

---

# Formal Specification of an API

Jens-Fabian Goetzmann

<b>1</b>	<b>Motivation</b> .....	2
<b>2</b>	<b>Introduction to the Area of Research</b> .....	2
2.1	Formal Specification of Object Oriented Software Systems .....	3
2.2	JavaCard .....	4
<b>3</b>	<b>Summary of the Article</b> .....	5
3.1	Introduction to JML .....	5
3.2	Actual Specification .....	6
3.3	Benefits of the Specification .....	7
3.4	Possible Applications for the Specification .....	8
<b>4</b>	<b>Comparison to other Approaches</b> .....	9
4.1	Informal Specification .....	9
4.2	Reference Implementation .....	9
4.3	Eiffel's Design by Contract .....	9
4.4	Larch .....	10
4.5	The Z notation .....	10
4.6	Java Language Features .....	10
4.7	Fully Automatic Code Verification .....	11
<b>5</b>	<b>Conclusion</b> .....	11
	<b>References</b> .....	12

## 1 Motivation

In this section, I will discuss the main reasons why a software system in general and an API in particular should be *formally specified*.

There are numerous reasons to formally specify software systems. A few of them are especially important for this article:

### *Unambiguous Documentation*

A good formal specification is a very useful kind of documentation: If it is at a well-chosen level of abstraction, it is easily understandable while not being ambiguous like natural language always is. In the case of an API, this is a very strong reason to develop a formal specification, because an API is destined to be used by different developers on both the API user and implementer side, who all have to fully and accurately understand the way that the API is supposed to work.

### *Automatic Processing*

In contrast to an informal specification, a formal specification can be automatically processed in numerous ways, for example to generate runtime or design tests or to generate human-readable documentation.

### *Formal Proofs*

A very interesting area of application is the possibility to conduct formal proofs and automatic program verification, where certain properties of both the API and applications using the API may be proved. Although program verification may be done without a formal specification, the specification can be a valuable resource for the verification and can on the other hand itself be verified.

### *Increasing Software Reliability and Correctness*

The overall goal of formal specifications is to increase the *reliability* and *correctness* of the software that is specified. In the case of an API, this means that the API is both implemented correctly (i.e. it behaves the way it was supposed to by the original developers) and used correctly (i.e. that all conditions that have to be taken care of when using classes or methods from the API are fulfilled). This in turn eliminates many frequent causes for errors.

## 2 Introduction to the Area of Research

In this section, I will give a short introduction to both the formal specification of object oriented software systems and to the JavaCard API, which is the system specified in [PvdBJ01].

## 2.1 Formal Specification of Object Oriented Software Systems

In (strictly<sup>1</sup>) object oriented software systems, the only thing that has to be specified are classes with their members: properties and methods. These two types of class members are fundamentally different and thus have different means of specification that are detailed in this section.

### Methods

#### *Preconditions*

*Preconditions* for methods state conditions that have to be fulfilled by the caller of the method. They typically include conditions such as the domain for the arguments, certain dependencies among the arguments or even requirements on the state of the class instance – although they should only refer to the publicly visible state of the class, because otherwise there is no way for the caller to ensure that the preconditions are met indeed.

If the precondition is not fulfilled by the caller, then the method can not guarantee any specific behavior or result.

A simple example of a precondition would be a setter function `setColor(Color newColor)`, for which the precondition might simply be that `newColor` may not be a `null` reference.

#### *Postconditions*

*Postconditions* for methods state conditions that will be fulfilled after the execution of the method is finished (if the preconditions have been fulfilled when the method was called). In contrast to the preconditions, the implementer of the method is responsible for the fulfillment of these conditions. Good postconditions fully specify the result that a method should achieve.

For the simple example mentioned above, the postcondition could simply be that the `color` property of the class instance equals the `newColor` argument of the method.

#### *Benefits and Obligations*

The system of pre- and postconditions offers benefits and obligations to both implementer and caller of the method.

The *implementer* is obliged to implement the method in a way that the postcondition is held after the execution of the method, while having the benefit of being guaranteed correct input data and fulfilment of other prerequisites.

The *caller* is obliged to make sure that all prerequisites for the method are fulfilled, while having the benefit of getting the desired result from the execution of the method.

<sup>1</sup>Java (and JavaCard) may be called a *strictly* object oriented language, because no language constructs such as functions and variables are allowed to live outside of classes. C++, in contrast, still allows global functions and variables and thus is not a strictly object oriented language.

## Properties

### *Class Invariants*

The basic idea of *class invariants* is the question: “How does a *valid* instance of this class look like?” Class invariants specify certain conditions that have to be fulfilled by *all* class instances in *all* visible states. “Visible states” means that these class invariants may be broken during the execution of class methods, but all public methods must ensure that the class invariant is restored upon exiting. They are very useful for specifying design decisions that may not be obvious if they are not explicitly stated.

A simple example of a class invariant for a `Vehicle` class may be: “The value of the `owner` property may not be a `null` reference.”

### *History Constraints*

While class invariants give conditions for the state of a class instance, *history constraints* do so for state transitions. They describe what constraints apply to the new state of each state transition based on the old state. The design decisions and behavior that can be expressed with history constraints is normally even more implicit than the ones specified by class invariants.

For the simple example of a `Vehicle` class, a class invariant may be: “For each state transition, the new value of the `mileage` property must be greater than or equal to the old value.”

## 2.2 JavaCard

### An Introduction to Java Card

JavaCard is an API (Application Programming Interface) for developing programs that are able to be run on smart cards. It only consists of about 50 classes, which makes it eligible for a (supplementary) formal specification.

The JavaCard API consists of a small subset of the J2SE<sup>2</sup> API, augmented with special classes for smart card programming. It was developed by Sun Microsystems.

### Available Specification

Sun Microsystems provides the following specification for Java Card:

- An *informal specification* in Javadoc style in [Sun00b], i.e. a description of all the classes, methods and properties and their behavior (where applicable) in natural language, i.e. English.

---

<sup>2</sup>Java 2 Standard Edition.

- A *reference implementation* in [Sun00a]. The JavaCard API is meant to be implemented by each smart card vendor for his or her own smart cards, so the reference implementation can already be seen as a formal specification – though one on a very low level of abstraction. However, this reference implementation contains methods declared as `native`, meaning that there is no reference implementation for them.

### 3 Summary of the Article

In this section, I am going to present a short summary of [PvdBJ01], which was the primary source of information for this paper and the accompanying presentation.

#### 3.1 Introduction to JML

The article starts with a quite verbose introduction to JML. JML is short for *Java Modeling Language* and was developed at Iowa State University [LBR99]. It borrows features from Eiffel – an object oriented programming language which contains means for formal specification directly in the language – and Larch – a “framework” for formal specification which can be applied to a number of languages, e.g. C. A brief overview over features and possibilities of JML can be found in [LLP<sup>+</sup>00].

#### Unique Features of JML

JML has a few unique features that make it a good choice for formally specifying Java software systems:

##### *Usage of Java Syntax*

JML uses Java syntax wherever possible. This is a difference to many formal specification languages such as Larch or the Z notation that make heavy use of mathematical notation. Using Java syntax in the formal specification language makes it easier for programmers familiar with Java to learn JML. However, it restricts the use of JML to Java as well.

##### *High Level of Detail*

JML offers means of specification that allow a quite high level of detail. All features detailed in section 2.1 are available, also notable is the existence of *quantors*, that can be used to specify that a certain condition has to be fulfilled “for all” members of a set or that there must “exist” at least one member in a set that fulfills a certain condition.

*Full Integration of Inheritance*

Because Java allows class inheritance, it is quite natural that JML must also integrate this concept into the specification language. Most notably, the following rules apply to methods that are overwritten by a subclass:

- The overwriting method may only *weaken* any preconditions, i.e. enlarge the domain of the method.
- The overwriting method may only *strengthen* any postconditions, i.e. make more guarantees about the result of the method.

The opposite is not allowed.

*Model Based Specification*

JML offers the possibility to define *model-only* classes, methods and properties. Some very basic concepts such as sequences, sets etc. are already present in JML as model-only classes, i.e. classes that have a specification of their behavior but no implementation. These facilities can be used to create a very sophisticated specification with a high level of abstraction.

**Specifying Software Systems with JML**

JML has two possibilities to actually specify Java code: The first possibility features specially crafted comments inside the original Java source code that resemble Javadoc comments, e.g. `//@requires newColor != null`. Instead of doing this, one may also put down the specification in `.jml` files that contain the class specifications just like the respective `.java` files, but without the actual Java implementation of the methods.

**3.2 Actual Specification**

As already mentioned above, the specification described in the article was a *supplementary* specification of the API; It was written by a different team than the one that developed the API. Also already mentioned was the existing specification of the API that could be used in order to write the formal specification: An informal specification in natural language and a reference implementation.

The formal specification that is described in the article focuses on giving explicit preconditions for all methods in the API. A special emphasis of the work lies on *exceptions* by defining method *behaviors* (i.e. pairs of pre- and postconditions) that state which exceptions

- will definitely be thrown,
- may be thrown or
- may not be thrown

by a method.

The specification given in the article also adds class invariants where it is useful for stating properties of class instances that can help one understand the conditions under which methods may be forced to throw an exception.

Unfortunately, the specification that was the result of the article is no longer available. An extended version of the specification can still be found online,<sup>3</sup> but it is no longer worked on.

### *Possible Extensions*

The specification originally given by the authors of the paper could easily be extended. The most natural extension is the addition of postconditions (other than the ones stating which exceptions could be thrown). Postconditions explicitly and formally specify the results of the methods and are therefore an important contribution to a full formal specification of the API. Another extension could be providing more details under the circumstances for each specific extension, other than simply stating that one of a given set of exceptions may be thrown.

However, since the time of the writing of the article, the authors have already implemented these extensions, and [PH06] already contains the extensions proposed here and can be deemed a quite complete specification of the JavaCard API.

### **3.3 Benefits of the Specification**

The specification provides benefits to both users (who wish to use the API to write Java programs eligible to be run on smart cards) and implementers of the API (who wish to make their smart cards be able to run JavaCard programs).

For the latter, the specification is a lot clearer and more detailed than the informal specification, while still being easy to understand. It also explicitly states many design decisions by the original developers that are not contained in the original informal specification and only implicitly contained in the reference implementation.

For the former, the specification gives a better description of the behavior of the classes and methods of the API than the informal specification could give. Again, conditions that have to be fulfilled by the user of the API are stated very clearly and easily understandable in the formal specification, and many conditions that arise from design decisions not mentioned in the informal specification, but only in the reference implementation (which a user of the API will typically never have a look at), are explicitly expressed in the formal specification.

---

<sup>3</sup>On Erik Poll's personal web site [PvdBJ02] or an updated version on [PH06].

### 3.4 Possible Applications for the Specification

The article describes three different projects that can make use of a given formal specification for a software system:

- Generation of run-time checks,
- Extended static checking and
- Automatic code verification.

I will discuss them a bit more in-depth in this section.

#### *Generation of Run-Time Checks*

The generation of run-time checks has been a project at the Iowa State University and is described in detail in [Bho00]. The focus of this project was to generate checks for all preconditions as Java source code and inject them into the source files containing the Java classes prior to the compilation. When the program was run, the fulfillment of the preconditions could then be tested for each method call, and if the preconditions did not hold for a call, then an exception would be thrown – just like in Eiffel when a contract is not met.

The run-time checks could also be enabled or disabled in runtime. The primary goal of these run-time checks was not to be included in production code, but to be used in the developing and debugging phase to find problems the moment they occur.

#### *Extended Static Checking*

Extended static checking of Java Code was a project at the Compaq System Research Center. The product that resulted from their efforts is called ESC/Java (Extended Static Checker for Java) and is described in detail in [FLL<sup>+</sup>02]. The idea of extended static checking is to extend the static checks performed by a compiler. To achieve this, Java code is turned into theorems in first order logic, and then certain properties (for example the given specification of a method) of the code are tried to be automatically proved. However, to maintain scalability, ESC/Java only looks at one method at a time and all other methods are assumed to work according to their specification. This makes this method suitable also for larger projects.

#### *Automatic Code Verification*

Automatic code verification is an approach where source code is converted to some kind of logic and then fed to a (semi-)automatic theorem prover, which then tries to prove some properties of the code. The difference to the extended static checking is that the entire source code is processed at once, not in method chunks.

The article references the LOOP tool, which was developed at the University of Nijmegen and the Technical University of Dresden and is described in-depth in [JvdBH<sup>+</sup>98]. The LOOP tool can take a JML specification as a proof obligation and thus prove (or refute) that a given implementation meets the formal specification.

## 4 Comparison to other Approaches

In this section, I will give a short overview over other possible approaches to specifying an object oriented software system like the JavaCard API.

### 4.1 Informal Specification

An informal specification in a natural language could be seen as an alternative to a formal specification. In the case of the JavaCard API, a structured informal specification was given.

The main advantage of an informal specification is the fact that it is easily understandable.

The disadvantages of informal specifications are numerous:

- They are not international.
- All specifications in a natural language are inherently ambiguous.
- They are processable by machines only to a limited extend.
- Many design decisions are very likely not to be documented in an informal specification.

### 4.2 Reference Implementation

A reference implementation in a programming language is a formal specification at a very low level of abstraction. For the JavaCard API, a reference implementation was given.

The main advantage of a reference implementation is that it is ensured to fully cover the behavior of the software system.

The disadvantages of a reference implementation are the following:

- It is not very well suited as a documentation.
- It may contain errors in the design (for example the famous merge-sort-bug).
- Methods declared as `native` have no specification.

### 4.3 Eiffel's Design by Contract

The object oriented language Eiffel contains features that allow formal specification of classes and methods directly in the source code. It could be seen as an alternative to the use of Java and JML.

The main advantage is the direct integration of the formal specification methods in the language itself. It therefore does not have to feature specially crafted comments, and conditions can be dynamically checked at run-time without the need for an additional tool such as the one described in section 3.4.

The disadvantages lie in the limitations of Eiffel's possibilities of specification: There are no quantors, history constraints or model classes. The biggest

disadvantage, however, is the language itself: It would not be possible to specify a Java API in Eiffel, and the amount of software written in Java exceeds the amount of software written in Eiffel by far.

#### 4.4 Larch

An alternative to using JML for the specification could be to use a specification language from the Larch family. Although Larch is a powerful specification language which features rich possibilities of specification just like JML, it has a few disadvantages when compared to JML:

- It features a full-blown mathematical notation which is more difficult to learn than JML's Java-like syntax.
- It may not be embedded into the source code like JML.
- It is no longer under active development.

#### 4.5 The Z notation

The Z notation is another powerful formal specification language. The same disadvantages as mentioned for Larch also apply here except for the lack of active development.

#### 4.6 Java Language Features

Java contains a few features that might be seen as a possible replacement for a formal specification. The two most obvious features in this case are the `assert` statement and the `throws` clause.

##### *The `assert` Statement*

The `assert` statement is a part of Java since Java 1.4, which was released in 2002 and thus it was no alternative at the time the article was written. The `assert` statement is a facility to include conditions that have to be fulfilled in arbitrary positions of the source code, checking of these assertions can be dynamically enabled or disabled without the need for a re-compilation.

The key advantage of using the `assert` statement is its direct integration into the Java Virtual Machine, which in turn leads to a much lower lack of performance when assertion checking is disabled when compared to the approach taken in [Bho00].

For the sake of a formal specification, Java assertions have quite a few disadvantages:

- They are not well suited for documentation.
- They can not be explicitly labelled as pre- or postconditions.
- They can not directly express quantors, class invariants or history constraints.

*The throws Clause*

The `throws` clause is a means to specify which exceptions a method may throw. Given the focus of [PvdBJ01] on exceptions, the `throws` should be evaluated as an alternative.

The main advantage of the `throws` clause is the fact that programmers of a method using the method for which the `throws` clause is given are *forced* to either catch the expression or issue a `throws` clause themselves. This leads to better code as known possible sources of errors are guaranteed to be covered.

The disadvantages from a specification point of view are the low documentation nature and the lack of a specification of the conditions under which a specific exception may be thrown.

**4.7 Fully Automatic Code Verification**

For the sake of proving certain properties of the code, one may also consider a fully automatic code verification without any additional information from a formal specification.

The main advantage of this approach is that mistakes or carelessness when the specification is elaborated is prevented.

The main disadvantages are:

- It is very difficult and time and resource consuming to carry out.
- It is not suited for documentation.
- It is not possible to verify `native` code.

**5 Conclusion**

The formal specification of an API in general and of the JavaCard API – that is destined to be implemented many times by different smart card vendors all over the world – in particular was shown to be a valuable addition to the existing documentation. It also offers possibilities for automatic processing such as the generation of run-time checks, extended static checking or verification that is just impossible without a formal specification.

The possible extensions of the specification have since already been implemented and the current specification can be seen as quite complete. The hope of the authors for an integration of the JML specification into the official JavaCard specification has not been fulfilled, however.

JML itself has since been extended and has established itself as a promising behavioral specification language. The integration of JML with ESC/Java and LOOP has also been completed, although the LOOP tool itself is no longer under active development. JML itself is still a living project and might be used as a valuable method in the process of designing Java software systems of all types.

## References

- [Bho00] Abhay Bhorkar. A Run-time Assertion Checker for Java Using JML. Technical Report #00-08, Department of Computer Science, Iowa State University, May 2000.
- [FLL<sup>+</sup>02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.
- [JvdBH<sup>+</sup>98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java Classes (Preliminary Report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
- [LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report #98-06c, Department of Computer Science, Iowa State University, January 1999.
- [LLP<sup>+</sup>00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.
- [PH06] Erik Poll and Engelbert Hubbers. Formal Interface Specifications for the JavaCard API 2.1.1, Updated Version. <http://www.sos.cs.ru.nl/research/escjava/esc2jcap.html>, 2006. Accessed: 2007-12-06.
- [PvdBJ01] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In *Proc. of the fourth working conference on smart card research and advanced applications*, pages 135–154, 2001.
- [PvdBJ02] Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal Interface Specifications for the JavaCard API 2.1.1. [http://www.cs.ru.nl/~erikpoll/papers/jc211\\_specs.html](http://www.cs.ru.nl/~erikpoll/papers/jc211_specs.html), 2002. Accessed: 2007-12-06.
- [Sun00a] Sun Microsystems Inc. Java Card Development Kit. [http://java.sun.com/products/javacard/dev\\_kit.html](http://java.sun.com/products/javacard/dev_kit.html), 2000. Accessed: 2007-12-15.
- [Sun00b] Sun Microsystems Inc. Java Card Platform Specification. <http://java.sun.com/products/javacard/specs.html>, 2000. Accessed: 2007-12-15.