript and Ajax. The tool also consists of a Visual comment editor add-ins for Visual Studio along with a authoring and documentation build environment.

Both Doxygen and Document! X, do not recognize JML specifications and cannot combine or generate JML specifications like JMLdoc does.

Other DBC Implementations for Java

Other DBC implementations projects for Java include:

- Contracts for Java (Cofoja) by Google
- Modern Jass
- Spring Contracts (Based on Java's Spring Framework)
- C4J

However, all these projects do not support JML elements or JML specifications and hence cannot produce documentation for JML specifications.

CHAPTER FIVE: DISCUSSIONS

Alternate Design

Aspect Oriented Programming (AOP) is a useful technique that enables adding executable blocks to the source code without explicitly changing it. AOP can also be used in Design by Contract style programming. Property-based crosscutting can be used to characterize more complex contract enforcement utilizing AOP. Since AOP provides modularity for tasks like tracing and logging, we could use AOP for logging the information of the JML and Java files. One could log and trace the information of JML and Java methods by using join points, pointcuts and advices. Join point is a point amid the execution of a program, such as the execution of a method or the handling of an exception. Advice is a move made by an aspect at a specific join point. Diverse sorts of advice include "around," "before" and "after" advice. Pointcut is a predicate that matches join points. Advice is connected with a pointcut expression and keeps running at any join point coordinated by the pointcut (for instance, the execution of a method with a specific name).

The documentation generator used in AspectJ is ajdoc. Ajdoc is also similar to Javadoc, it can render HTML documentation for pointcuts, advices and inter-type declarations, as well as Java constructs that Javadoc renders. Tools like ajdoc can be used to produce the HTML documentation in AspectJ. However, doing things dynamically doesn’t help with static documentation of the structure of the program. AOP could be used to edit the way the standard Java doclet works to modify it for JML, but the problem with this is that Javadoc relies on the code of the standard doclet, so, it is hard to maintain.

Future Work

JMLdoclet is by far the best suitable way to modify JMLdoc in order to make its design architecturally stronger, making it much easier to maintain as Javadoc or JML evolves. This section discusses some ideas to enhance the JMLdoc tool further. The tool currently generates documentation only in HTML format as it is works on the basis of Javadoc. New JMLdoclets can be developed in order to modify the tool to generate output in a various formats like TeX, XML. Also development of a single JMLdoclet with support for multiple output formats would be nice.

Class diagrams can be created from Javadoc comments in the source files. This would provide a better understanding of the structure of the inheritance and refinements of the specifications. Also, implementing dynamic views for the specifications would be useful, for example the ability to view normal behavioral specification and the ability to view exceptional behavioral specifications individually could be useful for better understanding.

CHAPTER SIX: CONCLUSIONS

Contributions

The primary contribution of this thesis is the provision for pluggable JMLdoclets for JMLdoc in OpenJML making it more aligned to the design of Javadoc and its provision for doclets. This kind of design makes JMLdoc very flexible and adaptable to different documentation requirements and it makes JMLdoc architecturally strong by eliminating the need to rely on Javadoc as well OpenJML. It also makes JMLdoc easier to maintain. This implementation still supports all the functions available in the previous implementation.

Conclusion

Using the JMLdoclet approach makes JMLdoc architecturally strong making it easier to adapt to various doclets designs and also more aligned with Javadoc's provision for doclet extension. Keeping JMLdoc aligned Javadoc ensures consistency as Javadoc evolves. JMLdoclet is by far the best suitable way to modify JMLdoc.

APPENDIX A: IMAGES

```
{
    "Doclet": {
        "PackageDoc": {
            "PackageHeader": "",
            "Summary": {
                "SummaryHeader": "",
                "InterfaceSummary": "",
                "ClassSummary": "",
                "EnumSummary": "",
                "ExceptionSummary": "",
                "ErrorSummary": "",
                "AnnotationTypeSummary": "",
                "SummaryFooter": ""
            },
            "PackageDescription": "",
            "PackageTags": "",
            "PackageFooter": ""
        },
        "AnnotationTypeDoc": {
            "AnnotationTypeHeader": "",
            "DeprecationInfo": "",
            "AnnotationTypeSignature": "",
            "AnnotationTypeDescription": "",
            "AnnotationTypeTagInfo": "",
            "MemberSummary": {
                "AnnotationTypeRequiredMemberSummary": "",
                "AnnotationTypeOptionalMemberSummary": ""
            },
            "AnnotationTypeRequiredMemberDetails": {
                "Header": "",
                "AnnotationTypeRequiredMember": {
                    "MemberHeader": "",
                    "Signature": "",
                    "DeprecationInfo": "",
                    "MemberComments": "",
                    "TagInfo": "",
                    "MemberFooter": ""
                }
```


Figure 15. Extended doclet.xml Part a.

Figure 16. Extended doclet.xml Part b.

```
            "FieldsSummaryJml": "",
            "FieldsInheritedSummary": "",
            "FieldsInheritedSummaryJml": "",
            "ConstructorsSummary": "",
            "ConstructorSummaryJml": "",
            "MethodsSummary": "",
            "MethodsSummaryJml": "",
            "MethodsInheritedSummary": "",
            "MethodsInheritedSummaryJml": ""
        },
        "EnumConstantsDetails": {
            "Header": "",
            "EnumConstant": {
                "EnumConstantHeader": "",
                "Signature": "",
                "DeprecationInfo": "",
                "EnumConstantComments": "",
                "TagInfo": "",
                "EnumConstantFooter": ""
            },
            "Footer": ""
        },
        "FieldDetails": {
            "Header": "",
            "FieldDoc": {
                "FieldHeader": "",
                "Signature": "",
                "DeprecationInfo": "",
                "FieldComments": "",
                "FieldCommentsJml": "",
                "TagInfo": "",
                "FieldFooter": ""
            },
            "Footer": "",
            "GhostFieldJml": "",
            "ModelFieldJml": "",
            "RepresentsDetailJml": ""
        },
```


Figure 17. Extended doclet.xml Part c.

```
        },
        "ConstructorDetails": {
            "Header": "",
            "ConstructorDoc": {
                "ConstructorHeader": "",
                "Signature": "",
                "DeprecationInfo": "",
                "ConstructorComments": "",
                "TagInfo": "",
                "ConstructorDocJml": "",
                "ConstructorFooter": ""
            },
            "Footer": "",
            "ModelConstructorJml": ""
        },
        "MethodDetails": {
            "Header": "",
            "MethodDoc": {
                "MethodHeader": "",
                "Signature": "",
                "DeprecationInfo": "",
                "MethodComments": "",
                "TagInfo": "",
                "MethodDocJml": "",
                "MethodFooter": ""
            },
            "Footer": "",
            "ModelMethodsJml": ""
        },
        "ClassFooter": ""
    },
    "ConstantSummary": {
        "Header": "",
        "Contents": "",
        "ConstantSummaries": {
            "PackageConstantSummary": {
                "PackageHeader": "",
                "ClassConstantSummary": {
                    "ClassHeader": "",
```


Figure 18. Extended doclet.xml Part d.

Figure 19. Extended doclet.xml Part e.

APPENDIX B: SOURCE CODE

```java
package org.jmlspecs.openjml.jmldoc.builders;

import com.sun.javadoc.ClassDoc;

public class BuilderFactoryJml extends BuilderFactory {

    private Configuration configuration;
    private WriterFactory writerFactory;
    private ClassDoc classDoc;

    public BuilderFactoryJml(Configuration configuration) {
        super(configuration);
        this.configuration = configuration;
        this.writerFactory = configuration.getWriterFactory();
    }

    @Override
    public AbstractBuilder getClassBuilder(ClassDoc classDoc,
            ClassDoc prevClass, ClassDoc nextClass, ClassTree classTree) throws Exception {

        return new ClassBuilderJml(configuration, classDoc,
                writerFactory.getClassWriter(classDoc, prevClass, nextClass,
                    classTree));
    }

   @Override
    public AbstractBuilder getMethodBuilder(ClassWriter classWriter) throws Exception {
        return new MethodBuilderJml(configuration,
            classWriter.getClassDoc(),
            writerFactory.getMethodWriter(classWriter));
    }

   @Override
    public AbstractBuilder getFieldBuilder(ClassWriter classWriter) throws Exception {
        return new FieldBuilderJml(configuration, classWriter.getClassDoc(),
                writerFactory.getFieldWriter(classWriter));
    }

    @Override
    public AbstractBuilder getConstructorBuilder(ClassWriter classWriter) throws Exception {
        return new ConstructorBuilderJml(configuration,
            classWriter.getClassDoc(), writerFactory.getConstructorWriter(
            classWriter));
    }

    @Override
    public AbstractBuilder getMemberSummaryBuilder(ClassWriter classWriter) throws Exception {
        return new MemberSummaryBuilderJml(classWriter, configuration);
    }

    @Override
    public AbstractBuilder getMemberSummaryBuilder(AnnotationTypeWriter annotationTypeWriter) throws Exception {
        return new MemberSummaryBuilderJml(annotationTypeWriter, configuration);
    }

}
```

Figure 20. Source Code BuilderFactoryJml.Java

```java
package org.jmlspecs.openjml.jmldoc.builders;

import java.util.HashSet;

public class ClassBuilderJml extends ClassBuilder {

    ClassWriterJml w;

    public ClassBuilderJml(Configuration configuration, ClassDoc classDoc,
            ClassWriter writer) {
        super(configuration);
        super.classDoc = classDoc;
        super.writer = writer;
        if (classDoc.isInterface()) {
            super.isInterface = true;
        } else if (classDoc.isEnum()) {
            super.isEnum = true;
            Util.setEnumDocumentation(configuration, classDoc);
        }
        if (containingPackagesSeen == null) {
            containingPackagesSeen = new HashSet<String>();
        }
        w = (ClassWriterJml) writer;
    }

    public void buildClassDocJml(XMLNode node) throws Exception {
        w.writeClassSpecs();
    }

    public void buildClassHeaderJml(XMLNode node) {
        w.printHtmlJmlHeader();
    }

}
```

Figure 21. Source Code ClassBuilderJml.Java

```
public ConfigurationJml() {
     builderXMLPath = "src/org/jmlspecs/openjml/jmldoc/resources/doclet.xml";
 }

 public InputStream getBuilderXML() throws FileNotFoundException {
     return builderXMLPath == null ?
         ConfigurationJml.class.getResourceAsStream("src/org/"
                 + "jmlspecs/openjml/jmldoc/resources/doclet.xml") :
         new FileInputStream(new File(builderXMLPath));
 }

 @NonNull @Override
 public BuilderFactory getBuilderFactory() {
     return new BuilderFactoryJml(this);
 }
```

Figure 22. Changes in Source Code of ConfigurationJml.java

```
package org.jmlspecs.openjml.jmldoc.builders;

import java.util.ArrayList;

public class FieldBuilderJml extends FieldBuilder {
    private FieldWriterJml w;
    protected FieldBuilderJml(Configuration configuration, ClassDoc classDoc,
            FieldWriter writer) {

        super(configuration);
        super.classDoc = classDoc;
        super.writer = writer;
        super.visibleMemberMap = new VisibleMemberMap(classDoc,
                VisibleMemberMap.FIELDS, configuration.nodeprecated);
        super.fields = new ArrayList<ProgramElementDoc>(
                super.visibleMemberMap.getLeafClassMembers(configuration));
        if (configuration.getMemberComparator() != null) {
            Collections.sort(super.fields, configuration.getMemberComparator());
        }
        w = (FieldWriterJml) writer;
    }

    public void buildFieldCommentsJml(XMLNode node) {
        w.writeJmlSpecs((FieldDoc) fields.get(currentFieldIndex));
    }

    public void buildGhostFieldJml(XMLNode node) {
        w.writeJmlGhostModelFieldDetail(classDoc, JmlToken.GHOST, "Ghost");
    }

    public void buildModelFieldJml(XMLNode node) {
        w.writeJmlGhostModelFieldDetail(classDoc, JmlToken.MODEL, "Model");
    }

    public void buildRepresentsDetailJml(XMLNode node) {
        w.writeJmlRepresentsDetail(classDoc);
    }
}
```

Figure 23. Source Code FieldBuilderJml.Java

```java
package org.jmlspecs.openjml.jmldoc.builders;

import java.util.ArrayList;

public class MethodBuilderJml extends MethodBuilder {

    private MethodWriterJml w;
    protected MethodBuilderJml(Configuration configuration, ClassDoc classDoc,
            MethodWriter writer) {

        super(configuration);
        super.classDoc = classDoc;
        super.writer = writer;
        super.visibleMemberMap = new VisibleMemberMap(classDoc,
                VisibleMemberMap.METHODS, configuration.nodeprecated);
        super.methods = new ArrayList<ProgramElementDoc>(
                super.visibleMemberMap.getLeafClassMembers(configuration));
        if (configuration.getMemberComparator() != null) {
            Collections.sort(super.methods,
                    configuration.getMemberComparator());
        }
        w = (MethodWriterJml) writer;

    }

    public void buildMethodDocJml(XMLNode node) {
        w.writeJmlSpecs((MethodDoc) methods.get(currentMethodIndex));
    }

    public void buildModelMethodsJml(XMLNode node) {
        w.writeJmlModelMethods(classDoc);
    }

}
```

Figure 24. Source Code MethodBuilderJml.java

```
package org.jmlspecs.openjml.jmldoc.builders;

import java.util.Iterator;

public class MemberSummaryBuilderJml extends MemberSummaryBuilder {

    Configuration configuration;
    ClassWriter   classWriter;
    public MemberSummaryBuilderJml(ClassWriter classWriter,
            Configuration configuration) throws Exception {
        super(configuration);
        super.classDoc = classWriter.getClassDoc();
        super.init(classWriter);
        this.configuration = configuration;
        this.classWriter = classWriter;
    }

    public MemberSummaryBuilderJml(AnnotationTypeWriter annotationTypeWriter,
            Configuration configuration) throws Exception {
        super(configuration);
        super.classDoc = annotationTypeWriter.getAnnotationTypeDoc();
        super.init(annotationTypeWriter);
    }

    public void buildFieldsSummaryJml(XMLNode node) throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.FIELDS];
        if (w == null) w = (MemberSummaryWriter) configuration
                .getWriterFactory().getFieldWriter(classWriter);
        ((FieldWriterJml) w).checkJmlSummary(classDoc);
    }

    public void buildFieldsInheritedSummaryJml(XMLNode node) throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.FIELDS];
        if (w == null) w = (MemberSummaryWriter) configuration
                .getWriterFactory().getFieldWriter(classWriter);
        VisibleMemberMap visibleMemberMap = visibleMemberMaps[VisibleMemberMap.FIELDS];
        for (Iterator<?> iter = visibleMemberMap.getVisibleClassesList()
                .iterator(); iter.hasNext();) {
            ClassDoc inhclass = (ClassDoc) (iter.next());
            if (!(inhclass.isPublic()
                    || Util.isLinkable(inhclass, configuration))) {
                continue;
            }
            if (inhclass == classDoc) {
                continue;
            }
            List<?> inhmembers = visibleMemberMap.getMembersFor(inhclass);
```

Figure 25. Source Code MemberSummaryBuilderJml.java Part a.

```
            ((FieldWriterJml) w).checkJmlInheritedSummary(classDoc, inhmembers);
        }
    }

    public void buildNestedClassesSummaryJml(XMLNode node) throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.INNERCLASSES];
        if (w == null)
            w = configuration.getWriterFactory().getMemberSummaryWriter(
                    classWriter, VisibleMemberMap.INNERCLASSES);
        ((NestedClassWriterJml) w).writeJmlNestedClassSummary(classDoc);
    }

    public void buildNestedClassesInheritedSummaryJml(XMLNode node)
            throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.INNERCLASSES];
        if (w == null)
            w = configuration.getWriterFactory().getMemberSummaryWriter(
                    classWriter, VisibleMemberMap.INNERCLASSES);
        ((NestedClassWriterJml) w)
                .writeJmlInheritedNestedClassSummaryFooter(classDoc);
    }

    public void buildMethodsSummaryJml(XMLNode node) throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.METHODS];
        if (w == null) w = (MemberSummaryWriter) configuration
                .getWriterFactory().getMethodWriter(classWriter);
        ((MethodWriterJml) w).writeJmlMethodSummary(classDoc);
    }

    public void buildMethodsInheritedSummaryJml(XMLNode node) throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.METHODS];
        if (w == null) w = (MemberSummaryWriter) configuration
                .getWriterFactory().getMethodWriter(classWriter);
        ((MethodWriterJml) w).writeJmlInheritedMemberSummaryFooter(classDoc);
    }

    public void buildConstructorSummaryJml(XMLNode node) throws Exception {
        MemberSummaryWriter w = memberSummaryWriters[VisibleMemberMap.CONSTRUCTORS];
        if (w == null) w = (MemberSummaryWriter) configuration
                .getWriterFactory().getConstructorWriter(classWriter);
        ((ConstructorWriterJml) w).writeJmlConstructorSummary(classDoc);
    }
```

Figure 26. Source Code MemberSummaryBuilderJml.java Part b.

```java
package org.jmlspecs.openjml.jmldoc.builders;

import java.util.ArrayList;

public class ConstructorBuilderJml extends ConstructorBuilder {

    private ConstructorWriterJml w;
    protected ConstructorBuilderJml(Configuration configuration,
            ClassDoc classDoc, ConstructorWriter writer) {

        super(configuration);
        super.classDoc = classDoc;
        super.writer = writer;
        super.visibleMemberMap = new VisibleMemberMap(classDoc,
                VisibleMemberMap.CONSTRUCTORS, configuration.nodeprecated);
        super.constructors = new ArrayList<ProgramElementDoc>(
                super.visibleMemberMap.getMembersFor(classDoc));
        for (int i = 0; i < super.constructors.size(); i++) {
            if (super.constructors.get(i).isProtected()
                    || super.constructors.get(i).isPrivate()) {
                writer.setFoundNonPubConstructor(true);
            }
        }
        if (configuration.getMemberComparator() != null) {
            Collections.sort(super.constructors,
                    configuration.getMemberComparator());
        }
        w = (ConstructorWriterJml) writer;
    }

    public void buildConstructorDocJml(XMLNode node) {
        w.writeJmlSpecs((ConstructorDoc) constructors.get(currentMethodIndex));
    }

    public void buildModelConstructorJml(XMLNode node) {
        w.writeJmlModelConstructors(classDoc);
    }

}
```

Figure 27. Source Code ConstructorBuilderJml.java

LIST OF REFERENCES

[1] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, February 2000. See www.cs.iastate.edu/~leavens/J ML.html.

[2] Arun David Raghavan. Design of a JML Documentation Generator. Master's Thesis, Iowa State University, 2000.2

[3] Donald E. Knuth. Literate Programming, Volume 27 of CSLI Lecture Notes. Center for the Study of Language and Information, Stanford University, 1992.

[4] Doxygen documentation generator. http://www.stack.nl/~dimitri/doxygen/.

[5] Document! X documentation generator. http://www.innovasys.com/product/dx/overview.

[6] iContract DBC implementation for Java. https://sourceforge.net/projects/icplus/.

[7] JTest/JContract DBC Implementation for Java. https://www.parasoft.com/press/major-development-in-java-testing-achieved-with-the-introduction-of-parasoft-jcontract-and-jtest/.

[8] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03, Department of Computer Science, Iowa State University, March 2000.

[9] David R. Cok. The OpenJML User Guide. http://openjml.org/.

[10] David R. Cok. The JMLdoc tool – Javadoc with JML specifications. http://openjml.org/.

[11] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes, 31(3):1-38, May, 2006

[12] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, 7(3):212-232, June 2005.