Java Modelling Language

Balint Armand Alexandru,
Barna Tudor Cristian,
Screciu Alin Constantin
Department of Computer Science,
West University
Timisoara, Romania

October 2021

Abstract

JML, short for "Java Modelling Language", is a behavioral interface specification language designed to specify Java classes and methods. JML is heavily based on syntactic sugar to make various notions and logic of the code easier to read, write and respectively, understand. In this paper we will go over JML, its uses, its integration, why it is superior to other documentation methods and lastly, giving a brief tutorial over on how to start implementing this behavioural interface specification language into your Java codes.

Keywords: Java, Java Modelling Language, Behavioral interface

Contents

1 Introduction

When it comes to writing code, it is generally acknowledged that a good practice is to structure and document your code in a way that it describes the functionality of the code well enough, and then, to make sure that the implementation complies with the requirements. Upon finding errors or erroneous output, one's first instinct is to revise the code for possible bugs or for bits of logic that do not match with what the requirements demand. Generally, this would be good enough to fix some mismanagement of the code, however, what if the requirement leads the programmer to the wrong place? Thus, programmers need tools designed to establish whether errors come from the program or from the task itself. JML (short for "Java Modelling Language") is one utensil to get us closer to reaching that point.

2 Motivation

With the rise of programmers and the need of advancement in regards to applications, there are stricter requirements and way less room for errors in today's age, unlike a decade ago; given that, needs for tools to fix mismanagement of and wrongly implemented code have arisen. As a result of that, we have taken on the task of ensuing a good introductory and compelling tutorial of JML in order to pique the interest of Java Programmers and convince them to start using it in furtherance of future large scale projects, of which code has to be proven correct before execution or launch.

3 Why use JML

JML is a notation for formally specifying the behavior and interfaces of Java classes and methods.“ [6]. Meaning it uses mathematical concepts in order to describe the behavior of a module without providing the concrete implementation details. This allows for the development of tools that assist in multiple ways throughout the entire software life cycle:

- While designing the code, one can use automated deduction and SAT-based techniques, to ensure the specifications imply the desired module behaviours[4]
- During the implementation stage static analysis helps in verifying if the described specifications are fulfilled.[3]
- Tools that seek bugs can be helped even by light-weight specifications, for example ensuring that referenced locations are non-null [3]
- Runtime assertion checking, which also helps in automating parts of testing[1]
- The generation of test units.

Finally, in critical systems, for example the Fly-by-wire life-critical system in aviation which replaces the conventional manual flight controls of an aircraft with an electronic interface [8] needs to be formally verified, as the life of the passengers and the flight crew depend on the reliability of the program. Tools based on combinations of automatic and interactive theorems proving aid in verification. Meaning, mathematically proving that an implementation is free of bugs and that it satisfies its formal specification in every possible execution .

According to [1], in the case where proprietary rights have to be protected, one could deliver the object code alongside the JML specifications to the customers. As such they would have precise documentation but not the implementation details due to the fact that JML specializations can also help with debugging. Moreover, this ensures the code can be improved in the future as clients would not depend on implementation details.

From [7] we conclude that classical documentation is a natural language, which for example is prone to cultural misinterpretations. It describes concrete implementation details such as return type and parameters used. Thanks to its well defined syntax and semantics JML offers a precise, unambiguous documentation of the behavior of Java program modules.

3.1 Benefits of JML over other specification languages

To start off, besides the ease of read, JML is made exclusively for Java, as opposed to other specification languages (languages such as VDM-SL or Z). Consequently, pure methods are allowed to be called in assertions (//@ \ensures (members < getMaxMembers() for example); together with that, JML also gives the important benefit of being able to specify exceptions, something that is vital for many systems in Java. Such features facilitate the learning process of JML for a Java developer.

More over, JML can be written directly into the source code, easing the implementation process due to how the specifications are written before the code. Although this can also be written in another ways, as specified in the following section, it will still help in specifying a given class or method.

These being said, below can be found a specification snippet of the same function, one in VDM-SL and the other in JML

```
// VDM-SL                 //JML
MethodName(x:real)r:real  /*@ requires x > 0;
pre x >=0                   @ ensures \result * \result = x && \result > 0
post r*r = x and r>=0       @*/
```

4 JML Notation and Usages

Created by Gary T. Leavens and Yoonsik Cheon, [5] JML as a whole is a behavioral interface specification language (BISL for short) designed to specify Java modules and make various logic notions easier to understand. Given how JML is made only for Java, its only use is for formal specifications regarding the behaviour of Java modules. Worth noting however is how JML only specifies the behaviour of Java modules and not of the whole program.

When and where should JML be used? To keep it simple, JML's role is to make the code clear for the other people who read it, meaning that it is strongly advised using JML at least on modules that have key requirements which, if not met, the program will throw an error. Furthermore, JML does not impose a place where its documentation should be written. [6] While it is mutually agreed upon as to writing it above the method / line of code in cause, its position is really up to the user to decide where it should go. One can use JML either as the documentation of the app or before the code itself. It is recommended however, to keep track of the fact that JML specifications can be used as an aid to proper reasoning in regards to the correctness of code, to one's use when it comes to analyzing certain properties of a design and lastly, several tools can use JML specifications to help debug and improve the code.

4.1 JML Tool Support

Pertaining to JML tools, there is a plethora of options. Starting off with the most generic option that every single code IDE offers when it comes to helping to write code, we have the type-checking and annotation parsing. Generally, one does not update informal comments in code as they develop the code further, however, a mere type-checker will find any JML assertions which refer to fields which no longer exist and all other typos made. Moving on, some tools offer more than just type-checking and parsing, something like runtime assertion checking and testing. [2] One way of checking the correctness of JML specifications is by runtime assertion checking, which stands for simply running the Java code and looking out for violations of the JML assertions. This can be done using the JML compiler *jmlc* or particularly, *jmlunit* which accomplishes the same task but it combines runtime assertion checking with unit testing. More information about it can be found in sections 3.2 and 3.3 of [2].

Another useful type of tool is the type which helps the user generate JML specifications in order to make their experience more convenient together with the aim of lowering the cost of producing JML specifications. Such tools are simple tools such as *jmlspec* which produces a simple barebone structure of a specification file from the Java source code, or the *Daikon* / *Houdini* tools which aim to detect possible invariants by observing the runtime behaviour of the program and respectively, use Java to deduce annotations for code.

The one and only downside which is present in JML and no tool can fix however, is how it does not support threads nor understands how threads work with each other so it also has 0 support for shared variables. This being said, [6] JML is currently limited to sequential specification; we say that JML specifies the sequential behavior of Java program modules.

5 Basics of JML

JML specifications work on a "design by contract" philosophy, meaning the implementation for a method or for all the methods of an abstract data type ensures an expected outcome provided certain constraints are satisfied. The assertion of these constraints, called pre and postconditions apply to methods. Constraints on all the methods of a class are called invariants.

The precondition is specified using the "requires" clause, the postcondition is specified using the "ensures" clause and the invariants are specified by "invariant". The JML notation is contained inside the java source code, in comments which start with the symbol "@" , and if they are multi line comments also end with it

single line:
"//@ <jml>"

multiline:
"/*@<jml>
@<jml>
@<jml>
@*/ "

It's a convention to use "@" to start every line in a multi line comment, but it is not necessary.

The annotation is generally located above the method it specifies, similar to invariants, which are also usually placed before the fields it mentions. Fomatting exampler example:

```
/*@ requires !account.auth;
  @ requires password == decode(account.correctPass);
  @ ensures  account.auth;
  @*/
public void login(String password)
...
```

A behavior of a method describes the states it is allowed to alter. This is specified through the description of:

- The set of states for which the method can be called.
- The locations which can be accessed or modified inside the method.
- The cases in which the method returns normally, exceptionally or it diverges

In JML, specifications can be either lightweight or heavyweight and helper or non-helper. The only difference between these is that the non-helper method or constructor doesn't require nor hold any of the object's invariants. A lightweight specification is the kind of specification where the user specifies only what they deem as important, whereas the heavyweight specifications will give default values to parts which are not specified by the user.

In order to go from a lightweight specification to a heavyweight one, the user must include at the beginning of the annotation one of the following:

```
[optional privacy modifier] behavior-keyword
where
behavior-keyword ::= behavior | normal_behavior | exceptional_behavior
```

Below can be found a lightweight specification case example

```
public abstract class Light {
protected boolean init_state, end_goal;
protected int helper;

/*@
  @     requires init_state;
  @     assignable helper;
  @     ensures end_goal;
  @*/
protected abstract int some_method();
}
```

And respectively, a heavyweight equivalent of the same example given above

```
public abstract class LightAsHeavy {
protected boolean init_state, end_goal;
protected int helper;

/*@ protected behavior
  @     requires init_state;
  @     diverges false;
  @     assignable helper;
  @     working_space \not_specified;
  @     ensures end_goal;
  @     signals_only \nothing;
```

```
  @     signals \not_specified;
  @*/
protected abstract int someMethod();
}
```

A very important detail is that there are different default values for unspecified clauses determined by the type of specification they are part of; therefore we shall now go through the possible clauses and their return values.

In the cause of **requires**, **ensures**, **signals** (which specifies what condition must hold after the given exception throws) and **when** (which allows concurrency aspects of a method or a constructor to be specified) the default value for the lightweight specification return is of type *\not_specified* while for the heavyweight variant, the return is *true*.

Similarly, for the **assignable** clause that specifies which locations outside of the method are assignable during the execution of the method; the default value for the lightweight is *\not_specified* while for the heavyweight it is *\everything*.

Forthwith after, we have the **accessible** and **callable** clauses, which specify which external locations may be read during the execution of the method and respectively, specify what methods are available to call, including the nested ones. The default return for both of these will be, in both cases, *\everything* .

Finally, we have **working_space** which determines the maximum amount of usable heap space and **signals_only** that specifies which exceptions may be thrown by the stated method. The return of these is, once again in both cases, *\not_specified* and namely, *\nothing*.

5.1 How to start implementing JML into your programs

To write JML specifications we need to place them into specially formatted Java comments. This means that a Java compiler will ignore the JML text. JML specifications have to be written in comments that either
1. begin with //@ or
2. begin with /*@ and end with @*/.
It is common practice for lines within such a block comment to have the first non-whitespace characters be a series of @ symbols, as in:

```
public class ClassName {
    /*@ requires requirement1;
      @ ensures assurance;
      @*/

    void method() { ... }
}
```

Note that a Java comment starting with @ as its very first character is a JML annotation; anything else is silently considered a Java comment. One pitfall with annotations is the following. Java annotations begin with @ (such as @Override). Thus a commented out Java annotation might well read "//@Override". This, however, is interpreted by JML tools as a JML annotation and will result in error messages. One way to avoid this ambiguity is to adopt the personal best practice of always placing a white space after the @ and before the JML keyword.
There are multiple types of JML annotations, some of those being:

- A modifier. Modifiers are single words, such as pure, that are syntactically similar to Java modifiers like public and static.
- Clauses. A JML clause begins with a keyword, such as ensures, followed by an expression or other information, and ending with a semicolon.
- Types. JML defines a number of new specification-only types, such as \real and \bigint.
- Expression tokens. These occur within JML expressions. They begin with a backslash. They can be either single words like \result or function-like, such as \old(x).

Let us take Bubble Sort as an example method. This method takes one argument, namely an array, which must not be NULL. Therefore, below the declaration of the method and above the first line of code, we have to specify a precondition using the requires keyword.

```
1 public class BubbleSort {
2  //@ requires arr != null;
3  public static void sort(int [] arr) {
4    for (int i = 0; i < arr.length; i++) {
5      for (int j = arr.length-1; j > i; j--) {
6        if (arr[j-1] < arr[j]) {
7          int tmp = arr[j];
8          arr[j] = arr[j-1];
9          arr[j-1] = tmp;
10         }
11       }
12     }
13   }
14 }
```

However, we will not stop here. This method should sort the given array, which means that we will also have to add a postcondition using the ensures keyword and due to that, we must also change the comment from a single line to a multiple line one.

```
1 public class BubbleSort {
2   /*@
3     @ requires arr != null;
4     @ ensures \forall int k; 0 <= k && k < arr.length -1;
5     @           arr[k] > arr[k+1];
6     @*/
7  public static void sort(int [] arr) {
8    for (int i = 0; i < arr.length; i++) {
9      for (int j = arr.length-1; j > i; j--) {
10       if (arr[j-1] < arr[j]) {
11          int tmp = arr[j];
12          arr[j] = arr[j-1];
13          arr[j-1] = tmp;
14         }
15       }
16     }
17   }
18 }
```

Worth noting is that we use the \forall expression which in the given example will increment k by one, as long as it satisfies the given bounds. Given that, we shall now specify the loops inside of the method

```
1 public class BubbleSort {
2   /*@
3     @ requires arr != null;
4     @ ensures \forall int k; 0 <= k && k < arr.length -1;
5     @          arr[k] > arr[k+1];
6     @*/
7  public static void sort(int [] arr) {
8    /*@ final ghost int n = arr.length;
9      @ loop_invariant 0 <= i <= n;
10     // elements up-to i are sorted
11     @ loop_invariant \forall int k; 0<= k < i;
12     @               \forall int l; k < l < n;
13     @                arr[k] >= arr[l];
14     @ decreasing n-i;
15     @*/
16 for (int i = 0; i < arr.length; i++) {
17      for (int j = arr.length-1; j > i; j--) {
18        if (arr[j-1] < arr[j]) {
19          int tmp = arr[j];
20          arr[j] = arr[j-1];
21          arr[j-1] = tmp;
22         }
```

```
23       }
24     }
25   }
26 }
```

In the scope above, we have used a ghost variable. A ghost variable is one that can only be accessed inside of the particular specification and it does not ncessarily mirror a value of the class, nor of the method. The decreasing keyword signifies that the "$n - i$" calculation will decrease after every iteration and will be different from zero.

Moving onto the second loop, we will now have our final form.

```
1 public class BubbleSort {
2   /*@
3     @ requires arr != null;
4     @ ensures \forall int k; 0 <= k && k < arr.length -1;
5     @          arr[k] > arr[k+1];
6     @*/
7  public static void sort(int [] arr) {
8    /*@ final ghost int n = arr.length;
9      @ loop_invariant 0 <= i <= n;
10     // elements up-to i are sorted
11     @ loop_invariant \forall int k; 0<= k < i;
12     @               \forall int l; k < l < n;
13     @              arr[k] >= arr[l];
14     @ decreasing n-i;
15     @*/
16 for (int i = 0; i < arr.length; i++) {
17   /*@ loop_invariant i <= j <= n-1;
18     // j-th element is always the largest
19     @ loop_invariant \forall int k; j <= k < n;
20     @               arr[j] >= arr[k];
21     // elements up-to i remain sorted
22     @ loop_invariant \forall int k; 0 <= k < i;
23     @                \forall int l; k < l < n;
24     @                arr[k] >= arr[l];
25     @ decreasing j;
26     @*/
27      for (int j = arr.length-1; j > i; j--) {
28        if (arr[j-1] < arr[j]) {
29          int tmp = arr[j];
30          arr[j] = arr[j-1];
31          arr[j-1] = tmp;
32           }
33       }
```

```
34     }
35   }
36 }
```

6 Code Example and Explanations

Below you will be presented a simple example taken from [6] where an abstract class called IntHeap will be defined. This class contains 2 methods named largest and respectively, size; both of them referring to a Heap of Integers.

```
1   package org.jmlspecs.samples.jmlrefman
2
3   public abstract class IntHeap{
4
5   //@ public model non_null int[] elements;
6
7   /*@ public normal_behaviour
8     @    requires elements.length >= 1;
9     @    assignable \nothing
10    @    ensures \result
11    @    == (\max int j;
12    @            0 <= j && j < elements.length;
13    @            elements[j]);
14    @*/
15   public abstract /*@ pure @*/ int largest();
16
17   //@ ensures \result == elements.length;
18   public abstract /*@ pure @*/ int size();
19   };
```

The interface of this Java class can be found on the 1st, 3rd, 15th and 18th lines. Line 1 specifies the package we're going to be using, line 3 the class name and lines 15 & 18 provide information regarding the methods of this class to the interface.

The behaviour of this class is specified in the JML annotations which are the lines that contain an at sign (@) at the or near the beginning of the line. The lines in cause here are 5, 7 to 14, 15, 17 and 18. While these may be treated as mere comments for the Java interpreters, however they are interpreted by JML and its tools. Notice that these can be of two types, either " //@ " or " /*@ " together with the ending of " @*/ " or " */ ", therefore, in order to start such a line, the comment must contain an at sign (@). Another thing worth noting is that there is no space between the comment and the at sign because, if it were to be like that, " // @ " would be treated as a comment and not an annotation.

6.1 In-depth code explanation

Referring to the above-given code, on the 5th line of the code given above, we can notice a field named elements which defines an array of integers. Without any modelling, one would think it is declared by mentioning its type (public, private or protected), the type of the array (int[] in our case) and lastly the name of the array. In JML however, this is declared in an annotation, therefore we are not allowed to manipulate it through code. These kind of declarations are labeled as "*specification-only*" fields via the JML model which is and should be treated as an abstraction. Now, with that, we can think of all IntHeap instances have such a field in them, fields whose values are determined by the rest of the defined class through different methods and constructors. Although, since we are talking about abstraction here, Java methods and constructors are not enough, we also need model fields. Objects of the IntHeap type do not have such fields, they are purely imaginary however, they are convenient due to the fact that they thoroughly explain the behaviour of the abstract data type (ADT for short). Worth mentioning is that these model fields do not impact the run time nor extra cost in size due to the fact that Java treats them as comments. And lastly for this line, the other annotation present here is "*non_ null*" which is pretty self explanatory invariant. This implies that in any possible state of the program, the value of the array elements must not be null.

Moving onto the 7th line, we can see another declaration of a method, therefore, we will cover the lines from 7 to 15 in one go. Line 7 simply states that this is a publicly visible method (intended to be used by clients) and the "*normal_ behaviour*" keyword which talks about the returns of the method. However, before going deeper into that, we have to look into the 2 types of method specifications, those being **heavyweight** and **lightweight**. A heavyweight specification (such as the one in cause) tells JML that the method is a complete one and needs to be respected fully, whereas a lightweight one tells JML that the method is not complete, or rather said, it only contains a part of what the specifier had in mind. With that cleared, let us continue with our example. The keyword "*normal_ behaviour*" tells JML that when the precondition(s) are met, the method must return normally, without throwing an exception; this being said, it implies the existence of a notation for the opposite outcome, which is "*exceptional_ behaviour*"

The most important part of this abstract method is the algorithm present on the lines 8 to 13 [6]; this bit imposes the method's precondition, the frame axiom and the normal post condition. On line 8, we notice the keyword "*requires*" which is the precondition of our method and it implies that the length of the array elements must be bigger or equal to 1, therefore abiding to the condition from line 5 where it states that a non-null array must be present. The frame axiom is found on the next line, namely line 9, where we notice an "*assignable*" keyword. As one would be able to already notice, this suggests that none of these elements can be assigned other values; and lastly but not least, the post-condition. The start of this can be found on line 10, up to line 13 and they state that the method "*result*" must be

equal to the biggest integer value in the elements array. Lines 11, 12 and 13 are in fact the same line just split into 3 different lines to ensure a cleaner readability and the 3 lines are:

- a declaration of some quantified variables (j in our case)
- a range predicate which implies that j must be greater or equal to 0 while not being bigger than the length of the array - 1. This is done in order to prevent an index out of bounds exception
- the element we are looking for, in our case it being a description of the elements in the array from which the maximum value is taken

Continuing, we notice that on lines 15 the keyword "*pure*" appears. This method simply tells us that the method in cause will either terminate successfully or throw an exception in using this method, allowing it to be used in assertions if needed / wanted to. Moreover, on line 17 we see a lightweight specification of what this method should achieve. To avoid repetition and keep it simple, "*ensures\result = elements.length*" just forwards the idea that upon calling this method, the result of it will be the length of the array while the "*pure*" keyword on line 18, once again, accomplishes the same thing as its other apparition on line 15.

7 Setting up JML

Due to how JML can operate only on Linux systems, in order to make it run on **Windows**, one will need to install WSL in order to "emulate" a Linux shell. To do so, start your Command Prompt with administrator privileges then run the following command "*wsl –install*" with 2 dashes. From there on, restart your PC to finish the installation and upon doing so, open cmd and write *wsl* to open it and then the following set of commands: "*sudo apt update*", "*sudo apt upgrade*" and finally "*sudo apt install*". Now we have configured the "application getter". Once done, we now have to install JML. To do so, we have to run the following commands in WSL: "*sudo apt install unzip*" in order to be able to unzip, "*cd*", "*mkdir openjml*" and "*cd openjml*" to create a folder where we will place JML. Now we will run a longer command to download JML from gGitHub "*wget https://github.com/OpenJML/OpenJML/releases/download/0.8.59/openjml-0.8.59-20211116.zip -O openjml.zip*". Once finalized, we only have to unzip, install Java and run. For that, the commands are: "*unzip openjml.zip*" in order to extract its contents from the compressed zip file, "*rm openjml.zip*" to remove the zip file, "*sudo apt install openjdk-8-jre*" to install java, "*sudo update-alternatives –config java*" with double dashes again and then you will choose "*openjdk-8*". You now have configured JML on Windows. To run it, skip to the end of this section.

To accomplish the same thing on a **Linux**-based Operating System, the process is shorter. Let us take Ubuntu for example: Firstly we open the terminal, "*cd*" and "*mkdir openjml*" + "*cd openjml*" then download the zip again via "*wget*

https://github.com/OpenJML/OpenJML/releases/download/0.8.59/openjml-0.8.59-20211116.zip -O openjml.zip", then unzip it "*unzip openjml.zip*" and remove it "*rm openjml.zip*". Finally, install jdk-8 via "*sudo apt install openjdk-8-jre*" and "*sudo update-alternatives –config java*" with double dashes and then choose "*openjdk-8*"

Lastly, running JML can be done easily, via running the following command "*java -jar openjml/openjml.jar -esc {[path1, path2, ...]}*" where path1,path2,... stand for a list of paths to java files

8 Related Work

The main piece of work that has proved its use to us is the JML reference manual released in 31st of May 2013. When it comes to JML, this reference manual acts as the encyclopedia of JML, detailing every single aspect of it, up to the last detail. Not only did this prove useful when researching for examples but also for explanations of absolutely anything one can think of asking.

Another descriptive and well-approached overview over the topic in question together with its applications was done by researchers at the Radboud Nijmegen University. The paper in question is titled "An Overview of JML tools and applications", it came out on 28th of September 2004; and brings an informative view to the reader about what JML is about, how to use it, its efficiency, its wide range of tools and gives a small glimpse of other existing applications that implement JML.

Both works named above have been of great use in regards to self-documenting ourselves to a high enough level that allows us to explain JML well enough in order for the newly-introduced to understand how JML works and why they should consider using it

9 Conclusion

To summarize, Java Modelling Language (JML for short) is a behavioral interface specification language designed to specify Java interfaces, classes and methods and while it does not provide support for threads, it does provide with specifications for the sequential behavior of Java program modules. Albeit there are other specification languages, JML is recommended due to its ease of readability, due to how it can be written directly into the source code, due to it being able to specify exceptions, something that is vital for many systems in java, and lastly, due to its good tool support.

10 Future Work

In regards to future work, the plan is to implement more abstract concepts of JML into daily code by improving our comprehension over it and together with that, giving it a more appealing look together with making it easier for non experienced people to deduce, set up and start using JML.

References

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: Automated testing based on Java predicates". In: *ACM SIGSOFT Software Engineering Notes* 27.4 (2002), pp. 123–133.

[2] Lilian Burdy et al. "An overview of JML tools and applications". In: *International journal on software tools for technology transfer* 7.3 (2005), pp. 212–232.

[3] John Hatcliff et al. "Behavioral interface specification languages". In: *ACM Computing Surveys (CSUR)* 44.3 (2012), pp. 1–58.

[4] Daniel Jackson. "Alloy: a language and tool for exploring software designs". In: *Communications of the ACM* 62.9 (2019), pp. 66–76.

[5] Gary T Leavens, Albert L Baker, and Clyde Ruby. "JML: A notation for detailed design". In: *behavioral specifications of Businesses and Systems*. Springer, 1999, pp. 175–188.

[6] Gary T Leavens et al. *JML reference manual*. 2008.

[7] Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. "Eiffel: Object-oriented design for software engineering". In: *European Software Engineering Conference*. Springer. 1987, pp. 221–229.

[8] JP Sutherland. *Fly-by-wire flight control systems*. Tech. rep. AIR FORCE FLIGHT DYNAMICS LAB WRIGHT-PATTERSON AFB OH, 1968.